

IAR C/C++ Compiler

Reference Guide

for Atmel® Corporation's
AVR Microcontroller Family



COPYRIGHT NOTICE

Copyright © 1996–2011 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.

Atmel is a registered trademark of Atmel® Corporation. AVR is a trademark of Atmel® Corporation.

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

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

EDITION NOTICE

Sixth edition: June 2011

Part number: CAVR-6

This guide applies to version 6.x of IAR Embedded Workbench® for AVR.

Internal reference: M10, csrct2010.1, V_100308, IMAE.

Brief contents

Tables	15
Preface	17
Part 1. Using the compiler	25
Getting started	27
Data storage	37
Functions	47
Placing code and data	53
The DLIB runtime environment	71
The CLIB runtime environment	109
Assembler language interface	117
Using C	135
Using C++	143
Efficient coding for embedded applications	159
Part 2. Reference information	179
External interface details	181
Compiler options	187
Data representation	231
Extended keywords	245
Pragma directives	265
Intrinsic functions	283
The preprocessor	293

Library functions	301
Segment reference	317
Implementation-defined behavior	333
Implementation-defined behavior for C89	349
Index	363

Contents

Tables	15
Preface	17
Who should read this guide	17
How to use this guide	17
What this guide contains	18
Other documentation	19
User and reference guides	19
The online help system	20
Further reading	20
Web sites	21
Document conventions	21
Typographic conventions	22
Naming conventions	22
 Part I. Using the compiler	25
Getting started	27
IAR language overview	27
Supported AVR devices	28
Building applications—an overview	28
Compiling	28
Linking	29
Basic project configuration	29
Processor configuration	30
Memory model	31
Size of double floating-point type	32
Optimization for speed and size	32
Runtime environment	32
Special support for embedded systems	34
Extended keywords	34
Pragma directives	34

Predefined symbols	35
Special function types	35
Accessing low-level features	35
Data storage	37
Introduction	37
Different ways to store data	37
Memory models	38
Specifying a memory model	38
Memory types	39
Using data memory attributes	39
Pointers and memory types	41
Structures and memory types	42
More examples	42
C++ and memory types	43
Auto variables—on the stack	44
The stack	44
Dynamic memory on the heap	45
Functions	47
Function-related extensions	47
Function storage	47
Using function memory attributes	47
Primitives for interrupts, concurrency, and OS-related programming	48
Interrupt functions	48
Monitor functions	49
C++ and special function types	51
Placing code and data	53
Segments and memory	53
What is a segment?	53
Placing segments in memory	54
Customizing the linker configuration file	54

Data segments	57
Static memory segments	57
The stack	61
The return address stack	63
The heap	64
Located data	65
User-defined segments	65
Code segments	65
Interrupt and reset vectors	66
Functions	66
Compiler-generated segments	66
C++ dynamic initialization	66
Verifying the linked result of code and data placement	67
Segment too long errors and range errors	67
Linker map file	67
Managing multiple address spaces	68
The DLIB runtime environment	71
Introduction to the runtime environment	71
Runtime environment functionality	71
Setting up the runtime environment	72
Using a prebuilt library	73
Choosing a library	74
Customizing a prebuilt library without rebuilding	75
Choosing formatters for printf and scanf	75
Choosing a printf formatter	76
Choosing a scanf formatter	77
Application debug support	78
Including C-SPY debugging support	78
The debug library functionality	79
The C-SPY Terminal I/O window	79
Low-level functions in the debug library	80
The C-SPY Terminal I/O window	80

Overriding library modules	81
Building and using a customized library	82
Setting up a library project	83
Modifying the library functionality	83
Using a customized library	83
System startup and termination	84
System startup	84
System termination	86
Customizing system initialization	87
__low_level_init	87
Modifying the file cstartup.s90	88
Library configurations	88
Choosing a runtime configuration	88
Standard streams for input and output	89
Implementing low-level character input and output	89
Configuration symbols for printf and scanf	91
Customizing formatting capabilities	92
File input and output	92
Locale	93
Locale support in prebuilt libraries	93
Customizing the locale support	94
Changing locales at runtime	95
Environment interaction	95
The getenv function	96
The system function	96
Signal and raise	96
Time	97
Strtod	97
Pow	97
Assert	98
Heaps	98
Managing a multithreaded environment	98
Multithread support in the DLIB library	99
Enabling multithread support	100

Changes in the linker configuration file	103
Checking module consistency	103
Runtime model attributes	104
Using runtime model attributes	104
Predefined runtime attributes	105
User-defined runtime model attributes	107
The CLIB runtime environment	109
Using a prebuilt library	110
Choosing a library	110
Input and output	111
Character-based I/O	111
Formatters used by printf and sprintf	112
Formatters used by scanf and sscanf	113
System startup and termination	113
System startup	114
System termination	114
Overriding default library modules	114
Customizing system initialization	114
C-SPY runtime interface	115
The debugger Terminal I/O window	115
Termination	115
Checking module consistency	115
Assembler language interface	117
Mixing C and assembler	117
Intrinsic functions	117
Mixing C and assembler modules	118
Inline assembler	119
Calling assembler routines from C	120
Creating skeleton code	120
Compiling the code	121
Calling assembler routines from C++	122
Calling convention	123
Choosing a calling convention	124

Function declarations	125
Using C linkage in C++ source code	125
Preserved versus scratch registers	126
Function entrance	127
Function exit	130
Restrictions for special function types	130
Examples	131
Function directives	132
Call frame information	133
CFI directives	133
Creating assembler source with CFI support	133
Using C	135
C language overview	135
Extensions overview	136
Enabling language extensions	137
IAR C language extensions	138
Extensions for embedded systems programming	138
Relaxations to Standard C	140
Using C++	143
Overview	143
Embedded C++	143
Extended Embedded C++	144
Enabling support for C++	144
EC++ feature descriptions	145
Using IAR attributes with Classes	145
Function types	148
New and Delete operators	149
Using static class objects in interrupts	150
Using New handlers	150
Templates	151
Debug support in C-SPY	151
EEC++ feature description	151
Templates	151

Variants of cast operators	154
Mutable	155
Namespace	155
The STD namespace	155
Pointer to member functions	155
C++ language extensions	156
Efficient coding for embedded applications	159
Selecting data types	159
Locating strings in ROM, RAM or flash	160
Using efficient data types	160
Floating-point types	161
Memory model and memory attributes for data	162
For details about the memory types, see <i>Memory types</i> , page 39. ...	162
Using the best pointer type	162
Anonymous structs and unions	162
Controlling data and function placement in memory	164
Data placement at an absolute location	165
Data and function placement in segments	166
Controlling compiler optimizations	168
Scope for performed optimizations	168
Multi-file compilation units	168
Optimization levels	169
Speed versus size	170
Fine-tuning enabled transformations	170
Facilitating good code generation	172
Writing optimization-friendly source code	173
Saving stack space and RAM memory	173
Function prototypes	173
Integer types and bit negation	174
Protecting simultaneously accessed variables	175
Accessing special function registers	176
Non-initialized variables	176

Part 2. Reference information	179
External interface details	181
Invocation syntax	181
Compiler invocation syntax	181
Passing options	181
Environment variables	182
Include file search procedure	182
Compiler output	183
Diagnostics	185
Message format	185
Severity levels	185
Setting the severity level	186
Internal error	186
Compiler options	187
Options syntax	187
Types of options	187
Rules for specifying parameters	187
Summary of compiler options	190
Descriptions of compiler options	193
Data representation	231
Alignment	231
Alignment on the AVR microcontroller	231
Byte order	232
Basic data types	232
Integer types	232
Floating-point types	235
Pointer types	237
Function pointers	237
Data pointers	237
Casting	238
Structure types	240
Alignment	240

General layout	240
Type qualifiers	240
Declaring objects volatile	240
Declaring objects volatile and const	242
Declaring objects const	242
Data types in C++	243
Extended keywords	245
General syntax rules for extended keywords	245
Type attributes	245
Object attributes	248
Summary of extended keywords	249
Descriptions of extended keywords	250
Pragma directives	265
Summary of pragma directives	265
Descriptions of pragma directives	266
Intrinsic functions	283
Summary of intrinsic functions	283
Descriptions of intrinsic functions	284
The preprocessor	293
Overview of the preprocessor	293
Descriptions of predefined preprocessor symbols	294
Descriptions of miscellaneous preprocessor extensions	299
Library functions	301
Library overview	301
Header files	301
Library object files	302
Alternative more accurate library functions	302
Reentrancy	302
The longjmp function	303
IAR DLIB Library	303
C header files	303

C++ header files	304
Library functions as intrinsic functions	307
Added C functionality	307
Symbols used internally by the library	309
IAR CLIB Library	309
Library definitions summary	310
AVR-specific library functions	310
Specifying read and write formatters	311
Segment reference	317
Summary of segments	317
Descriptions of segments	318
Implementation-defined behavior	333
Descriptions of implementation-defined behavior	333
Implementation-defined behavior for C89	349
Descriptions of implementation-defined behavior	349
Index	363

Tables

1: Typographic conventions used in this guide	22
2: Naming conventions used in this guide	22
3: Summary of processor configuration	31
4: Memory model characteristics	39
5: Memory types and their corresponding memory attributes	40
6: Function memory attributes	47
7: XLINK segment memory types	54
8: Memory layout of a target system (example)	55
9: Memory types with corresponding segment groups	58
10: Segment name suffixes	58
11: Prebuilt libraries	74
12: Customizable items	75
13: Formatters for printf	76
14: Formatters for scanf	77
15: Levels of debugging support in runtime libraries	78
16: Functions with special meanings when linked with debug library	80
17: Library configurations	88
18: Descriptions of printf configuration symbols	91
19: Descriptions of scanf configuration symbols	92
20: Low-level I/O files	93
21: Heaps and memory types	98
22: Library objects using TLS	99
23: Macros for implementing TLS allocation	102
24: Example of runtime model attributes	104
25: Predefined runtime model attributes	105
26: Runtime libraries	110
27: Registers used for passing parameters	128
28: Passing parameters in registers	129
29: Registers used for returning values	130
30: Language extensions	137
31: Compiler optimization levels	169

32: Compiler environment variables	182
33: Error return codes	184
34: Compiler options summary	190
35: Accessing variables with aggregate initializers	208
36: Integer types	232
37: Floating-point types	235
38: Function pointers	237
39: Data pointers	237
40: <code>size_t</code> typedef	239
41: <code>ptrdiff_t</code> typedef	239
42: Type of volatile accesses treated in a special way	242
43: Extended keywords summary	249
44: Pragma directives summary	265
45: Intrinsic functions summary	283
46: Predefined symbols	294
47: Traditional Standard C header files—DLIB	304
48: Embedded C++ header files	305
49: Standard template library header files	305
50: New Standard C header files—DLIB	306
51: IAR CLIB Library header files	310
52: Segment summary	317
53: Message returned by <code>strerror()</code> —IAR DLIB library	348
54: Message returned by <code>strerror()</code> —IAR DLIB library	359
55: Message returned by <code>strerror()</code> —IAR CLIB library	362

Preface

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

Who should read this guide

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

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

How to use this guide

When you start using the IAR C/C++ Compiler for AVR, 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 Project Management and Building 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 to store data in memory, focusing on the different memory models and memory type attributes.
- *Functions* gives a brief overview of function-related extensions—mechanisms for controlling functions—and describes some of these mechanisms in more detail.
- *Placing code and data* describes the concept of segments, introduces the linker 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 introducing the file `cstartup`, how to use modules for locale, and file I/O.
- *The CLIB runtime environment* gives an overview of the CLIB runtime libraries and how to customize them. The chapter also describes system initialization and introduces the file `cstartup`.
- *Assembler language interface* contains information required when parts of an application are written in assembler language. This includes the calling convention.
- *Using C* gives an overview of the two supported variants of the C language and an overview of the compiler extensions, such as extensions to Standard C.
- *Using C++* gives an overview of the two levels of C++ support: The industry-standard EC++ and IAR Extended EC++.
- *Efficient coding for embedded applications* gives hints about how to write code that compiles to efficient code for an embedded application.

Part 2. Reference information

- *External interface details* provides reference information about how the compiler interacts with its environment—the invocation syntax, methods for passing options to the compiler, environment variables, the include file search procedure, and the different types of compiler output. The chapter also describes how the compiler's diagnostic system works.
- *Compiler options* explains how to set options, gives a summary of the options, and contains detailed reference information for each compiler option.

- *Data representation* describes the available data types, pointers, and structure types. This chapter also gives information about type and object attributes.
- *Extended keywords* gives reference information about each of the AVR-specific keywords that are extensions to the standard C/C++ language.
- *Pragma directives* gives reference information about the pragma directives.
- *Intrinsic functions* gives reference information about functions to use for accessing AVR-specific low-level features.
- *The preprocessor* gives a brief overview of the preprocessor, including reference information about the different preprocessor directives, symbols, and other related information.
- *Library functions* gives an introduction to the C or C++ library functions, and summarizes the header files.
- *Segment reference* gives reference information about the compiler's use of segments.
- *Implementation-defined behavior* describes how the compiler handles the implementation-defined areas of the C language standard.

Other documentation

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

USER AND REFERENCE GUIDES

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

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

- Programming for the IAR Assembler for AVR, see the *IAR Assembler Reference Guide for AVR*.
- Using the IAR DLIB Library, see the *DLIB Library Reference information*, available in the online help system.
- Using the IAR CLIB Library, see the *IAR C Library Functions Reference Guide*, available in the online help system.
- Porting application code and projects created with a previous version of the IAR Embedded Workbench for AVR, see the *IAR Embedded Workbench® Migration Guide for AVR®*.
- Developing safety-critical applications using the MISRA C guidelines, see the *IAR Embedded Workbench® MISRA C:2004 Reference Guide* or the *IAR Embedded Workbench® MISRA C:1998 Reference Guide*.

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

THE ONLINE HELP SYSTEM

The context-sensitive online help contains:

- Comprehensive information about debugging using the IAR C-SPY® Debugger
- Reference information about the menus, windows, and dialog boxes in the IDE
- Compiler reference information
- Keyword reference information for the DLIB library functions. To obtain reference information for a function, select the function name in the editor window and press F1. Note that if you select a function name in the editor window and press F1 while using the CLIB library, you will get reference information for the DLIB library.

FURTHER READING

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

- Barr, Michael, and Andy Oram, ed. *Programming Embedded Systems in C and C++*. O'Reilly & Associates.
- Harbison, Samuel P. and Guy L. Steele (contributor). *C: A Reference Manual*. Prentice Hall.
- Josuttis, Nicolai M. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley.
- Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall.
- Labrosse, Jean J. *Embedded Systems Building Blocks: Complete and Ready-To-Use Modules in C*. R&D Books.

- Lippman, Stanley B. and Josée Lajoie. *C++ Primer*. Addison-Wesley.
- Mann, Bernhard. *C für Mikrocontroller*. Franzis-Verlag. [Written in German.]
- Meyers, Scott. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley.
- Meyers, Scott. *More Effective C++*. Addison-Wesley.
- Meyers, Scott. *Effective STL*. Addison-Wesley.
- Stroustrup, Bjarne. *The C++ Programming Language*. Addison-Wesley.
- Stroustrup, Bjarne. *Programming Principles and Practice Using C++*. Addison-Wesley.
- Sutter, Herb. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley.

WEB SITES

Recommended web sites:

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

Document conventions

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

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

TYPOGRAPHIC CONVENTIONS

This guide uses the following typographic conventions:





Style	Used for
computer	<ul style="list-style-type: none">• Source code examples and file paths.• Text on the command line.• Binary, hexadecimal, and octal numbers.
parameter	A placeholder for an actual value used as a parameter, for example <i>filename.h</i> where <i>filename</i> represents the name of the file.
[option]	An optional part of a command.
[a b c]	An optional part of a command with alternatives.
{a b c}	A mandatory part of a command with alternatives.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
italic	<ul style="list-style-type: none">• A cross-reference within this guide or to another guide.• Emphasis.
...	An ellipsis indicates that the previous item can be repeated an arbitrary number of times.
	Identifies instructions specific to the IAR Embedded Workbench® IDE interface.
	Identifies instructions specific to the command line interface.
	Identifies helpful tips and programming hints.
	Identifies warnings.

Table 1: Typographic conventions used in this guide

NAMING CONVENTIONS

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

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

Table 2: Naming conventions used in this guide

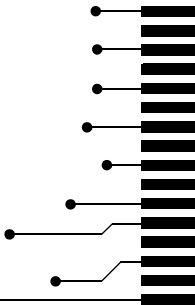
Brand name	Generic term
IAR Assembler™ for AVR	the assembler
IAR XLINK Linker™	XLINK, the linker
IAR XAR Library Builder™	the library builder
IAR XLIB Librarian™	the librarian
IAR DLIB Library™	the DLIB library
IAR CLIB Library™	the CLIB library

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

Part I. Using the compiler

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

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





Getting started

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

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

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

IAR language overview

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

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

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

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

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

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

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

Supported AVR devices

The IAR C/C++ Compiler for AVR supports all devices based on the standard Atmel® Corporation AVR microcontroller.

An up-to-date list of supported AVR devices is available on the IAR Systems web site.

Building applications—an overview

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

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

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



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

COMPILING

In the command line interface, the following line compiles the source file `myfile.c` into the object file `myfile.r90` using the default settings:

```
iccavr myfile.c
```

You must also specify some critical options, see *Basic project configuration*, page 29.

LINKING

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

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

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

```
xlink myfile.r90 myfile2.r90 -s __program_start -f lnkm128s.xcl
cl3s-ec_mul.r90 -o aout.a90 -r -FIntel-extended
```

In this example, `myfile.r90` and `myfile2.r90` are object files, `lnkm128s.xcl` is the linker configuration file, and `cl3s-ec_mul.r90` is the runtime library. The option `-s` specifies the label where the application starts. The option `-o` specifies the name of the output file, and the option `-r` is used for specifying the output format UBROF, which can be used for debugging in C-SPY®.

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

Basic project configuration

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

You need to make settings for:

- Processor configuration
- Memory model
- Size of double floating-point type
- Optimization settings

- Runtime environment.

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

PROCESSOR CONFIGURATION

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

The `--cpu` option versus the `-v` option

There are two processor options that can be used for configuring the processor support:

`--cpu=device` and `-vn`

Your application may only use one processor option at a time, and the same processor option must be used by all user and library modules to maintain consistency.

Both options set up default behavior—implicit assumptions—but note that the `--cpu` option is more precise because it contains more information about the intended target than the more generic `-v` option. The `--cpu` option knows, for example, how much flash memory is available in the given target.

The `--cpu=device` option implicitly sets up all internal compiler settings needed to generate code for the processor variant you are using. These options are implicitly controlled when you use the `--cpu` option: `--eecr_address`, `--eeprom_size`, `--enhanced_core`, `--spmcr_address`, `-v` and `--64k_flash`.

Because these options are automatically set when you use the `--cpu` option, you cannot set them explicitly. For information about implicit assumptions when using the `-v` option, see *Summary of processor configuration for -v*, page 31. For more information about the generated code, see *-v*, page 226.



Use the `--cpu` or `-v` option to specify the AVR device; see the chapter *Compiler options* for syntax information.



See the *IAR Embedded Workbench® IDE Project Management and Building Guide* for information about setting project options in the IDE.

Summary of processor configuration for -v

This table summarizes the memory characteristics for each -v option:

Generic processor option	Available memory models	Function memory attribute	Max addressable data	Max module and/or program size
-v0 (default)	Tiny	__nearfunc	≤ 256 bytes	≤ 8 Kbytes
-v1	Tiny, Small	__nearfunc	≤ 64 Kbytes	≤ 8 Kbytes
-v2	Tiny	__nearfunc	≤ 256 bytes	≤ 128 Kbytes
-v3	Tiny, Small	__nearfunc	≤ 64 Kbytes	≤ 128 Kbytes
-v4	Small, Large	__nearfunc	≤ 16 Mbytes	≤ 128 Kbytes
-v5	Tiny, Small	__farfunc*	≤ 64 Kbytes	≤ 8 Mbytes
-v6	Small, Large	__farfunc*	≤ 16 Mbytes	≤ 8 Mbytes

Table 3: Summary of processor configuration

Note:

- *) When using the -v5 or the -v6 option, it is possible, for individual functions, to override the __farfunc attribute and instead use the __nearfunc attribute
- Pointers with function memory attributes have restrictions in implicit and explicit casts between pointers and between pointers and integer types. For details about the restrictions, see *Casting*, page 238.
- -v2: There are currently no devices that match this processor option, which has been added to support future devices
- All implicit assumptions for a given -v option are also true for corresponding --cpu options.

It is important to be aware of the fact that the -v option does not reflect the amount of used data, but the maximum amount of addressable data. This means that, for example, if you are using a microcontroller with 16 Mbytes addressable data, but you are not using more than 256 bytes or 64 Kbytes of data, you must still use either the -v4 or the -v6 option for 16 Mbytes data.

MEMORY MODEL

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

In the compiler, you can set a default memory access method by selecting a memory model. There are three memory models available—Tiny, Small, and Large. Your choice of processor option determines which memory models are available.

For more details about memory models, see *Memory models*, page 38.

SIZE OF DOUBLE FLOATING-POINT TYPE

Floating-point values are represented by 32- and 64-bit numbers in standard IEEE 754 format. If you use the compiler option `--64bit_doubles`, you can make the compiler use 64-bit doubles. The data type `float` is always represented using 32 bits.

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 induction variable elimination.

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

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

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

RUNTIME ENVIRONMENT

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

Two different sets of runtime libraries are provided:

- The IAR DLIB Library, which supports Standard C and C++. This library also supports floating-point numbers in IEEE 754 format and it can be configured to include different levels of support for locale, file descriptors, multibyte characters, etc.
- The IAR CLIB Library is a light-weight library, which is not fully compliant with Standard C. Neither does it fully support floating-point numbers in IEEE 754 format or does it support Embedded C++. (This library is used by default).

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

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

For more information about the runtime environments, see the chapters *The DLIB runtime environment* and *The CLIB runtime environment*, respectively.



Setting up for the runtime environment 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 for the DLIB library there are different configurations—Normal and Full—which include different levels of support for locale, file descriptors, multibyte characters, et cetera. See *Library configurations*, page 88, for more information.

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



Setting up for the runtime environment from the command line

On the compiler command line, specify whether you want the system header files for DLIB or CLIB by using the `--dlib` option or the `--clib` option. If you use the DLIB library, you can use the `--dlib_config` option instead if you also want to explicitly define which library configuration to be used.

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

```
dlavr-3s-ec_mul-64-f.r90
```

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

In addition to these options you might need to specify application-specific compiler and linker options, for example the include path to the application-specific header files, for example:

```
-I application\inc
```

For information about the prebuilt library object files for the IAR DLIB Library, see *Using a prebuilt library*, page 73. Make sure to use the object file that matches your other project options.

For information about the prebuilt object files for the IAR CLIB Library, see *Using a prebuilt library*, page 110. 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 75 (DLIB) and *Input and output*, page 111 (CLIB).
- The size of the stack and the heap, see *The stack*, page 61, and *The heap*, page 64, respectively.

Special support for embedded systems

This section briefly describes the extensions provided by the compiler to support specific features of the AVR 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 203 for additional information.

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

PRAGMA DIRECTIVES

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

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

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

PREDEFINED SYMBOLS

With the predefined preprocessor symbols, you can inspect your compile-time environment, for example time of compilation, processor variant, and memory model in use.

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

SPECIAL FUNCTION TYPES

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

For more information, see *Primitives for interrupts, concurrency, and OS-related programming*, page 48.

ACCESSING LOW-LEVEL FEATURES

For hardware-related parts of your application, accessing low-level features is essential. The compiler supports several ways of doing this: intrinsic functions, mixing C and assembler modules, and inline assembler. For information about the different methods, see *Mixing C and assembler*, page 117.

Data storage

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

Introduction

The AVR microcontroller is based on the Harvard architecture—thus code and data have separate memory spaces and require different access mechanisms. Code and different types of data are located in memory spaces as follows:

- The internal flash space, which is used for code, `__flash` declared objects, and initializers
- The data space, which can consist of external ROM, used for constants, and RAM areas used for the stack, for registers, and for variables
- The EEPROM space, which is used for variables.

DIFFERENT WAYS TO STORE DATA

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

- Auto variables

All variables that are local to a function, except those declared static, are stored on the stack. These variables can be used as long as the function executes. When the function returns to its caller, the memory space is no longer valid.

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

In this case, the memory is allocated once and for all. The word static in this context means that the amount of memory allocated for this kind of variables does not change while the application is running. For more information, see *Memory models*, page 38 and *Memory types*, page 39.

- Dynamically allocated data.

An application can allocate data on the *heap*, where the data remains valid until it is explicitly released back to the system by the application. This type of memory is useful when the number of objects is not known until the application executes. Note that there are potential risks connected with using dynamically allocated data in systems with a limited amount of memory, or systems that are expected to run for a long time. For more information, see *Dynamic memory on the heap*, page 45.

Memory models

Technically, the memory model specifies the default memory type attribute and the default data pointer attribute. This means that the memory model controls the following:

- The default placement of static and global variables, and constant literals
- Dynamically allocated data, for example data allocated with `malloc`, or, in C++, the operator `new`
- The default pointer type
- The placement of the runtime stack. For details about stack placement, see *CSTACK*, page 319.

For information about how to specify a memory type for individual objects, see *Using data memory attributes*, page 39.

SPECIFYING A MEMORY MODEL

Four memory models are implemented: Tiny, Small, Large, and Huge. These models are controlled by the `--memory_model` option. Each model has a default memory type and a default pointer size. The code size will also be reduced somewhat if the Tiny or Small memory model is used.

If you do not specify a memory model option, the compiler will use the Tiny memory model for all processor options, except for `-v4` and `-v6`, where the Small memory model will be used. For information about processor options, see *Summary of processor configuration for -v*, page 31.

Your project can only use one memory model at a time, and the same model must be used by all user modules and all library modules. However, you can override the default memory type for individual data objects and pointers by explicitly specifying a memory attribute, see *Using data memory attributes*, page 39.

This table summarizes the different memory models:

Memory model	Default memory attribute	Default data pointer	Max. stack range	Placement of data
Tiny	<code>__tiny</code>	<code>__tiny</code>	≤ 256 bytes	-v0, -v1, -v2, -v3, -v5
Small	<code>__near</code>	<code>__near</code>	≤ 64 Kbytes	-v1, -v3, -v4, -v5, -v6
Large	<code>__far</code>	<code>__far</code>	≤ 16 Mbytes	-v4, -v6
Huge	<code>__huge</code>	<code>__huge</code>	≤ 16 Mbytes	-v4, -v6

Table 4: Memory model characteristics



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



Use the `--memory_model` option to specify the memory model for your project; see `--memory_model, -m`, page 211.

Memory types

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

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

For example, the memory accessible using the near memory access method is called memory of near type, or simply near memory.

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

USING DATA MEMORY ATTRIBUTES

The compiler provides a set of *extended keywords*, which can be used as *data memory attributes*. These keywords let you override the default memory type for individual data

objects, which means that you can place data objects in other memory areas than the default memory. This also means that you can fine-tune the access method for each individual data object, which results in smaller code size.

This table summarizes the available memory attributes:

Memory attribute	Pointer size	Memory space	Address range	Max object size
__tiny	1 byte	Data	0x0-0xFF	128 bytes
__near	2 bytes	Data	0x0-0xFFFF	32 Kbytes
__far	3 bytes	Data	0x0-0xFFFFFFFF (16-bit pointer arithmetics)	32 Kbytes
__huge	3 bytes	Data	0x0-0xFFFFFFFF	8 Mbytes
__tinyflash	1 byte	Code	0x0-0xFF	128 bytes
__flash	2 bytes	Code	0x0-0xFFFF	32 Kbytes
__farflash	3 bytes	Code	0x0-0xFFFFFFFF (16-bit pointer arithmetics)	32 Kbytes
__hugeflash	3 bytes	Code	0x0-0xFFFFFFFF	8 Mbytes
__eeprom	1 bytes	EEPROM	0x0-0xFF	128 bytes
__eeprom	2 bytes	EEPROM	0x0-0xFFFF	32 Kbytes
__io	N/A	I/O space	0x0-0x3F	4 bytes
__io	N/A	Data	0x60-0xFF	4 bytes
__ext_io	N/A	Data	0x100-0xFFFF	4 bytes
__generic	2 bytes	Data or Code	The most significant bit (MSB) determines whether this pointer points to code space (1) or data space (0). The small generic pointer is generated for the processor options -v0 and -v1.	32 Kbytes
	3 bytes			8 Mbytes
__regvar	N/A	Data	0x4-0x0F	4 bytes

Table 5: 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 203 for additional information.

For more information about each keyword, see *Descriptions of extended keywords*, page 250.

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 245.

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

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

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

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

Type definitions

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

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

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

POINTERS AND MEMORY TYPES

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

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

```
int __far * MyPtr;
```

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

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

Whenever possible, pointers should be declared without memory attributes. For example, the functions in the standard library are all declared without explicit memory types.

Differences between pointer types

A pointer must contain information needed to specify a memory location of a certain memory type. This means that the pointer sizes are different for different memory types.

In the compiler, it is illegal to convert pointers between different types without explicit casts. For more information, see *Casting*, page 238.

For more information about pointers, see *Pointer types*, page 237.

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 eeprom memory.

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

__eeprom struct MyStruct gamma;
```

This declaration is incorrect:

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

MORE EXAMPLES

The following is a series of examples with descriptions. First, some integer variables are defined and then pointer variables are introduced. Finally, a function accepting a pointer to an integer in flash memory is declared. The function returns a pointer to an integer in

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

<code>int MyA;</code>	A variable defined in default memory.
<code>int __far MyB;</code>	A variable in far memory.
<code>__eprom int MyC;</code>	A variable in eprom memory.
<code>int * MyD;</code>	A pointer stored in default memory. The pointer points to an integer in default memory.
<code>int __far * MyE;</code>	A pointer stored in default memory. The pointer points to an integer in far memory.
<code>int __far * __eprom MyF;</code>	A pointer stored in eprom memory pointing to an integer stored in far memory.
<code>int __eprom * MyFunction(int __far *);</code>	A declaration of a function that takes a parameter which is a pointer to an integer stored in far memory. The function returns a pointer to an integer stored in eprom memory.

C++ and memory types

Instances of C++ classes are placed into a memory (just like all other objects) either implicitly, or explicitly using memory type attributes or other IAR language extensions. Non-static member variables, like structure fields, are part of the larger object and cannot be placed individually into specified memories.

In non-static member functions, the non-static member variables of a C++ object can be referenced via the `this` pointer, explicitly or implicitly. The `this` pointer is of the default data pointer type unless class memory is used, see *Using IAR attributes with Classes*, page 145.

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

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

For more information about C++ classes, see *Using IAR attributes with Classes*, page 145.

Auto variables—on the stack

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

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

THE STACK

The stack can contain:

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

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

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

Advantages

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

It is possible for a function to call itself either directly or indirectly—a *recursive function*—and each invocation can store its own data on the stack.

Potential problems

The way the stack works makes it impossible to store data that is supposed to live after the function returns. The following function demonstrates a common programming

mistake. It returns a pointer to the variable `x`, a variable that ceases to exist when the function returns.

```
int *MyFunction()
{
    int x;
    /* Do something here. */
    return &x; /* Incorrect */
}
```

Another problem is the risk of running out of stack. This will happen when one function calls another, which in turn calls a third, etc., and the sum of the stack usage of each function is larger than the size of the stack. The risk is higher if large data objects are stored on the stack, or when recursive functions are used.

Dynamic memory on the heap

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

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

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

The compiler supports heaps in both tiny, near, far, and huge memory. For more information about this, see *The heap*, page 64.

Potential problems

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

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

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

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

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

Functions

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

Function-related extensions

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

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

The compiler uses compiler options, extended keywords, pragma directives, and intrinsic functions to support this.

For more information about optimizations, see *Efficient coding for embedded applications*, page 159. For information about the available intrinsic functions for accessing hardware operations, see the chapter *Intrinsic functions*.

Function storage

USING FUNCTION MEMORY ATTRIBUTES

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

Memory attribute	Address range	Pointer size	Used in processor option
<code>__nearfunc</code>	0–0x1FFFE (128 Kbytes)	16 bits	–v0, –v1, –v2, –v3, –v4
<code>__farfunc</code>	0–0x7FFFFFFE (8 Mbytes)	24 bits	–v5, –v6

Table 6: Function memory attributes

When using the –v5 or the –v6 option, it is possible, for individual functions, to override the `__farfunc` attribute and instead use the `__nearfunc` attribute. The default

memory can be overridden by explicitly specifying a memory attribute in the function declaration or by using the `#pragma type_attribute` directive:

```
#pragma type_attribute=__nearfunc
void MyFunc(int i)
{
    ...
}
```

It is possible to call a `__nearfunc` function from a `__farfunc` function and vice versa. Only the size of the function pointer is affected.

It is possible to place functions into named segments using either the `@` operator or the `#pragma location` directive. For more information, see *Controlling data and function placement in memory*, page 164.

Pointers with function memory attributes have restrictions in implicit and explicit casts between pointers and between pointers and integer types. For information about the restrictions, see *Casting*, page 238.

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

Primitives for interrupts, concurrency, and OS-related programming

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

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

INTERRUPT FUNCTIONS

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

Interrupt service routines

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

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

Interrupt vectors and the interrupt vector table

For the AVR microcontroller, the interrupt vector table always starts at the address `0x0` and is placed in the `INTVEC` segment. The interrupt vector is the offset into the interrupt vector table. The interrupt vector table contains pointers to interrupt routines, including the reset routine. The AT90S80515 device has 13 interrupt vectors and one reset vector. For this reason, you should specify 14 interrupt vectors, each of two bytes.

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

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

Defining an interrupt function—an example

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

```
#pragma vector = 0x14
__interrupt void MyInterruptRoutine(void)
{
    /* Do something */
}
```

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

MONITOR FUNCTIONS

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

To define a monitor function, you can use the `__monitor` keyword. For more information, see `__monitor`, page 257.



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

Example of implementing a semaphore in C

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

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

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

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

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

        return 0;
    }
}

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

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

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

```

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

/* An example of using the semaphore. */

void MyProgram(void)
{
    GetLock();

    /* Do something here. */

    ReleaseLock();
}

```

C++ AND SPECIAL FUNCTION TYPES

C++ member functions can be declared using special function types. However, this restriction applies: Interrupt member functions must be static. When a non-static member function is called, it must be applied to an object. When an interrupt occurs and the interrupt function is called, there is no object available to apply the member function to.

Placing code and data

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

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

Segments and memory

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

WHAT IS A SEGMENT?

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

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

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

Segment memory type

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

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

Segment memory type	Description
CODE	For executable code
DATA	For data placed in RAM
XDATA	For data placed in EEPROM

Table 7: XLINK segment memory types

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

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

Placing segments in memory

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

In particular, the linker configuration file specifies:

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

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

CUSTOMIZING THE LINKER CONFIGURATION FILE

The `config` directory contains ready-made linker configuration files (filename extension `.xcl`). The files contain the information required by the linker, and are ready to be used. The only change you will normally have to make to the supplied linker configuration file is to customize it so it fits the target system memory map. If, for

example, your application uses additional external RAM, you must add details about the external RAM memory area.

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

Range	Type
0x100–0xFFFF	RAM
0x0–0x1FFFF	Flash
0x0–0xFFF	EEPROM
0x1000–0xFFFF	External RAM

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

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

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 configuration file

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

- The CPU used:
`-ca90`
 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 configuration file). Segments can be placed using the `-z` and `-P` options. The former will place the segment parts in the order they are found, while the latter will try to rearrange them to make better use of the memory. The `-P` option is useful when the memory where the segment should be placed is not continuous.

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

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

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

Using the -Z command for sequential placement

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

The following illustrates how to use the `-Z` command to place the segment `MYSEGMENTA` followed by the segment `MYSEGMENTB` in CODE memory (that is, flash memory) in the memory range `0x1000-0xCFFF`.

```
-Z (CODE) MYSEGMENTA, MYSEGMENTB=1000-CFFF
```

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

```
-Z (CODE) MYSEGMENTA=1000-CFFF
-Z (CODE) MYCODE
```

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

```
-Z (CODE) MYSMALLSEGMENT=1000-20FF
-Z (CODE) MYLARGESEGMENT=1000-CFFF
```



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

Using the -P command for packed placement

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

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

```
-P (DATA) MYDATA=0-FFF, 1000-1FFF
```

If your application has an additional RAM area in the memory range `0xF000-0xF7FF`, you can simply add that to the original definition:

```
-P (DATA) MYDATA=0-FFF, 1000-1FFF, F000-F7FF
```

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

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

Data segments

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

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

STATIC MEMORY SEGMENTS

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

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

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

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

Segment naming

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

Memory type	Segment group	Memory range
tiny	TINY	0x0-0xFF
near	NEAR	0x0-0xFFFF
far	FAR	0x0-0xFFFFFFFF
huge	HUGE	0x0-0xFFFFFFFF
EEPROM	EEPROM	0x0-0xFF or 0x0-0xFFFF, depending on your device

Table 9: Memory types with corresponding segment groups

Some of the declared data is placed in non-volatile memory, for example flash memory, and some of the data is placed in RAM. For this reason, it is also important to know the XLINK segment memory type of each segment. For more information about segment memory types, see *Segment memory type*, page 54.

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

Categories of declared data	Suffix	Segment memory type
Non-initialized data	N	DATA
Zero-initialized data	Z	DATA
Non-zero initialized data	I	DATA
Initializers for the above	ID	CODE
Constants	C	DATA
Persistent data	P	DATA
Data placed in flash memory	F	CODE

Table 10: Segment name suffixes

For information about all supported segments, see *Summary of segments*, page 317.

Examples

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

```
__near int j;  
__near int i = 0;
```

The near variables that are to be initialized to zero when the system starts are placed in the segment `NEAR_Z`.

`__no_init __near int j;` The near non-initialized variables are placed in the segment `NEAR_N`.

`__near int j = 4;` The near non-zero initialized variables are placed in the segment `NEAR_I` in RAM, and the corresponding initializer data in the segment `NEAR_ID` in CODE.

Initialized data

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

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

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

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

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

<code>NEAR_I</code>	<code>0x1000-0x10FF</code> and <code>0x1200-0x12FF</code>
<code>NEAR_ID</code>	<code>0x4000-0x41FF</code>

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:

<code>NEAR_I</code>	<code>0x1000-0x10FF</code> and <code>0x1200-0x12FF</code>
<code>NEAR_ID</code>	<code>0x4000-0x40FF</code> and <code>0x4200-0x42FF</code>

Note that the gap between the ranges will also be copied. Note also that the `NEAR_ID` segment holding the initializers is always located in flash memory and that these initializers are only used once, that is before reaching the main function.

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

Initialization of local aggregates at function invocation

Initialized aggregate auto variables—struct, union, and array variables local to a function—have the initial values in blocks of memory. As an auto variable is allocated either in registers or on the stack, the initialization has to take place every time the function is called. Assume the following example:

```
void f()
{
    struct block b = { 3, 4, 2, 3, 6634, 234 };
    ...
}
```

The initializers are copied to the `b` variable allocated on the stack each time the function is entered.

The initializers can either come from the code memory space (flash) or from the data memory space (optional external ROM). By default, the initializers are located in segments with the suffix `_C` and these segments are copied from external ROM to the stack.

If you use either the `-y` option or the `--initializers_in_flash` option, the aggregate initializers are located in segments with the suffix `_F`, which are copied from flash memory to the stack. The advantage of storing these initializers in flash is that valuable data space is not wasted. The disadvantage is that copying from flash is slower.

Initialization of constant objects

There are different ways of initializing constant objects.

By default, constant objects are placed in segments with the suffix `_C`, which are located in the optional external ROM that resides in the data memory space. The reason for this is that it must be possible for a default pointer—a pointer without explicit memory attributes—to point to the object, and a default pointer can only point to the data memory space.

However, if you do not have any external ROM in the data memory space, and for single chip applications you most likely do not have it, the constant objects have to be placed in RAM and initialized as any other non-constant variables. To achieve this, use the `-y` option, which means the objects are placed in segments with the suffix `_ID`.

If you want to place an object in flash memory, you can use any of the memory attributes `__tinyflash`, `__flash`, `__farflash`, or `__hugeflash`. The object becomes a flash object, which means you cannot take the address of it and store it in a default pointer. However, it is possible to store the address in either a `__flash` pointer or a `__generic` pointer, though neither of these are default pointers. Note that if you attempt to take the address of a constant `__flash` object and use it as a default pointer object, the compiler will issue an error. If you make an explicit cast of the object to a default pointer object,

the error message disappears, instead there will be problems at runtime as the cast cannot copy the object from the flash memory to the data memory.

To access strings located in flash memory, you must use alternative library routines that expect flash strings. A few such alternative functions are provided—they are declared in the `pgmspace.h` header file. They are flash alternatives for some common C library functions with an extension `_P`. For your own code, you can always use the `__flash` keyword when passing the strings between functions. For reference information about the alternative functions, see *AVR-specific library functions*, page 310.

Data segments for static memory in the default linker configuration file

As described in the section , static data can be placed in many different segments depending on the application requirements and your target system. In the linker configuration file the segment definitions can look like this:

```
/* First, the segments to be placed in ROM are defined. */
-Z (CODE) TINY_F=0-FF
-Z (CODE) NEAR_F=0-1FF
-Z (CODE) TINY_ID, NEAR_ID=0-1FFF

/* Then, the RAM data segments are defined. */
-Z (DATA) TINY_I, TINY_Z, TINY_N=60-FF
-Z (DATA) NEAR_I, NEAR_Z=60-25F

/* Then, the segments to be placed in external EPROM are defined.
*/
-Z (DATA) NEAR_C=EXT_EPROM_BASE:+_EXT_EPROM_SIZE

* The _EXT_EPROM_BASE and _EXT_EPROM_SIZE symbols are defined in
the linker command file template, where they have the value 0. If
you want to use those symbols, you must provide values that suit
the hardware. This method can also be used for placing other
types of objects in the external memory space. */
```

THE STACK

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

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.

If external SRAM is available it is possible to place the stack there. However, the external memory is slower than the internal stack so moving it to external memory will decrease the performance.

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



Stack size allocation in the IDE

Choose **Project>Options**. In the **General Options** category, click the **Stack/Heap** tab. Add the required stack size in the **Stack size** text box.



Stack size allocation from the command line

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

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

```
-D_CSTACK_SIZE=size
```

Note: This line can be 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=60-25F
```

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



Stack size considerations

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

THE RETURN ADDRESS STACK

The return address stack is used for storing the return address when a `CALL`, `RCALL`, `ICALL`, or `EICALL` instruction is executed. Each call will use two or three bytes of return address stack. An interrupt will also place a return address on this stack.

The data segment used for holding the return address stack is called `RSTACK`.

To determine the size of the return address stack, see *Stack size considerations*, page 62. Notice however that if the cross-call optimization has been used (`-z9` without `--no_cross_call`), the value can be off by as much as a factor of six depending on how many times the cross-call optimizer has been run (`--cross_call_passes`). Each cross-call pass adds one level of calls, for example, two cross-call passes might result in a tripled stack usage.

If external SRAM is available, it is possible to place the stack there. However, the external memory is slower than the internal memory so moving the stacks to external memory will normally decrease the system performance; see `--enable_external_bus`, page 204.

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



RSTACK size allocation in the IDE

Choose **Project>Options**. In the **General Options** category, click the **Stack/Heap** tab. Add the required stack size in the **Return address stack size** text box.



RSTACK size allocation from the command line

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

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

```
-D_RSTACK_SIZE=size
```

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



Placement of RSTACK segment

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

```
-Z (DATA) RSTACK+_RSTACK_SIZE=60-25F
```

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

THE HEAP

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

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

- The linker segment used for the heap, which differs between the DLIB and the CLIB runtime environment
- The steps involved for allocating the heap size, which differs depending on which build interface you are using
- The steps involved for placing the heap segments in memory.

Heap segments in DLIB

To access a heap in a specific memory, use the appropriate memory attribute as a prefix to the standard functions `malloc`, `free`, `calloc`, and `realloc`, for example:

```
__near_malloc
```

If you use any of the standard functions without a prefix, the function will be mapped to the default memory type `near`.

Each heap will reside in a segment with the name `_HEAP` prefixed by a memory attribute, for example `NEAR_HEAP`.

For information about available heaps, see *Heaps*, page 98.

Heap segments in the CLIB runtime environment

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



Heap size allocation in the IDE

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

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



Heap size allocation from the command line

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

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

```
-D_TINY_HEAP_SIZE=size
-D_NEAR_HEAP_SIZE=size
-D_FAR_HEAP_SIZE=size
-D_HUGE_HEAP_SIZE=size
```



```
-D_HEAP_SIZE=size /* For CLIB */
```

These lines can be prefixed with the comment character `//` because the IDE controls the heap size allocation. To make the directive take effect, remove the comment character.

Specify the appropriate size for your application.



Placement of heap segment

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

```
-Z (DATA) HEAP+_TINY_HEAP_SIZE=60-25F
```

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

Use the same method for all used heaps.



Heap size and standard I/O

If you have excluded `FILE` descriptors from the DLIB runtime environment, like in the normal configuration, there are no input and output buffers at all. Otherwise, like in the full configuration, be aware that the size of the input and output buffers is set to 512 bytes in the `stdio` library header file. If the heap is too small, I/O will not be buffered, which is considerably slower than when I/O is buffered. If you execute the application using the simulator driver of C-SPY, you are not likely to notice the speed penalty, but it is quite noticeable when the application runs on an AVR microcontroller. If you use the standard I/O library, you should set the heap size to a value which accommodates the needs of the standard I/O buffer.

LOCATED DATA

A variable that is explicitly placed at an address, for example by using the `#pragma location` directive or the `@` operator, is placed in the `SEGMENT_AN` segment. The individual segment part of the segment knows its location in the memory space, and it *should* not be specified in the linker configuration file.

USER-DEFINED SEGMENTS

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

Code segments

This section contains descriptions of the segments used for storing code, and the interrupt vector table. For information about all segments, see *Summary of segments*, page 317.

INTERRUPT AND RESET VECTORS

The interrupt vector table contains pointers to interrupt routines, including the reset routine. The table is placed in the segment `INTVEC`. The AT90S80515 device has 13 interrupt vectors and one reset vector. For this reason, you should specify 14 interrupt vectors, each of two bytes.

The linker directive would then look like this:

```
-Z(CODE)INTVEC=0-1B /* 14 interrupt vectors; 2 bytes each */
```

FUNCTIONS

Functions are placed in the `CODE` or `FARCODE` segments, depending on which `-v` processor option you are using. The `-v` option implicitly determines the default function memory attributes `__nearfunc` or `__farfunc`, which in turn determines the used segments for the functions. For information about which attribute is used by default for each `-v` option, see *Summary of processor configuration for -v*, page 31.

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

```
-Z(CODE)CODE=0-1FFF
```

Compiler-generated segments

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

- The `SWITCH` segment which contains data statements used in the switch library routines. These tables are encoded in such a way as to use as little space as possible.
- The `INITTAB` segment contains the segment initialization description blocks that are used by the `__segment_init` function which is called by `CSTARTUP`. This table consists of a number of `SegmentInitBlock_Type` objects. This type is declared in the `segment_init.h` file which is located in the `avr\src\lib` directory.

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

```
-Z(CODE)SWITCH, INITTAB=0-1FFF
```

C++ dynamic initialization

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

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

For example:

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

For additional information, see *DIFUNCT*, page 320.

Verifying the linked result of code and data placement

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

SEGMENT TOO LONG ERRORS AND RANGE ERRORS

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

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

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

LINKER MAP FILE

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

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



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

Normally, XLINK will not generate an output file if any errors, such as range errors, occur during the linking process. Use the option **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 more information about the listing options and the linker listing, see the *IAR Linker and Library Tools Reference Guide*, and the *IAR Embedded Workbench® IDE Project Management and Building Guide*.

MANAGING MULTIPLE ADDRESS SPACES

Output formats that do not support more than one memory space—like `MOTOROLA` and `INTEL-HEX`—might require up to one output file per memory space. This causes no problems if you are only producing output to one memory space (flash), but if you also are placing objects in EEPROM or an external ROM in `DATA` memory space, the output format cannot represent this, and the linker issues this error message:

```
Error[e133]: The output format Format cannot handle multiple
address spaces. Use format variants (-y -O) to specify which
address space is wanted.
```

To limit the output to flash memory, make a copy of the linker configuration file for the device and memory model you are using, and put it in your project directory. Use this copy in your project and add this line at the end of the file:

```
-y (CODE)
```

To produce output for the other memory space(s), you must generate one output file per memory space (because the output format you chose does not support more than one memory space). Use the XLINK option `-O` for this purpose.

For each additional output file, you must specify format, XLINK segment type, and file name. For example:

```
-Omotorola,(DATA)=external_rom.a90
-Omotorola,(XDATA)=eeprom.a90
```

Note: As a general rule, an output file is only necessary if you use non-volatile memory. In other words, output from the data space is only necessary if the data space contains external ROM.

The IAR Postlink utility

You can also use the IAR Postlink utility, delivered with the compiler. This application takes as input an object file (of the XLINK `simple` format) and extracts one or more of its XLINK segment types into one file (which can be in either Intel extended hex

format or Motorola S-record format). For example, it can put all code segments into one file, and all EEPROM segments into another.

See the `postlink.htm` document for more information about IAR Postlink.

The DLIB runtime environment

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

DLIB can be used with both the C and the C++ languages. CLIB, on the other hand, can only be used with the C language. For more information, see the chapter *The CLIB runtime environment*.

Introduction to the runtime environment

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

RUNTIME ENVIRONMENT FUNCTIONALITY

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

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

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

- Support for hardware features:
 - Direct access to low-level processor operations by means of *intrinsic* functions, such as functions for interrupt mask handling
 - Peripheral unit registers and interrupt definitions in include files
 - Special compiler support for accessing strings in flash memory, see *AVR-specific library functions*, page 310.
- Runtime environment support, that is, startup and exit code and low-level interface to some library functions.

- A floating-point environment (fenv) that contains floating-point arithmetics support, see *fenv.h*, page 307.
- Special compiler support, for instance functions for switch handling or integer arithmetics.

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

SETTING UP THE RUNTIME ENVIRONMENT

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

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

- Choose which library to use—the DLIB or the CLIB library
Use the compiler option `--clib` or `--dlib`, respectively. For more information about the libraries, see *Library overview*, page 301.
- Choose which runtime library object file to use
The IDE will automatically choose a runtime library based on your project settings. If you build from the command line, you must specify the object file explicitly. See *Using a prebuilt library*, page 73.
- Choose which predefined runtime library configuration to use—Normal, or Full
You can configure the level of support for certain library functionality, for example, locale, file descriptors, and multibyte characters. If you do not specify anything, a default library configuration file that matches the library object file is automatically used. To specify a library configuration explicitly, use the `--dlib_config` compiler option. See *Library configurations*, page 88.
- Optimize the size of the runtime library
You can specify the formatters used by the functions `printf`, `scanf`, and their variants, see *Choosing formatters for printf and scanf*, page 75. You can also specify the size and placement of the stacks and the heaps, see *The stack*, page 61, and *The heap*, page 64, respectively.
- Include debug support for runtime and I/O debugging
The library offers C-SPY debug support and if you want to debug your application, you must choose to use it, see *Application debug support*, page 78
- Adapt the library functionality
Some library functions must be customized to suit your target hardware, for example low-level functions for character-based I/O, environment functions, signal functions,

and time functions. This can be done without rebuilding the entire library, see *Overriding library modules*, page 81.

- Customize system initialization

It is likely that you need to customize the source code for system initialization, for example, your application might need to initialize memory-mapped special function registers, or omit the default initialization of data segments. You do this by customizing the routine `__low_level_init`, which is executed before the data segments are initialized. See *System startup and termination*, page 84 and *Customizing system initialization*, page 87.

- Configure your own library configuration files

In addition to the prebuilt library configurations, you can make your own library configuration, but that requires that you *rebuild* the library. This gives you full control of the runtime environment. See *Building and using a customized library*, page 82.

- Manage a multithreaded environment

In a multithreaded environment, you must adapt the runtime library to treat all library objects according to whether they are global or local to a thread. See *Managing a multithreaded environment*, page 98.

- Check module consistency

You can use runtime model attributes to ensure that modules are built using compatible settings, see *Checking module consistency*, page 103.

Using a prebuilt library

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

- Type of library
- Processor option (`-v`)
- Memory model option (`--memory_model`)
- AVR enhanced core option (`--enhanced_core`)
- Small flash memory option (`--64k_flash`)
- 64-bit doubles option (`--64bit_doubles`)
- Library configuration—Normal, or Full.

CHOOSING A LIBRARY



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



- If you build your application from the command line, make the following settings:
- Specify which library object file to use on the XLINK command line, for instance:
`dlavr-3s-ec_mul-64-f.r90`
 - If you do not specify a library configuration, the default will be used. However, you can specify the library configuration explicitly for the compiler:
`--dlib_config C:\...\dlavr-3s-ec_mul-64-f.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 `avr\lib\dlib`.

These are some examples of how to decode a library name:

Library	Generic processor option	Memory model	Core	Small flash	64-bit doubles	Library configuration
dlavr-3s-ec_mul-sf-n.r90	-v3	Small	X	X	--	Normal
dlavr-3s-ec_mul-64-f.r90	-v3	Small	X	--	X	Full

Table 11: Prebuilt libraries

Library filename syntax

The names of the libraries are constructed in this way:

`{library}{target}-{cpu}-{memModel}-{core}-{smallFlash}-{64BitDoubles}-{lib_config}.r90`

where

- | | |
|-------------------------|--|
| <code>{library}</code> | is dl for the IAR DLIB runtime environment |
| <code>{target}</code> | is avr |
| <code>{memModel}</code> | is either t, s, l, or h for Tiny, Small, Large, or Huge memory model, respectively |

<code>{core}</code>	is <code>ec_mul</code> or <code>ec_nomul</code> when enhanced core is used depending on whether the <code>MUL</code> instruction is available or not, or <code>xmega</code> if the device is from the <code>xmega</code> family. If neither the enhanced core nor the <code>xmega</code> core is used, this value is not specified.
<code>{smallFlash}</code>	is <code>sf</code> when the small flash memory is available. When small flash memory is not available, this value is not specified
<code>{64bitDoubles}</code>	is <code>64</code> when 64-bit doubles are used. When 32-bit doubles are used, this value is not specified.
<code>{lib_config}</code>	is one of <code>n</code> or <code>f</code> for normal and full, respectively.

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

CUSTOMIZING A PREBUILT LIBRARY WITHOUT REBUILDING

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

These items can be customized:

Items that can be customized	Described in
Formatters for <code>printf</code> and <code>scanf</code>	<i>Choosing formatters for <code>printf</code> and <code>scanf</code></i> , page 75
Startup and termination code	<i>System startup and termination</i> , page 84
Low-level input and output	<i>Standard streams for input and output</i> , page 89
File input and output	<i>File input and output</i> , page 92
Low-level environment functions	<i>Environment interaction</i> , page 95
Low-level signal functions	<i>Signal and raise</i> , page 96
Low-level time functions	<i>Time</i> , page 97
Size of heaps, stacks, and segments	<i>Placing code and data</i> , page 53

Table 12: Customizable items

For information about how to override library modules, see *Overriding library modules*, page 81.

Choosing formatters for `printf` and `scanf`

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

Note: If you rebuild the library, you can optimize these functions even further, see *Configuration symbols for printf and scanf*, page 91.

CHOOSING A PRINTF FORMATTER

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

This table summarizes the capabilities of the different formatters:

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

Table 13: Formatters for printf

For information about how to fine-tune the formatting capabilities even further, see *Configuration symbols for printf and scanf*, page 91.



Specifying the print formatter in the IDE

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



Specifying the printf formatter from the command line

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

```
-e_PrintfFull=_Printf
-e_PrintfFullNoMb=_Printf
-e_PrintfLarge=_Printf
-e_PrintfLargeNoMb=_Printf
```

```
_e_PrintfSmall=_Printf
-e_PrintfSmallNoMb=_Printf
-e_PrintfTiny=_Printf
```

CHOOSING A SCANF FORMATTER

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

This table summarizes the capabilities of the different formatters:

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

Table 14: Formatters for scanf

For information about how to fine-tune the formatting capabilities even further, see Configuration symbols for printf and scanf, page 91.



Specifying the scanf formatter in the IDE

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



Specifying the scanf formatter from the command line

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

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

```
_e_ScanfSmall=_Scanf
_e_ScanfSmallNoMb=_Scanf
```

Application debug support

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

INCLUDING C-SPY DEBUGGING SUPPORT

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

This table describes the different levels of debugging support:

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

Table 15: Levels of debugging support in runtime libraries

*** If you build your application project with this level of debug support, certain functions in the library are replaced by functions that communicate with the IAR C-SPY Debugger. For more information, see *The debug library functionality*, page 79.**



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

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

THE DEBUG LIBRARY FUNCTIONALITY

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

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

The mechanism used for implementing this feature works as follows:

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

THE C-SPY TERMINAL I/O WINDOW

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

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

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

Speeding up terminal output

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

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

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

```
-e__write_buffered=__write
```

LOW-LEVEL FUNCTIONS IN THE DEBUG LIBRARY

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

Function in DLIB low-level interface	Response by C-SPY
abort	Notifies that the application has called abort *
clock	Returns the clock on the host computer
__close	Closes the associated host file on the host computer
__exit	C-SPY notifies that the end of the application was reached *
__lseek	Seeks in the associated host file on the host computer
__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 *
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 16: Functions with special meanings when linked with debug library

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

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

THE C-SPY TERMINAL I/O WINDOW

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

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

See the *C-SPY® Debugging Guide for AVR* for more information about the Terminal I/O window.

Speeding up terminal output

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

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

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

```
-e__write_buffered=__write
```

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 system startup code. 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 `avr\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:

```
iccavr library_module.c
```

This creates a replacement object module file named *library_module.r90*.

Note: The memory 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.r90* to the XLINK command line, either directly or by using an extended linker command file, for example:

```
xlink library_module.r90 dlavr-3s-ec_mul-64-n.r90
```

Make sure that *library_module.r90* 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.r90*, 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

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

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

In those cases, you must:

- Set up a library project
- Make the required library modifications

- Build your customized library
- Finally, make sure your application project will use the customized library.

Note: To build IAR Embedded Workbench projects from the command line, use the IAR Command Line Build Utility (`iarbuild.exe`). There is also a batch file (`build_libs.bat`) provided for building the library from the command line.

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

SETTING UP A LIBRARY PROJECT

The IDE provides a library project template which can be used for customizing the runtime environment configuration. This library template uses the Full library configuration, see Table 17, *Library configurations*, page 88.



In the IDE, modify the generic options in the created library project to suit your application, see *Basic project configuration*, page 29.

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

MODIFYING THE LIBRARY FUNCTIONALITY

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

The library functionality is determined by a set of *configuration symbols*. The default values of these symbols are defined in the file `DLib_Defaults.h`. This read-only file describes the configuration possibilities. Your library also has its own library configuration file `dlavrCustom.h`, which sets up that specific library with the required library configuration. For more information, see Table 12, *Customizable items*, page 75.

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

Modifying the library configuration file

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

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

USING A CUSTOMIZED LIBRARY

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



In the IDE you must do these steps:

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

System startup and termination

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

The code for handling startup and termination is located in the source files `cstartup.s90`, `_exit.s90`, and `low_level_init.c` located in the `avr\src\lib` directory.

For information about how to customize the system startup code, see *Customizing system initialization*, page 87.

SYSTEM STARTUP

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

For the hardware initialization, it looks like this:

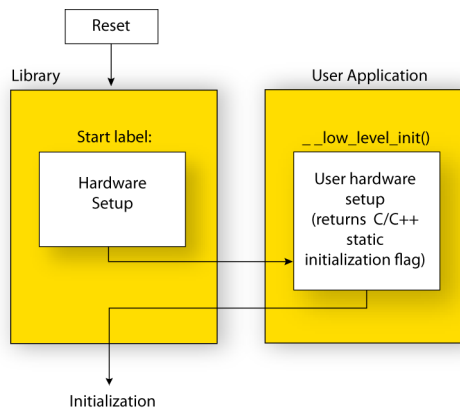


Figure 1: Target hardware initialization phase

- When the CPU is reset it will jump to the program entry label `__program_start` in the system startup code.
- The external data and address buses are enabled if needed.
- The stack pointer is initialized to the end of the `CSTACK` or `RSTACK` segment, respectively.
- The function `__low_level_init` is called if you defined it, giving the application a chance to perform early initializations.

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

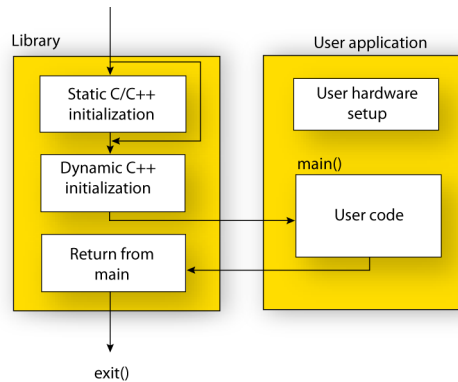


Figure 2: C/C++ initialization phase

- Static and global variables are initialized except for `__no_init`, `__tinyflash`, `__flash`, `__farflash`, `__hugeflash`, and `__eeprom` declared variables. That is, zero-initialized variables are cleared and the values of other initialized variables are copied from ROM to RAM memory. This step is skipped if `__low_level_init` returns zero. For more information, see *Initialized data*, page 59
- Static C++ objects are constructed
- The `main` function is called, which starts the application.

SYSTEM TERMINATION

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

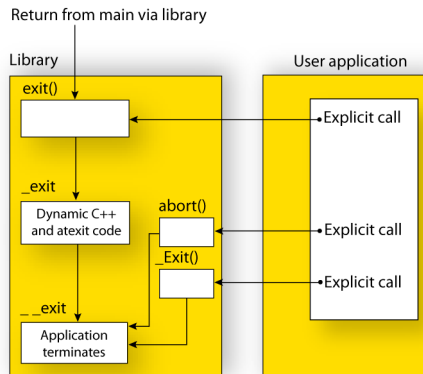


Figure 3: System termination phase

An application can terminate normally in two different ways:

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

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

The default `exit` function is written in C. It calls a small assembler function `_exit` that will perform these operations:

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

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

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

C-SPY interface to system termination

If your project is linked with the XLINK options **With runtime control modules** or **With I/O emulation modules**, the normal `__exit` and `abort` functions are replaced with special ones. C-SPY will then recognize when those functions are called and can take appropriate actions to simulate program termination. For more information, see *Application debug support*, page 78.

Customizing system initialization

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

You can do this by providing a customized version of the routine `__low_level_init`, which is called from `cstartup` before the data segments are initialized. Modifying the file `cstartup.s90` directly should be avoided.

The code for handling system startup is located in the source files `cstartup.s90` and `low_level_init.c`, located in the `avr\src\lib` directory.

Note: Normally, you do not need to customize either of the files `cstartup.s90` or `_exit.s90`.



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 82.

Note: Regardless of whether you modify the routine `__low_level_init` or the file `cstartup.s90`, you do not have to rebuild the library.

__LOW_LEVEL_INIT

You can customize the routine `__low_level_init`, which is called from the system startup code before the data segments are initialized.

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

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

MODIFYING THE FILE CSTARTUP.S90

As noted earlier, you should not modify the file `cstartup.s90` if a customized version of `__low_level_init` is enough for your needs. However, if you do need to modify the file `cstartup.s90`, 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 81.

Note that you must make sure that the linker uses the start label used in your version of `cstartup.s90`. For information about how to change the start label used by the linker, read about the `-s` option in the *IAR Linker and Library Tools Reference Guide*.

Library configurations

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

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

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

These predefined library configurations are available:

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

Table 17: Library configurations

CHOOSING A RUNTIME CONFIGURATION

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

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

- Prebuilt configuration of your choice—to specify a runtime configuration explicitly, use the `--dlib_config` compiler option. See *--dlib_config*, page 202.
- Your own configuration—you can define your own configurations, which means that you must modify the configuration file. Note that the library configuration file describes how a library was built and thus cannot be changed unless you rebuild the library. For more information, see *Building and using a customized library*, page 82.

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

Standard streams for input and output

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

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

IMPLEMENTING LOW-LEVEL CHARACTER INPUT AND OUTPUT

To implement low-level functionality of the `stdin` and `stdout` streams, you must write the functions `__read` and `__write`, respectively. You can find template source code for these functions in the `avr\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 82. Note that customizing the low-level routines for input and output does not require you to rebuild the library.

Note: If you write your own variants of `__read` or `__write`, special considerations for the C-SPY runtime interface are needed, see *Application debug support*, page 78.

Example of using `__write`

The code in this example uses memory-mapped I/O to write to an LCD display, whose port is assumed to be located at address 0x8:

```
#include <stdint.h>

__no_init volatile unsigned char lcdIO @ 8;

size_t __write(int handle,
               const unsigned char *buf,
```

```

                                size_t bufSize)
{
    size_t nChars = 0;

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

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

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

    return nChars;
}

```

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

Example of using `__read`

The code in this example uses memory-mapped I/O to read from a keyboard, whose port is assumed to be located at `0x8`:

```

#include <stddef.h>

__no_init volatile unsigned char kbIO @ 8;

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

    /* Check for stdin
       (only necessary if FILE descriptors are enabled) */

```

```
if (handle != 0)
{
    return -1;
}

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

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

return nChars;
}
```

For information about the @ operator, see *Controlling data and function placement in memory*, page 164.

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 75.

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

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

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

Printf configuration symbols	Includes support for
<code>_DLIB_PRINTF_MULTIBYTE</code>	Multibyte characters
<code>_DLIB_PRINTF_LONG_LONG</code>	Long long (ll qualifier)
<code>_DLIB_PRINTF_SPECIFIER_FLOAT</code>	Floating-point numbers
<code>_DLIB_PRINTF_SPECIFIER_A</code>	Hexadecimal floating-point numbers
<code>_DLIB_PRINTF_SPECIFIER_N</code>	Output count (%n)
<code>_DLIB_PRINTF_QUALIFIERS</code>	Qualifiers h, l, L, v, t, and z

Table 18: Descriptions of printf configuration symbols

Printf configuration symbols	Includes support for
<code>_DLIB_PRINTF_FLAGS</code>	Flags <code>-</code> , <code>+</code> , <code>#</code> , and <code>0</code>
<code>_DLIB_PRINTF_WIDTH_AND_PRECISION</code>	Width and precision
<code>_DLIB_PRINTF_CHAR_BY_CHAR</code>	Output char by char or buffered

Table 18: Descriptions of printf configuration symbols (Continued)

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

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

Table 19: Descriptions of scanf configuration symbols

CUSTOMIZING FORMATTING CAPABILITIES

To customize the formatting capabilities, you must;

- 1 Set up a library project, see *Building and using a customized library*, page 82.
- 2 Define the configuration symbols according to your application requirements.

File input and output

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

Note that file I/O capability in the library is only supported by libraries with full library configuration, see *Library configurations*, page 88. In other words, file I/O is supported when the configuration symbol `__DLIB_FILE_DESCRIPTOR` is enabled. If not enabled, functions taking a `FILE *` argument cannot be used.

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

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

Table 20: Low-level I/O files

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

Note: If you link your library with I/O debugging support, C-SPY variants of the low-level I/O functions are linked for interaction with C-SPY. For more information, see *Application debug support*, page 78.

Locale

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

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

The DLIB library can be used in two main modes:

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

LOCALE SUPPORT IN PREBUILT LIBRARIES

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

- All prebuilt libraries support the C locale only

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

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

CUSTOMIZING THE LOCALE SUPPORT

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

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

Locale configuration symbols

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

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

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

If you want to customize the locale support, you simply define the locale configuration symbols required by your application. For more information, see *Building and using a customized library*, page 82.

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

Building a library without support for locale interface

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

Building a library with support for locale interface

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

CHANGING LOCALES AT RUNTIME

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

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

lang_REGION

or

lang_REGION.encoding

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

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

Example

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

```
setlocale (LC_ALL, "sv_FI.Utf8");
```

Environment interaction

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

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

THE GETENV FUNCTION

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

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

```
key=value\0
```

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

For example:

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

If you need a more sophisticated environment variable handling, you should implement your own `getenv`, and possibly `putenv` function. This does not require that you rebuild the library. You can find source templates in the files `getenv.c` and `environ.c` in the `avr\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 81.

THE SYSTEM FUNCTION

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

If you decide to rebuild the library, you can find source templates in the library project template. For more information, see *Building and using a customized library*, page 82.

Note: If you link your application with support for I/O debugging, the functions `getenv` and `system` are replaced by C-SPY variants. For more information, see *Application debug support*, page 78.

Signal and raise

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

This does not require that you rebuild the library. You can find source templates in the files `signal.c` and `raise.c` in the `avr\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 81.

If you decide to rebuild the library, you can find source templates in the library project template. For more information, see *Building and using a customized library*, page 82.

Time

To make the `__time32`, `__time64`, and `date` functions work, you must implement the functions `clock`, `__time32`, `__time64`, and `__getzone`. Whether you use `__time32` or `__time64` depends on which interface you use for `time_t`, see *time.h*, page 418.

To implement these functions does not require that you rebuild the library. You can find source templates in the files `clock.c`, `time.c`, `time64.c`, and `getzone.c` in the `avr\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 81.

If you decide to rebuild the library, you can find source templates in the library project template. For more information, see *Building and using a customized library*, page 82.

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

Note: If you link your application with support for I/O debugging, the functions `clock` and `time` are replaced by C-SPY variants that return the host clock and time respectively. For more information, see *Application debug support*, page 78.

Strtod

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

- 1 Enable the configuration symbol `_DLIB_STRTOD_HEX_FLOAT` in the library configuration file.
- 2 Rebuild the library, see *Building and using a customized library*, page 82.

Pow

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

Assert

If you linked your application with support for runtime debugging, C-SPY will be notified about failed asserts. If this is not the behavior you require, you must add the source file `xreportassert.c` to your application project. Alternatively, you can rebuild the library. The `__ReportAssert` function generates the assert notification. You can find template code in the `avr\src\lib` directory. For more information, see *Building and using a customized library*, page 82. To turn off assertions, you must define the symbol `NDEBUG`.



In the IDE, this symbol `NDEBUG` is by default defined in a Release project and *not* defined in a Debug project. If you build from the command line, you must explicitly define the symbol according to your needs. See *NDEBUG*, page 299.

Heaps

The runtime environment supports heaps in these memory types:

Memory type	Segment name	Memory attribute	Used by default in memory model
Tiny	TINY_HEAP	__tiny	Tiny
Near	NEAR_HEAP	__near	Near
Far	FAR_HEAP	__far	Far
Huge	HUGE_HEAP	__huge	Huge

Table 21: Heaps and memory types

See *The heap*, page 64 for information about how to set the size for each heap. To use a specific heap, the prefix in the table is the memory attribute to use in front of `malloc`, `free`, `calloc`, and `realloc`, for example `__near_malloc`. The default functions will use one of the specific heap variants, depending on project settings such as memory model. For information about how to use a specific heap in C++, see *New and Delete operators*, page 149.

Managing a multithreaded environment

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

There are three possible scenarios, and you need to consider which one that applies to you:

- If you are using an RTOS that supports the multithreading provided by the DLIB library, the RTOS and the DLIB library will handle multithreading which means you do not need to adapt the DLIB library.
- If you are using an RTOS that does not support or only partly supports the multithreading provided by the DLIB library, you probably need to adapt both the RTOS and the DLIB library.
- If you are not using an RTOS, you must adapt the DLIB library to get support for multithreading.

MULTITHREAD SUPPORT IN THE DLIB LIBRARY

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

- The heap, in other words when `malloc`, `new`, `free`, `delete`, `realloc`, or `calloc` is used.
- The file system (only available in the Full library configuration), but not the file streams themselves. The file system is updated when a stream is opened or closed, in other words when `fopen`, `fclose`, `fdopen`, `fflush`, or `freopen` is used.
- The signal system, in other words when `signal` is used.
- The temporary file system, in other words when `tmpnam` is used.
- Initialization of static function objects.

These library objects use TLS:

Library objects using TLS	When these functions are used
Error functions	<code>errno</code> , <code>strerror</code>
Locale functions	<code>localeconv</code> , <code>setlocale</code>
Time functions	<code>asctime</code> , <code>localtime</code> , <code>gmtime</code> , <code>mktime</code>
Multibyte functions	<code>mbrlen</code> , <code>mbrtowc</code> , <code>mbsrtowc</code> , <code>mbtowc</code> , <code>wcrtomb</code> , <code>wcsrtomb</code> , <code>wctomb</code>
Rand functions	<code>rand</code> , <code>srand</code>
Miscellaneous functions	<code>atexit</code> , <code>strtok</code>

Table 22: Library objects using TLS

ENABLING MULTITHREAD SUPPORT

To enable multithread support in the library, you must:

- Add `#define _DLIB_THREAD_SUPPORT` in `DLib_product.h` and rebuild your library
- Implement code for the library's system locks interface
- If file streams are used, implement code for the library's file stream locks interface or redirect the interface to the system locks interface (using the linker option `-e`)
- Implement source code that handles thread creation, thread destruction, and TLS access methods for the library
- Modify the linker configuration file accordingly
- If any of the C++ variants are used, use the compiler option `--guard_calls`. Otherwise, function static variables with dynamic initializers might be initialized simultaneously by several threads.

You can find the required declaration of functions and definitions of macros in the `DLib_Threads.h` file, which is included by `yvals.h`.

System locks interface

This interface must be fully implemented for system locks to work:

```
typedef void *__iar_Rmtx; /* Lock info object */

void __iar_system_Mtxinit(__iar_Rmtx *); /* Initialize a system
                                         lock */
void __iar_system_Mtxdst(__iar_Rmtx *); /* Destroy a system lock */
void __iar_system_Mtxlock(__iar_Rmtx *); /* Lock a system lock */
void __iar_system_Mtxunlock(__iar_Rmtx *); /* Unlock a system
                                           lock */
```

The lock and unlock implementation must survive nested calls.

File streams locks interface

This interface is only needed for the Full library configuration. If file streams are used, they can either be fully implemented or they can be redirected to the system locks interface. This interface must be implemented for file streams locks to work:

```
typedef void *__iar_Rmtx; /* Lock info object */

void __iar_file_Mtxinit(__iar_Rmtx *); /* Initialize a file lock */
void __iar_file_Mtxdst(__iar_Rmtx *); /* Destroy a file lock */
void __iar_file_Mtxlock(__iar_Rmtx *); /* Lock a file lock */
void __iar_file_Mtxunlock(__iar_Rmtx *); /* Unlock a file lock */
```

The lock and unlock implementation must survive nested calls.

DLIB lock usage

The number of locks that the DLIB library assumes exist are:

- `_FOPEN_MAX`—the maximum number of file stream locks. These locks are only used in the Full library configuration, in other words only if both the macro symbols `_DLIB_FILE_DESCRIPTOR` and `_FILE_OP_LOCKS` are true.
- `_MAX_LOCK`—the maximum number of system locks.

Note that even if the application uses fewer locks, the DLIB library will initialize and destroy all of the locks above.

For information about the initialization and destruction code, see `xsyslock.c`.

TLS handling

The DLIB library supports TLS memory areas for two types of threads: the *main thread* (the `main` function including the system startup and exit code) and *secondary threads*.

The main thread's TLS memory area:

- Is automatically created and initialized by your application's startup sequence
- Is automatically destructed by the application's destruct sequence
- Is located in the segment `__DLIB_PERTHREAD`
- Exists also for non-threaded applications.

Each secondary thread's TLS memory area:

- Must be manually created and initialized
- Must be manually destructed
- Is located in a manually allocated memory area.

If you need the runtime library to support secondary threads, you must override the function:

```
void *__iar_dlib_perthread_access(void *sympb);
```

The parameter is the address to the TLS variable to be accessed—in the main thread's TLS area—and it should return the address to the symbol in the current TLS area.

Two interfaces can be used for creating and destroying secondary threads. You can use the following interface that allocates a memory area on the heap and initializes it. At deallocation, it destroys the objects in the area and then frees the memory.

```
void *__iar_dlib_perthread_allocate(void);
void __iar_dlib_perthread_deallocate(void *);
```

Alternatively, if the application handles the TLS allocation, you can use this interface for initializing and destroying the objects in the memory area:

```
void __iar_dlib_perthread_initialize(void *);
void __iar_dlib_perthread_destroy(void *);
```

These macros can be helpful when you implement an interface for creating and destroying secondary threads:

Macro	Description
__IAR_DLIB_PERTHREAD_SIZE	Returns the size needed for the TLS memory area.
__IAR_DLIB_PERTHREAD_INIT_SIZE	Returns the initializer size for the TLS memory area. You should initialize the rest of the TLS memory area, up to __IAR_DLIB_PERTHREAD_SIZE to zero.
__IAR_DLIB_PERTHREAD_SYMBOL_OFFSET(<i>symbolptr</i>)	Returns the offset to the symbol in the TLS memory area.

Table 23: Macros for implementing TLS allocation

Note that the size needed for TLS variables depends on which DLIB resources your application uses.

This is an example of how you can handle threads:

```
#include <yvals.h>

/* A thread's TLS pointer */
void _DLIB_TLS_MEMORY *TLSp;

/* Are we in a secondary thread? */
int InSecondaryThread = 0;

/* Allocate a thread-local TLS memory
   area and store a pointer to it in TLSp. */
void AllocateTLS()
{
    TLSp = __iar_dlib_perthread_allocate();
}

/* Deallocate the thread-local TLS memory area. */
void DeallocateTLS()
{
    __iar_dlib_perthread_deallocate(TLSp);
}
```

```

/* Access an object in the
   thread-local TLS memory area. */
void _DLIB_TLS_MEMORY *__iar_dlib_perthread_access(
    void _DLIB_TLS_MEMORY *sympb)
{
    char _DLIB_TLS_MEMORY *p = 0;
    if (InSecondaryThread)
        p = (char _DLIB_TLS_MEMORY *) TLSp;
    else
        p = (char _DLIB_TLS_MEMORY *)
            __segment_begin("__DLIB_PERTHREAD");

    p += __IAR_DLIB_PERTHREAD_SYMBOL_OFFSET(sympb);
    return (void _DLIB_TLS_MEMORY *) p;
}

```

The `TLSp` variable is unique for each thread, and must be exchanged by the RTOS or manually whenever a thread switch occurs.

CHANGES IN THE LINKER CONFIGURATION FILE

If threads are used, the main thread's TLS memory area must be initialized by naive copying because the initializers are used for each secondary thread's TLS memory area as well. Insert the following statement in your linker configuration file:

```
-Q__DLIB_PERTHREAD=__DLIB_PERTHREAD_init
```

Both the `__DLIB_PERTHREAD` segment and the `__DLIB_PERTHREAD_init` segment must be placed in default memory for RAM and ROM, respectively.

Finally, the startup code must copy `__DLIB_PERTHREAD_init` to `__DLIB_PERTHREAD`.

Checking module consistency

This section introduces the concept of runtime model attributes, a mechanism used by the IAR compiler, assembler, and linker to ensure that modules are built using compatible settings.

When developing an application, it is important to ensure that incompatible modules are not used together. For example, in the compiler, it is possible to specify the size of the `double` floating-point type. If you write a routine that only works for 64-bit doubles, it is possible to check that the routine is not used in an application built using 32-bit doubles.

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

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

Table 24: Example of runtime model attributes

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

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

USING RUNTIME MODEL ATTRIBUTES

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

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

For detailed syntax information, see *rtmodel*, page 278.

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

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

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

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

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

PREDEFINED RUNTIME ATTRIBUTES

The table below shows the predefined runtime model attributes that are available for the compiler. These can be included in assembler code or in mixed C/C++ and assembler code.

Runtime model attribute	Value	Description
<code>__rt_version</code>	<i>n</i>	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>__cpu</code>	0–6	Corresponds to the <code>-v</code> option used. For more information, see <code>-v</code> , page 226.
<code>__cpu_name</code>	<i>device</i>	Corresponds to the processor device name, for example AT90S2343 or ATmega8515. Note that for the FpSLic device, the value is AT94Kxx.
<code>__double_size</code>	32 or 64	The size, in bits, of the double floating-point type. The default size is 32. Use the compiler option <code>--64bit_doubles</code> to override the default.
<code>__64bit_doubles</code>	Enabled or disabled	Corresponds to the compiler option <code>--64bit_doubles</code> .

Table 25: Predefined runtime model attributes

Runtime model attribute	Value	Description
__enhanced_core	Enabled	Available if the compiler option <code>--enhanced_core</code> or <code>--cpu</code> for a target processor with enhanced core is used.
__memory_model	1-4	Corresponds to the used memory model, where the value can be 1, 2, 3, or 4 for Tiny, Small, Large, or Huge, respectively.
__no_rampd	Enabled or disabled	Defined for targets with >64 Kbytes of data memory. Disabled if the target processor has a RAMPD register, otherwise enabled.

Table 25: Predefined runtime model attributes (Continued)

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

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

Examples

For an example of using the runtime model attribute `__rt_version` for checking the module consistency as regards the used calling convention, see *Hints for using the default calling convention*, page 124.

The following assembler source code provides a function, `myCounter`, that counts the number of times it has been called by increasing the register `R16`. The routine assumes that the application does not use `R16` for anything else, that is, the register is locked for usage. To ensure this, a runtime module attribute, `__reg_r16`, is defined with a value `counter`. This definition will ensure that this specific module can only be linked with either other modules containing the same definition, or with modules that do not set this attribute. Note that the compiler sets this attribute to `free`, unless the register is locked.

```
rtmodel      "__reg_r16", "counter"
module       myCounter
public      myCounter
rseg        CODE:CODE:NOROOT(1)
myCounter:  subi      r16, -1
            ret
            end
```

If this module is used in an application that contains modules where the register R16 is not locked, the linker issues an error:

```
Error[e117]: Incompatible runtime models. Module myCounter
specifies that '__reg_r16' must be 'counter', but module part1
has the value 'free'
```

USER-DEFINED RUNTIME MODEL ATTRIBUTES

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

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

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


The CLIB runtime environment

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

The standard library uses a small set of low-level input and output routines for character-based I/O. This chapter describes how the low-level routines can be replaced by your own version. The chapter also describes how you can choose printf and scanf formatters.

The chapter then describes system initialization and termination. It presents how an application can control what happens before the start function main is called, and the method for how you can customize the initialization. Finally, the C-SPY® runtime interface is covered.

Note that the legacy CLIB runtime environment is provided for backward compatibility and should not be used for new application projects.

For information about migrating from CLIB to DLIB, see the *IAR Embedded Workbench® Migration Guide for AVR®*.

Using a prebuilt library

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

- Type of library
- Processor option (-v)
- Memory model option (--memory_model)
- AVR enhanced core option (--enhanced_core)
- Small flash memory option (--64k_flash)
- 64-bit doubles option (--64bit_doubles).

CHOOSING A LIBRARY



The IDE includes the correct runtime library based on the options you select. See the *IAR Embedded Workbench® IDE Project Management and Building Guide* for more information.



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

```
c10t.r90
```

The CLIB runtime environment includes the C standard library. The linker will include only those routines that are required—directly or indirectly—by your application. For more information about the runtime libraries, see the chapter *Library functions*.

These are some examples of how to decode a library name:

Library file	Generic processor option	Memory model	Core	Small flash	64-bit doubles
c10t.r90	-v0	Tiny	--	--	--
c11s-64.r90	-v1	Small	--	--	X
c161-ec_mul-64.r90	-v6	Large	X	--	X

Table 26: Runtime libraries

Library filename syntax

The runtime library names are constructed in this way:

```
{type}{cpu}{memModel}-{core}-{smallFlash}-{64BitDoubles}.r90
```

where

- {type} is c1 for the IAR CLIB Library

- `{cpu}` is a value from 0 to 6, matching the `-v` option
- `{memModel}` is one of `t`, `s`, `l`, or `h` for the Tiny, Small, Large, or Huge memory model, respectively
- `{smallFlash}` is `sf` when the small flash memory is available. When small flash memory is not available, this value is not specified
- `{64BitDoubles}` is 64 when 64-bit doubles are used. When 32-bit doubles are used, this value is not specified.

Input and output

You can customize:

- The functions related to character-based I/O
- The formatters used by `printf/sprintf` and `scanf/sscanf`.

CHARACTER-BASED I/O

The functions `putchar` and `getchar` are the fundamental C functions for character-based I/O. For any character-based I/O to be available, you must provide definitions for these two functions, using whatever facilities the hardware environment provides.

The creation of new I/O routines is based on these files:

- `putchar.c`, which serves as the low-level part of functions such as `printf`
- `getchar.c`, which serves as the low-level part of functions such as `scanf`.

The code example below shows how memory-mapped I/O could be used to write to a memory-mapped I/O device:

```
__no_init volatile unsigned char devIO @ 8;

int putchar(int outChar)
{
    devIO = outChar;
    return outChar;
}
```

The exact address is a design decision. For example, it can depend on the selected processor variant.

For information about how to include your own modified version of `putchar` and `getchar` in your project build process, see *Overriding library modules*, page 81.

FORMATTERS USED BY PRINTF AND SPRINTF

The `printf` and `sprintf` functions use a common formatter, called `_formatted_write`. The full version of `_formatted_write` is very large, and provides facilities not required in many embedded applications. To reduce the memory consumption, two smaller, alternative versions are also provided in the Standard C library.

`_medium_write`

The `_medium_write` formatter has the same functions as `_formatted_write`, except that floating-point numbers are not supported. Any attempt to use a `%f`, `%g`, `%G`, `%e`, or `%E` specifier will produce a runtime error:

```
FLOATS? wrong formatter installed!
```

`_medium_write` is considerably smaller than `_formatted_write`.

`_small_write`

The `_small_write` formatter works in the same way as `_medium_write`, except that it supports only the `%%`, `%d`, `%o`, `%c`, `%s`, and `%x` specifiers for integer objects, and does not support field width or precision arguments. The size of `_small_write` is 10–15% that of `_formatted_write`.



Specifying the printf formatter in the IDE

- 1 Choose **Project>Options** and select the **General Options** category. Click the **Library options** tab.
- 2 Select the appropriate **Printf formatter** option, which can be either **Small**, **Medium**, or **Large**.



Specifying the printf formatter from the command line

To use the `_small_write` or `_medium_write` formatter, add the corresponding line in the linker configuration file:

```
-e_small_write=_formatted_write
```

or

```
-e_medium_write=_formatted_write
```

To use the full version, remove the line.

Customizing printf

For many embedded applications, `sprintf` is not required, and even `printf` with `_small_write` provides more facilities than are justified, considering the amount of memory it consumes. Alternatively, a custom output routine might be required to support particular formatting needs or non-standard output devices.

For such applications, a much reduced version of the `printf` function (without `sprintf`) is supplied in source form in the file `intwri.c`. This file can be modified to meet your requirements, and the compiled module inserted into the library in place of the original file; see *Overriding library modules*, page 81.

FORMATTERS USED BY SCANF AND SSCANF

Similar to the `printf` and `sprintf` functions, `scanf` and `sscanf` use a common formatter, called `_formatted_read`. The full version of `_formatted_read` is very large, and provides facilities that are not required in many embedded applications. To reduce the memory consumption, an alternative smaller version is also provided.

`_medium_read`

The `_medium_read` formatter has the same functions as the full version, except that floating-point numbers are not supported. `_medium_read` is considerably smaller than the full version.



Specifying the scanf formatter in the IDE

- 1 Choose **Project>Options** and select the **General Options** category. Click the **Library options** tab.
- 2 Select the appropriate **Scanf formatter** option, which can be either **Medium** or **Large**.



Specifying the read formatter from the command line

To use the `_medium_read` formatter, add this line in the linker configuration file:

```
-e _medium_read=_formatted_read
```

To use the full version, remove the line.

System startup and termination

This section describes the actions the runtime environment performs during startup and termination of applications.

The code for handling startup and termination is located in the source files `cstartup`, `_exit.s90`, and `low_level_init.c` located in the `avr\src\lib` directory.

SYSTEM STARTUP

When an application is initialized, a number of steps are performed:

- The custom function `__low_level_init` is called, giving the application a chance to perform early initializations
- The external data and address buses are enabled if needed
- The stack pointers are initialized to the end of `CSTACK` and `RSTACK`, respectively
- Static variables are initialized except for `__no_init` and `__eeprom` declared variables; this includes clearing zero-initialized memory and copying the ROM image of the RAM memory of the remaining initialized variables
- The `main` function is called, which starts the application.

Note that the system startup code contains code for more steps than described here. The other steps are applicable to the DLIB runtime environment.

SYSTEM TERMINATION

An application can terminate normally in two different ways:

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

Because the C standard states that the two methods should be equivalent, the `cstartup` 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 assembler.

When the application is built in debug mode, C-SPY stops when it reaches the special code label `?C_EXIT`.

An application can also exit by calling the `abort` function. The default function just calls `__exit` to halt the system, without performing any type of cleanup.

Overriding default library modules

The IAR CLIB Library contains modules which you probably need to override with your own customized modules, for example for character-based I/O, without rebuilding the entire library. For information about how to override default library modules, see *Overriding library modules*, page 81, in the chapter *The DLIB runtime environment*.

Customizing system initialization

For information about how to customize system initialization, see *Customizing system initialization*, page 87.

C-SPY runtime interface

The low-level debugger interface is used for communication between the application being debugged and the debugger itself. The interface is simple: C-SPY will place breakpoints on certain assembler labels in the application. When code located at the special labels is about to be executed, C-SPY will be notified and can perform an action.

THE DEBUGGER TERMINAL I/O WINDOW

When code at the labels `?C_PUTCHAR` and `?C_GETCHAR` is executed, data will be sent to or read from the debugger window.

For the `?C_PUTCHAR` routine, one character is taken from the output stream and written. If everything goes well, the character itself is returned, otherwise `-1` is returned.

When the label `?C_GETCHAR` is reached, C-SPY returns the next character in the input field. If no input is given, C-SPY waits until the user types some input and presses the Return key.

To make the Terminal I/O window available, the application must be linked with the XLINK option **With I/O emulation modules** selected. See the *IAR Embedded Workbench® IDE Project Management and Building Guide*.

TERMINATION

The debugger stops executing when it reaches the special label `?C_EXIT`.

Checking module consistency

For information about how to check module consistency, see *Checking module consistency*, page 103.

Assembler language interface

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

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

Finally, the chapter covers how you can implement support for call frame information in your assembler routines for use in the C-SPY® Call Stack window.

Mixing C and assembler

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

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

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

INTRINSIC FUNCTIONS

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

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

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

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

MIXING C AND ASSEMBLER MODULES

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

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

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

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

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

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

The first issue is discussed in the section *Calling assembler routines from C*, page 120. The following two are covered in the section *Calling convention*, page 123.

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 133.

The recommended method for mixing C or C++ and assembler modules is described in *Calling assembler routines from C*, page 120, and *Calling assembler routines from C++*, page 122, respectively.

INLINE ASSEMBLER

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

```
bool flag;

void Func()
{
    while (!flag)
    {
        asm("IN R0, PIND \n"
            "STS flag, R0");
    }
}
```

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 might have on the surrounding code are not taken into consideration. If, for example, registers or memory locations are altered, they might have to be restored within the sequence of inline assembler instructions for the rest of the code to work properly.

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

- The compiler's various optimizations will disregard any effects of the inline sequences, which will not be optimized at all
- In general, assembler directives will cause errors or have no meaning. Data definition directives will however work as expected
- Auto variables cannot be accessed.

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

Calling assembler routines from C

An assembler routine that will be called from C must:

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

```
extern int foo(void);
```

or

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

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

CREATING SKELETON CODE

The recommended way to create an assembler language routine with the correct interface is to start with an assembler language source file created by the C compiler. Note that you must create skeleton code for each function prototype.

The following example shows how to create skeleton code to which you can easily add the functional body of the routine. The skeleton source code only needs to declare the

variables required and perform simple accesses to them. In this example, the assembler routine takes an `int` and a `char`, and then returns an `int`:

```
extern int gInt;
extern char gChar;

int Func(int arg1, char arg2)
{
    int locInt = arg1;
    gInt = arg1;
    gChar = arg2;
    return locInt;
}

int main()
{
    int locInt = gInt;
    gInt = Func(locInt, gChar);
    return 0;
}
```

Note: In this example we use a low optimization level when compiling the code to show local and global variable access. If a higher level of optimization is used, the required references to local variables could be removed during the optimization. The actual function declaration is not changed by the optimization level.

COMPILING THE CODE



In the IDE, specify list options on file level. Select the file in the workspace window. Then choose **Project>Options**. In the **C/C++ Compiler** category, select **Override inherited settings**. On the **List** page, deselect **Output list file**, and instead select the **Output assembler file** option and its suboption **Include source**. Also, be sure to specify a low level of optimization.



Use these options to compile the skeleton code:

```
iccavr skeleton.c -lA .
```

The `-lA` option creates an assembler language output file including C or C++ source lines as assembler comments. The `.` (period) specifies that the assembler file should be named in the same way as the C or C++ module (`skeleton`), but with the filename extension `s90`. Also remember to specify the memory model you are using, a low level of optimization, and `-e` for enabling language extensions.

The result is the assembler source output file `skeleton.s90`.

Note: The `-lA` option creates a list file containing call frame information (CFI) directives, which can be useful if you intend to study these directives and how they are



used. If you only want to study the calling convention, you can exclude the `CFI` directives from the list file. In the IDE, choose **Project>Options>C/C++ Compiler>List** and deselect the suboption **Include call frame information**. On the command line, use the option `-lB` instead of `-lA`. Note that `CFI` information must be included in the source code to make the C-SPY Call Stack window work.

The output file

The output file contains the following important information:

- The calling convention
- The return values
- The global variables
- The function parameters
- How to create space on the stack (auto variables)
- Call frame information (CFI).

The `CFI` directives describe the call frame information needed by the Call Stack window in the debugger. For more information, see *Call frame information*, page 133.

Calling assembler routines from C++

The C calling convention does not apply to C++ functions. Most importantly, a function name is not sufficient to identify a C++ function. The scope and the type of the function are also required to guarantee type-safe linkage, and to resolve overloading.

Another difference is that non-static member functions get an extra, hidden argument, the `this` pointer.

However, when using C linkage, the calling convention conforms to the C calling convention. An assembler routine can therefore be called from C++ when declared in this manner:

```
extern "C"
{
    int MyRoutine(int);
}
```

In C++, data structures that only use C features are known as PODs (“plain old data structures”), they use the same memory layout as in C. However, we do not recommend that you access non-PODs from assembler routines.

The following example shows how to achieve the equivalent to a non-static member function, which means that the implicit `this` pointer must be made explicit. It is also possible to “wrap” the call to the assembler routine in a member function. Use an inline member function to remove the overhead of the extra call—this assumes that function inlining is enabled:

```
class MyClass;

extern "C"
{
    void DoIt(MyClass *ptr, int arg);
}

class MyClass
{
public:
    inline void DoIt(int arg)
    {
        ::DoIt(this, arg);
    }
};
```

Note: Support for C++ names from assembler code is extremely limited. This means that:

- Assembler list files resulting from compiling C++ files cannot, in general, be passed through the assembler.
- It is not possible to refer to or define C++ functions that do not have C linkage in assembler.

Calling convention

A calling convention is the way a function in a program calls another function. The compiler handles this automatically, but, if a function is written in assembler language, you must know where and how its parameters can be found, how to return to the program location from where it was called, and how to return the resulting value.

It is also important to know which registers an assembler-level routine must preserve. If the program preserves too many registers, the program might be ineffective. If it preserves too few registers, the result would be an incorrect program.

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

- Choosing a calling convention

- Function declarations
- C and C++ linkage
- Preserved versus scratch registers
- Function entrance
- Function exit
- Return address handling.

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

CHOOSING A CALLING CONVENTION

You can choose between two calling conventions:

- The old calling convention offers a simple assembler interface. It is compatible with the calling convention used in version 1.x of the compiler. Even though this convention is not used by default, it is recommended for use when mixing C and assembler code
- The new calling convention is default. It is more efficient than the old calling convention, but also more complex to understand and subject to change in later versions of the compiler.



To choose another calling convention than the default, use the `--version1_calls` command line option. You can also declare individual functions to use the old calling convention by using the `__version1` function attribute, for example:

```
extern
__version_1 void doit(int arg);
```

For details about the `--version1_calls` option and the `__version1` attribute, see `--version1_calls`, page 227 and `__version_1`, page 263, respectively.



In the IDE, choose **Use ICCA90 1.x calling convention** on the **Project>C/C++ Compiler>Code** page.



Hints for using the default calling convention

The default calling convention is very complex, and therefore not recommended for use when calling assembler routines. However, if you intend to use it for your assembler routines, you should create a list file and see how the compiler assigns the different parameters to the available registers. For an example, see *Creating skeleton code*, page 120.

If you intend to use the default calling convention, you should *also* specify a value to the runtime model attribute `__rt_version` using the `RTMODEL` assembler directive:

```
RTMODEL "__rt_version", "value"
```

The parameter *value* should have the same value as the one used internally by the compiler. For information about what value to use, see the generated list file. If the calling convention changes in future compiler versions, the runtime model value used internally by the compiler will also change. Using this method gives a module consistency check, because the linker produces errors for mismatches between the values.

For more information about checking module consistency, see *Checking module consistency*, page 103.

FUNCTION DECLARATIONS

In C, a function must be declared in order for the compiler to know how to call it. A declaration could look as follows:

```
int MyFunction(int first, char * second);
```

This means that the function takes two parameters: an integer and a pointer to a character. The function returns a value, an integer.

In the general case, this is the only knowledge that the compiler has about a function. Therefore, it must be able to deduce the calling convention from this information.

USING C LINKAGE IN C++ SOURCE CODE

In C++, a function can have either C or C++ linkage. To call assembler routines from C++, it is easiest if you make the C++ function have C linkage.

This is an example of a declaration of a function with C linkage:

```
extern "C"
{
    int F(int);
}
```

It is often practical to share header files between C and C++. This is an example of a declaration that declares a function with C linkage in both C and C++:

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

int F(int);
```

```
#ifdef __cplusplus
}
#endif
```

PRESERVED VERSUS SCRATCH REGISTERS

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

Scratch registers

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

For both calling conventions, these 14 registers can be used as scratch registers by a function:

R0-R3, R16-R23, and R30-R31

Preserved registers

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

For both calling conventions, these registers are preserved registers:

R4-R15 and R24-R27

Note that the registers R4-R15 can be locked from the command line and used for global register variables; see `--lock_regs`, page 210 and `__regvar`, page 260.

Special registers

For some registers, you must consider certain prerequisites:

- The stack pointer—register Y—must at all times point to the last element on the stack. In the eventuality of an interrupt, everything below the point the stack pointer points to, will be destroyed.
- If using the `-v4` or `-v6` processor option, the `RAMPY` register is part of the data stack pointer.
- If using a processor option which uses any of the registers `EIND`, `RAMPX`, or `RAMPZ`, these registers are treated as scratch registers.

FUNCTION ENTRANCE

During a function call, the calling function:

- passes the parameters, either in registers or on the stack
- pushes any other parameters on the data stack (CSTACK)

Control is then passed to the called function with the return address being automatically pushed on the return address stack (RSTACK).

The called function:

- stores any local registers required by the function on the data stack
- allocates space for its auto variables and temporary values
- proceeds to run the function itself.

Register parameters versus stack parameters

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

Hidden parameters

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

- A function returning structures or unions larger than 4 bytes gets an extra hidden parameter, which is a default pointer—depending on the used memory model—pointing to the location where the result should be stored. This pointer must be returned to the callee.
- For non-static C++ member functions, the `this` pointer is passed as the first parameter (but placed after the return structure pointer, if there is one). Note that static member functions do not have a `this` pointer.

Register parameters

For both calling conventions, the registers available for passing parameters are:

R16–R23

Parameters are allocated to registers using a first-fit algorithm, using parameter alignment requirements according to this table:

Parameters	Alignment	Passed in registers
8-bit values	1	R16, R17, R18, R19, R20, R21, R22, R23
16-bit values	2	R17:R16, R19:R18, R21:R20, R23:R22
24-bit values	4	R18:R17:R16, R22:R21:R20
32-bit values	4	R19:R18:R17:R16, R23:R22:R21:R20
64-bit values	8	R23:R22:R21:R20:R19:R18:R17:R16

Table 27: Registers used for passing parameters

Register assignment using the default calling convention

In the default calling convention, as many parameters as possible are passed in registers. The remaining parameters are passed on the stack. The compiler may change the order of the parameters in order to achieve the most efficient register usage.

The algorithm for assigning parameters to registers is quite complex in the default calling convention. For details, you should create a list file and see how the compiler assigns the different parameters to the available registers, see *Creating skeleton code*, page 120.

Below follows some examples of combinations of register assignment.

A function with the following signature (not C++):

```
void Func(char __far * a, int b, char c, int d)
```

would have a allocated to R18:R17:R16, b to R21:R20 (alignment requirement prevents R20:R19), c to R19 (first fit), and d to R23:R22 (first fit).

Another example:

```
void bar(char a, int b, long c, char d)
```

This would result in a being allocated to R16 (first fit), b to R19:R18 (alignment), c to R23:R22:R21:R20 (first fit), and d to R17 (first fit).

A third example:

```
void baz(char a, char __far * b, int c, int d)
```

This would give, a being allocated to R16, b to R22:R21:R20, c to R19:R18, and d to the stack.

Register assignment using the old calling convention

In the old calling convention, the two left-most parameters are passed in registers if they are scalar and up to 32 bits in size.

This table shows some of the possible combinations:

Parameters*	Parameter 1	Parameter 2
f (b1, b2, ...)	R16	R20
f (b1, w2, ...)	R16	R20, R21
f (w1, l1, ...)	R16, R17	R20, R21, R22, R23
f (l1, b2, ...)	R16, R17, R18, R19	R20
f (l1, l2, ...)	R16, R17, R18, R19	R20, R21, R22, R23

Table 28: Passing parameters in registers

* Where **b** denotes an 8-bit data type, **w** denotes a 16-bit data type, and **l** denotes a 32-bit data type. If the first and/or second parameter is a 3-byte pointer, it will be passed in R16–R19 or R20–R22 respectively.

Stack parameters and layout

A function call creates a stack frame as follows:

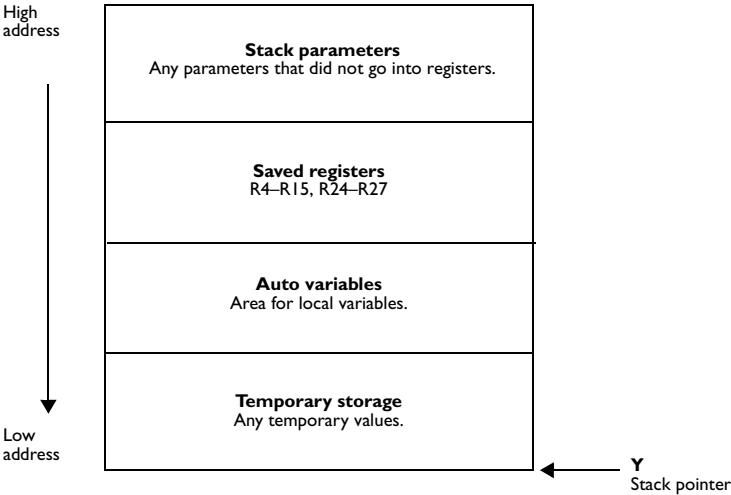


Figure 4: Stack image after the function call

Note that only registers that are used will be saved.

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

For the old calling convention, the registers available for returning values are R16–R19. The default calling convention can use the registers R16–R23 for returning values.

Return values	Passed in registers
8-bit values	R16
16-bit values	R17 : R16
24-bit values	R18 : R17 : R16
32-bit values	R19 : R18 : R17 : R16
64-bit values	R23 : R22 : R21 : R20 : R19 : R18 : R17 : R16

Table 29: Registers used for returning values

Note that the size of a returned pointer depends on the memory model in use; appropriate registers are used accordingly.

Stack layout at function exit

Normally, it is the responsibility of the called function to clean the stack. The only exception is for ellipse functions—functions with a variable argument list such as `printf`—for which it is the responsibility of the caller to clean the stack.

Return address handling

A function written in assembler language should, when finished, return to the caller. At a function call, the return address is automatically stored on the RSTACK (not the CSTACK).

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

RESTRICTIONS FOR SPECIAL FUNCTION TYPES

Interrupt functions

Interrupt functions differ from ordinary C functions in that:

- If used, flags and scratch registers are saved

- Calls to interrupt functions are made via interrupt vectors; direct calls are not allowed
- No arguments can be passed to an interrupt function
- Interrupt functions return by using the `RETI` function.

For more information about interrupt functions, see *Interrupt functions*, page 48.

Monitor functions

A monitor function causes interrupts to be disabled during execution of the function. At function entry, the status register `SREG` is saved and global interrupts are disabled. At function exit, the global interrupt enable bit (`I`) is restored in the `SREG` register, and thereby the interrupt status existing before the function call is also restored.

For more information about monitor functions, see *Monitor functions*, page 49.

EXAMPLES

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

Example 1

Assume this function declaration:

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

This function takes one parameter in the register `R17:R16`, and the return value is passed back to its caller in the register `R17:R16`.

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

```
subi    R16, FF
sbci    R17, FF
ret
```

Example 2

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

```
struct MyStruct
{
    long long mA;
    long long mB;
};

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

The calling function must reserve 16 bytes on the top of the stack and copy the contents of the `struct` to that location. The integer parameter `y` is passed in the register `R17:R16`. The return value is passed back to its caller in the register `R17:R16`.

Example 3

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

```
struct MyStruct
{
    long long mA;
    long long mB;
};

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

It is the responsibility of the calling function to allocate a memory location for the return value and pass a pointer to it as a hidden first parameter. The pointer to the location where the return value should be stored is passed in the first suitable register/register pair, which is `R16`, `R17:R16`, and `R18:R17:R16` for the Tiny, Small, Large, and Huge memory model, respectively. The parameter `x` is passed in `R19:R18`, `R19:R18`, and `R21:R20` for the Tiny, Small, Large, and Huge memory model, respectively.

Assume that the function instead was declared to return a pointer to the structure:

```
struct MyStruct *MyFunction(int x);
```

In this case, the return value is a scalar, so there is no hidden parameter. The parameter `x` is passed in `R17:R16`, and the return value is returned in `R16`, `R17:R16`, and `R18:R17:R16` for the Tiny, Small, and Large memory model, respectively.

FUNCTION DIRECTIVES

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

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

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

Call frame information

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

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

CFI DIRECTIVES

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

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

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

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

For AVR, the defined resources depend on which processor option you are using. You can find the resources in the list file section `CFI Names`.

CREATING ASSEMBLER SOURCE WITH CFI SUPPORT

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

- I Start with suitable C source code, for example:

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

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



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



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

Using C

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

C language overview

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

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

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

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

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

Inline assembler

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

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

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

The syntax is:

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

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

```
asm ("label:      nop\n"
    "              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 117.

Extensions overview

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

This is an overview of the available extensions:

- IAR C language extensions

For information about available language extensions, see *IAR C language extensions*, page 138. For more information about the extended keywords, see the chapter *Extended keywords*. For information about C++, the two levels of support for the language, and C++ language extensions; see the chapter *Using C++*.

- Pragma directives

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

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

- Preprocessor extensions

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

- Intrinsic functions

The intrinsic functions provide direct access to low-level processor operations and can be very useful in, for example, time-critical routines. The intrinsic functions compile into inline code, either as a single instruction or as a short sequence of instructions. For more information about using intrinsic functions, see *Mixing C and assembler*, page 117. For information about available functions, see the chapter *Intrinsic functions*.

- Library functions

The IAR DLIB Library provides the C and C++ library definitions that apply to embedded systems. For more information, see *IAR DLIB Library*, page 303. For information about AVR-specific library functions, see *AVR-specific library functions*, page 310.

Note: Any use of these extensions, except for the pragma directives, makes your source code inconsistent with Standard C.

ENABLING LANGUAGE EXTENSIONS

You can choose different levels of language conformance by means of project options:

Command line	IDE*	Description
<code>--strict</code>	Strict	All <i>IAR C language extensions</i> are disabled; errors are issued for anything that is not part of Standard C.
None	Standard	All extensions to <i>Standard C</i> are enabled, but no extensions for embedded systems programming. For information about extensions, see <i>IAR C language extensions</i> , page 138.
<code>-e</code>	Standard with IAR extensions	All <i>IAR C language extensions</i> are enabled.

Table 30: Language extensions

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

IAR C language extensions

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

- *Extensions for embedded systems programming*—extensions specifically tailored for efficient embedded programming for the specific microcontroller you are using, typically to meet memory restrictions
- *Relaxations to Standard C*—that is, the relaxation of some minor Standard C issues and also some useful but minor syntax extensions, see *Relaxations to Standard C*, page 140.

EXTENSIONS FOR EMBEDDED SYSTEMS PROGRAMMING

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

- Memory attributes, type attributes, and object attributes
For information about the related concepts, the general syntax rules, and for reference information, see the chapter *Extended keywords*.
- Placement at an absolute address or in a named segment
The @ operator or the directive #pragma location can be used for placing global and static variables at absolute addresses, or placing a variable or function in a named segment. For more information about using these features, see *Controlling data and function placement in memory*, page 164, and *location*, page 274.
- Alignment control
Each data type has its own alignment; for more information, see *Alignment*, page 231. If you want to change the alignment, the #pragma pack and #pragma data_alignment directives are available. If you want to check the alignment of an object, use the __ALIGNOF__() operator.

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

- __ALIGNOF__ (type)
- __ALIGNOF__ (expression)

In the second form, the expression is not evaluated.

- Anonymous structs and unions

C++ includes a feature called anonymous unions. The compiler allows a similar feature for both structs and unions in the C programming language. For more information, see *Anonymous structs and unions*, page 162.

- Bitfields and non-standard types

In Standard C, a bitfield must be of the type `int` or `unsigned int`. Using IAR C language extensions, any integer type or enumeration can be used. The advantage is that the struct will sometimes be smaller. For more information, see *Bitfields*, page 233.

- `static_assert()`

The construction `static_assert(const-expression, "message");` can be used in C/C++. The construction will be evaluated at compile time and if `const-expression` is false, a message will be issued including the `message` string.

- Parameters in variadic macros

Variadic macros are the preprocessor macro equivalents of `printf` style functions. The preprocessor accepts variadic macros with no arguments, which means if no parameter matches the `...` parameter, the comma is then deleted in the `" , ##__VA_ARGS__"` macro definition. According to Standard C, the `...` parameter must be matched with at least one argument.

Dedicated segment operators

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

<code>__segment_begin</code>	Returns the address of the first byte of the named segment.
<code>__segment_end</code>	Returns the address of the first byte <i>after</i> the named segment.
<code>__segment_size</code>	Returns the size of the named segment in bytes.

Note:

The operators can be used on named segments defined in the linker file.

These operators behave syntactically as if declared like:

```
void * __segment_begin(char const * segment)
void * __segment_end(char const * segment)
size_t * __segment_size(char const * segment)
```

When you use the `@` operator or the `#pragma location` directive to place a data object or a function in a user-defined segment in the linker configuration file, the segment

operators can be used for getting the start and end address of the memory range where the segments were placed.

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

Example

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

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

See also *segment*, page 279, and *location*, page 274.

RELAXATIONS TO STANDARD C

This section lists and briefly describes the relaxation of some Standard C issues and also some useful but minor syntax extensions:

- **Arrays of incomplete types**
An array can have an incomplete `struct`, `union`, or `enum` type as its element type. The types must be completed before the array is used (if it is), or by the end of the compilation unit (if it is not).
- **Forward declaration of `enum` types**
The extensions allow you to first declare the name of an `enum` and later resolve it by specifying the brace-enclosed list.
- **Accepting missing semicolon at the end of a `struct` or `union` specifier**
A warning—instead of an error—is issued if the semicolon at the end of a `struct` or `union` specifier is missing.
- **Null and `void`**
In operations on pointers, a pointer to `void` is always implicitly converted to another type if necessary, and a null pointer constant is always implicitly converted to a null pointer of the right type if necessary. In Standard C, some operators allow this kind of behavior, while others do not allow it.
- **Casting pointers to integers in static initializers**
In an initializer, a pointer constant value can be cast to an integral type if the integral type is large enough to contain it. For more information about casting pointers, see *Casting*, page 238.

- Taking the address of a register variable

In Standard C, it is illegal to take the address of a variable specified as a register variable. The compiler allows this, but a warning is issued.

- `long float` means `double`

The type `long float` is accepted as a synonym for `double`.

- Repeated `typedef` declarations

Redeclarations of `typedef` that occur in the same scope are allowed, but a warning is issued.

- Mixing pointer types

Assignment and pointer difference is allowed between pointers to types that are interchangeable but not identical; for example, `unsigned char *` and `char *`. This includes pointers to integral types of the same size. A warning is issued.

Assignment of a string constant to a pointer to any kind of character is allowed, and no warning is issued.

- Non-top level `const`

Assignment of pointers is allowed in cases where the destination type has added type qualifiers that are not at the top level (for example, `int **` to `int const **`).

Comparing and taking the difference of such pointers is also allowed.

- Non-lvalue arrays

A non-lvalue array expression is converted to a pointer to the first element of the array when it is used.

- Comments at the end of preprocessor directives

This extension, which makes it legal to place text after preprocessor directives, is enabled unless the strict Standard C mode is used. The purpose of this language extension is to support compilation of legacy code; we do *not* recommend that you write new code in this fashion.

- An extra comma at the end of `enum` lists

Placing an extra comma is allowed at the end of an `enum` list. In strict Standard C mode, a warning is issued.

- A label preceding a `}`

In Standard C, a label must be followed by at least one statement. Therefore, it is illegal to place the label at the end of a block. The compiler allows this, but issues a warning.

Note that this also applies to the labels of `switch` statements.

- Empty declarations

An empty declaration (a semicolon by itself) is allowed, but a remark is issued (provided that remarks are enabled).

- Single-value initialization

Standard C requires that all initializer expressions of static arrays, structs, and unions are enclosed in braces.

Single-value initializers are allowed to appear without braces, but a warning is issued. The compiler accepts this expression:

```
struct str
{
    int a;
} x = 10;
```

- Declarations in other scopes

External and static declarations in other scopes are visible. In the following example, the variable `y` can be used at the end of the function, even though it should only be visible in the body of the `if` statement. A warning is issued.

```
int test(int x)
{
    if (x)
    {
        extern int y;
        y = 1;
    }

    return y;
}
```

- Expanding function names into strings with the function as context

Use any of the symbols `__func__` or `__FUNCTION__` inside a function body to make the symbol expand into a string that contains the name of the current function. Use the symbol `__PRETTY_FUNCTION__` to also include the parameter types and return type. The result might, for example, look like this if you use the `__PRETTY_FUNCTION__` symbol:

```
"void func(char) "
```

These symbols are useful for assertions and other trace utilities and they require that language extensions are enabled, see `-e`, page 203.

- Static functions in function and block scopes

Static functions may be declared in function and block scopes. Their declarations are moved to the file scope.

- Numbers scanned according to the syntax for numbers

Numbers are scanned according to the syntax for numbers rather than the `pp-number` syntax. Thus, `0x123e+1` is scanned as three tokens instead of one valid token. (If the `--strict` option is used, the `pp-number` syntax is used instead.)

Using C++

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

Overview

Embedded C++ is a proper subset of the C++ programming language which is intended for embedded systems programming. It was defined by an industry consortium, the Embedded C++ Technical Committee. Performance and portability are particularly important in embedded systems development, which was considered when defining the language. EC++ offers the same object-oriented benefits as C++, but without some features that can increase code size and execution time in ways that are hard to predict.

EMBEDDED C++

The following C++ features are supported:

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

C++ features that are excluded are those that introduce overhead in execution time or code size that are beyond the control of the programmer. Also excluded are features added very late before Standard C++ was defined. Embedded C++ thus offers a subset of C++ which is efficient and fully supported by existing development tools.

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

- Templates
- Multiple and virtual inheritance
- Exception handling
- Runtime type information

- New cast syntax (the operators `dynamic_cast`, `static_cast`, `reinterpret_cast`, and `const_cast`)
- Namespaces
- The `mutable` attribute.

The exclusion of these language features makes the runtime library significantly more efficient. The Embedded C++ library furthermore differs from the full C++ library in that:

- The standard template library (STL) is excluded
- Streams, strings, and complex numbers are supported without the use of templates
- Library features which relate to exception handling and runtime type information (the headers `except`, `stdexcept`, and `typeinfo`) are excluded.

Note: The library is not in the `std` namespace, because Embedded C++ does not support namespaces.

EXTENDED EMBEDDED C++

IAR Systems' Extended EC++ is a slightly larger subset of C++ which adds these features to the standard EC++:

- Full template support
- Multiple and virtual inheritance
- Namespace support
- The `mutable` attribute
- The cast operators `static_cast`, `const_cast`, and `reinterpret_cast`.

All these added features conform to the C++ standard.

To support Extended EC++, this product includes a version of the standard template library (STL), in other words, the C++ standard chapters utilities, containers, iterators, algorithms, and some numerics. This STL is tailored for use with the Extended EC++ language, which means no exceptions, no multiple inheritance, and no support for runtime type information (`rtti`). Moreover, the library is not in the `std` namespace.

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

Enabling support for C++



In the compiler, the default language is C.

To compile files written in Embedded C++, you must use the `--ec++` compiler option. See `--ec++`, page 203.

To take advantage of *Extended* Embedded C++ features in your source code, you must use the `--eec++` compiler option. See `--eec++`, page 203.

For EC++ and EEC++, you must also use the IAR DLIB runtime library.



To enable EC++ or EEC++ in the IDE, choose **Project>Options>C/C++ Compiler>Language** and select the appropriate standard.

EC++ feature descriptions

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

USING IAR ATTRIBUTES WITH CLASSES

Static data members of C++ classes are treated the same way global variables are, and can have any applicable IAR type, memory, and object attribute.

Member functions are in general treated the same way free functions are, and can have any applicable IAR type, memory, and object attributes. Virtual member functions can only have attributes that are compatible with default function pointers, and constructors and destructors cannot have any such attributes.

The location operator `@` and the `#pragma location` directive can be used on static data members and with all member functions.

Example

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

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

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

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

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

The this pointer

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

Class memory

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

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

Example

```

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

    int mI;
};

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

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

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

```

Note: To place the C class in huge memory is not allowed, unless using the huge memory model, because a huge pointer cannot be implicitly converted into a __far pointer.

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

```
class __far C;
```

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

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

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

```
class __far D : public C
```

```

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

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

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

```

Note that the following is not allowed, unless using the huge memory model, because huge is not inside far memory:

```

class __huge G:public C
{
};

```

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

```

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

```

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

For more information about memory types, see *Memory types*, page 39.

FUNCTION TYPES

A function type with extern "C" linkage is compatible with a function that has C++ linkage.

Example

```
extern "C"
{
    typedef void (*FpC)(void);    // A C function typedef
}

typedef void (*FpCpp)(void);    // A C++ function typedef

FpC F1;
FpCpp F2;
void MyF(FpC);

void MyG()
{
    MyF(F1);                      // Always works
    MyF(F2);                      // FpCpp is compatible with FpC
}
```

NEW AND DELETE OPERATORS

There are operators for `new` and `delete` for each memory that can have a heap, that is, tiny, near, far, and huge memory.

```
// Assumes that there is a heap in both __near and __far memory
void __near *operator new __near(__near_size_t);
void __far *operator new __far (__far_size_t);
void operator delete(void __near *);
void operator delete(void __far *);

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

Use this syntax if you want to override both global and class-specific operator `new` and operator `delete` for any data memory.

Note that there is a special syntax to name the operator `new` functions for each memory, while the naming for the operator `delete` functions relies on normal overloading.

New and delete expressions

A `new` expression calls the operator `new` function for the memory of the type given. If a class, struct, or union type with a class memory is used, the class memory will determine the operator `new` function called. For example,

```

void MyF()
{
    // Calls operator new __huge(__huge_size_t)
    int __huge *p = new __huge int;

    // Calls operator new __near(__near_size_t)
    int __near *q = new int __near;

    // Calls operator new[] __near(__near_size_t)
    int __near *r = new __near int[10];

    // Calls operator new __huge(__huge_size_t)
    class __huge S
    {
    };
    S *s = new S;

    // Calls operator delete(void __huge *)
    delete p;
    // Calls operator delete(void __near *)
    delete s;

    int __huge *t = new __far int;
    delete t; // Error: Causes a corrupt heap
}

```

Note that the pointer used in a `delete` expression must have the correct type, that is, the same type as that returned by the `new` expression. If you use a pointer to the wrong memory, the result might be a corrupt heap.

USING STATIC CLASS OBJECTS IN INTERRUPTS

If interrupt functions use static class objects that need to be constructed (using constructors) or destroyed (using destructors), your application will not work properly if the interrupt occurs before the objects are constructed, or after the objects are destroyed.

To avoid this, make sure that these interrupts are not enabled until the static objects have been constructed, and are disabled when returning from `main` or calling `exit`. For information about system startup, see *System startup and termination*, page 84.

Function local static class objects are constructed the first time execution passes through their declaration, and are destroyed when returning from `main` or when calling `exit`.

USING NEW HANDLERS

To handle memory exhaustion, you can use the `set_new_handler` function.

If you do not call `set_new_handler`, or call it with a NULL new handler, and `operator new` fails to allocate enough memory, it will call `abort`. The `nothrow` variant of the new operator will instead return NULL.

If you call `set_new_handler` with a non-NULL new handler, the provided new handler will be called by `operator new` if `operator new` fails to allocate memory. The new handler must then make more memory available and return, or abort execution in some manner. The `nothrow` variant of `operator new` will never return NULL in the presence of a new handler.

TEMPLATES

Extended EC++ supports templates according to the C++ standard, but not the `export` keyword. The implementation uses a two-phase lookup which means that the keyword `typename` must be inserted wherever needed. Furthermore, at each use of a template, the definitions of all possible templates must be visible. This means that the definitions of all templates must be in include files or in the actual source file.

DEBUG SUPPORT IN C-SPY

C-SPY® has built-in display support for the STL containers. The logical structure of containers is presented in the watch views in a comprehensive way that is easy to understand and follow. For more information about displaying STL containers in the C-SPY debugger, see the *C-SPY® Debugging Guide for AVR*.

EEC++ feature description

This section describes features that distinguish Extended EC++ from EC++.

TEMPLATES

Templates and data memory attributes

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

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

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

Example

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

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

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

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

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

Example

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

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

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

Example

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

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

Non-type template parameters

It is allowed to have a reference to a memory type as a template parameter, even if pointers to that memory type are not allowed.

Example

```
extern int __near X;

template<__near int &y>
void Func()
{
    y = 17;
}

void Bar()
{
    Foo<X>();
}
```

The standard template library

The STL (standard template library) delivered with the product is tailored for Extended EC++, see *Extended Embedded C++*, page 144.

The containers in the STL, like `vector` and `map`, are memory attribute aware. This means that a container can be declared to reside in a specific memory type which has the following consequences:

- The container itself will reside in the chosen memory
- Allocations of elements in the container will use a heap for the chosen memory
- All references inside it use pointers to the chosen memory.

Example

```
#include <vector>
```

```

vector<int> D;                                // D placed in default
memory,                                     // using the default heap,
                                           // uses default pointers
vector<int __near> __near X;                 // X placed in near memory,
                                           // heap allocation from
                                           // near, uses pointers to
                                           // near memory
vector<int __huge> __near Y;                 // Y placed in near memory,
                                           // heap allocation from
                                           // huge, uses pointers to
                                           // huge memory

```

Note that this is illegal:

```
vector<int __near> __huge Z;
```

Note also that `map<key, T>`, `multimap<key, T>`, `hash_map<key, T>`, and `hash_multimap<key, T>` all use the memory of `T`. This means that the `value_type` of these collections will be `pair<key, const T> mem` where *mem* is the memory type of `T`. Supplying a key with a memory type is not useful.

Example

Note that two containers that only differ by the data memory attribute they use cannot be assigned to each other. Instead, the templated assign member method must be used.

```

#include <vector>

vector<int __near> X;
vector<int __huge> Y;

void MyF()
{
    // The templated assign member method will work
    X.assign(Y.begin(), Y.end());
    Y.assign(X.begin(), X.end());
}

```

VARIANTS OF CAST OPERATORS

In Extended EC++ these additional variants of C++ cast operators can be used:

```

const_cast<to>(from)
static_cast<to>(from)
reinterpret_cast<to>(from)

```

MUTABLE

The `mutable` attribute is supported in Extended EC++. A `mutable` symbol can be changed even though the whole class object is `const`.

NAMESPACE

The namespace feature is only supported in *Extended* EC++. This means that you can use namespaces to partition your code. Note, however, that the library itself is not placed in the `std` namespace.

THE STD NAMESPACE

The `std` namespace is not used in either standard EC++ or in Extended EC++. If you have code that refers to symbols in the `std` namespace, simply define `std` as nothing; for example:

```
#define std
```

You must make sure that identifiers in your application do not interfere with identifiers in the runtime library.

POINTER TO MEMBER FUNCTIONS

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

Example

```
class X
{
public:
    __nearfunc void F();
};

void (__nearfunc X::*PMF)(void) = &X::F;
```

C++ language extensions

When you use the compiler in any C++ mode and enable IAR language extensions, the following C++ language extensions are available in the compiler:

- In a friend declaration of a class, the `class` keyword can be omitted, for example:

```
class B;
class A
{
    friend B;           //Possible when using IAR language
                        //extensions
    friend class B; //According to the standard
};
```

- Constants of a scalar type can be defined within classes, for example:

```
class A
{
    const int mSize = 10; //Possible when using IAR language
                        //extensions
    int mArr[mSize];
};
```

According to the standard, initialized static data members should be used instead.

- In the declaration of a class member, a qualified name can be used, for example:

```
struct A
{
    int A::F(); // Possible when using IAR language extensions
    int G();    // According to the standard
};
```

- It is permitted to use an implicit type conversion between a pointer to a function with C linkage (`extern "C"`) and a pointer to a function with C++ linkage (`extern "C++"`), for example:

```
extern "C" void F(); // Function with C linkage
void (*PF)()         // PF points to a function with C++ linkage
    = &F; // Implicit conversion of function pointer.
```

According to the standard, the pointer must be explicitly converted.

- If the second or third operands in a construction that contains the `?` operator are string literals or wide string literals (which in C++ are constants), the operands can be implicitly converted to `char *` or `wchar_t *`, for example:

```
bool X;

char *P1 = X ? "abc" : "def";           //Possible when using IAR
                                         //language extensions
char const *P2 = X ? "abc" : "def";    //According to the standard
```

- Default arguments can be specified for function parameters not only in the top-level function declaration, which is according to the standard, but also in `typedef` declarations, in pointer-to-function function declarations, and in pointer-to-member function declarations.
- In a function that contains a non-static local variable and a class that contains a non-evaluated expression (for example a `sizeof` expression), the expression can reference the non-static local variable. However, a warning is issued.
- An anonymous union can be introduced into a containing class by a `typedef` name. It is not necessary to first declare the union. For example:

```
typedef union
{
    int i,j;
} U; // U identifies a reusable anonymous union.

class A
{
public:
    U; // OK -- references to A::i and A::j are allowed.
};
```

In addition, this extension also permits *anonymous classes* and *anonymous structs*, as long as they have no C++ features (for example, no static data members or member functions, and no non-public members) and have no nested types other than other anonymous classes, structs, or unions. For example:

```
struct A
{
    struct
    {
        int i,j;
    }; // OK -- references to A::i and A::j are allowed.
};
```

- The friend class syntax allows nonclass types as well as class types expressed through a `typedef` without an elaborated type name. For example:

```
typedef struct S ST;

class C
{
public:
    friend S; // Okay (requires S to be in scope)
    friend ST; // Okay (same as "friend S;")
    // friend S const; // Error, cv-qualifiers cannot
                        // appear directly
};
```

Note: If you use any of these constructions without first enabling language extensions, errors are issued.

Efficient coding for embedded applications

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

The topics discussed are:

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

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

Selecting data types

For efficient treatment of data, you should consider the data types used and the most efficient placement of the variables. This section provides useful information for efficient treatment of data:

- Locating strings in ROM, RAM or flash
- Using efficient data types
- Floating-point types
- Memory model and memory attributes for data
- Using the best pointer type
- Anonymous structs and unions.

LOCATING STRINGS IN ROM, RAM OR FLASH

With the IAR C/C++ Compiler for AVR there are three possible locations for storing strings:

- In external ROM in the data memory space
- In internal RAM in the data memory space
- In flash in the code memory space.

To read more about this, see *Initialized data*, page 59.

Locating strings in flash

This can be done on individual strings or for the whole application/file using the option `--string_literals_in_flash`. Examples on how to locate individual strings into flash:

```
__flash char str1[] = "abcdef";
__flash char str2[] = "ghi";
__flash char __flash * pVar[] = { str1, str2 };
```

String literals cannot be put in flash automatically, but you can use a local static variable instead:

```
#include <pgmspace.h>
void f (int i)
{
    static __flash char sl[] = "%d cookies\n";
    printf_P(sl, i);
}
```

This does not result in more code compared to allowing string literals in flash.

To use flash strings, you must use alternative library routines that expect flash strings. A few such alternative functions are provided—they are declared in the `pgmspace.h` header file. They are flash alternatives for some common C library functions with an extension `_P`. For your own code, you can always use the `__flash` keyword when passing the strings between functions.

For reference information about the alternative functions, see *AVR-specific library functions*, page 310.

USING EFFICIENT DATA TYPES

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

- Use small and unsigned data types, (`unsigned char` and `unsigned short`) unless your application really requires signed values.

- Try to avoid 64-bit data types, such as `double` and `long long`.
- Bitfields with sizes other than 1 bit should be avoided because they will result in inefficient code compared to bit operations.
- When using arrays, it is more efficient if the type of the index expression matches the index type of the memory of the array. For `__near`, the index type is `int`.
- Using floating-point types on a microprocessor without a math co-processor is very inefficient, both in terms of code size and execution speed.
- Declaring a pointer parameter to point to `const` data tells the calling function that the data pointed to will not change, which opens for better optimizations.

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

FLOATING-POINT TYPES

Using floating-point types on a microprocessor without a math coprocessor is very inefficient, both in terms of code size and execution speed. Thus, you should consider replacing code that uses floating-point operations with code that uses integers, because these are more efficient.

The compiler supports two floating-point formats—32 and 64 bits. The 32-bit floating-point type `float` is more efficient in terms of code size and execution speed. However, the 64-bit format `double` supports higher precision and larger numbers.

In the compiler, the floating-point type `float` always uses the 32-bit format. The format used by the `double` floating-point type depends on the setting of the `--64bit_doubles` compiler option (**Use 64-bit doubles**).

By default, a *floating-point constant* in the source code is treated as being of the type `double`. This can cause innocent-looking expressions to be evaluated in double precision. In the example below `a` is converted from a `float` to a `double`, the `double` constant `1.0` is added and the result is converted back to a `float`:

```
double Test(float a)
{
    return a + 1.0;
}
```

To treat a floating-point constant as a `float` rather than as a `double`, add the suffix `f` to it, for example:

```
double Test(float a)
{
    return a + 1.0f;
}
```

For more information about floating-point types, see *Floating-point types*, page 235.

MEMORY MODEL AND MEMORY ATTRIBUTES FOR DATA

For many applications it is sufficient to use the memory model feature to specify the default memory for the data objects. However, for individual objects it might be necessary to specify other memory attributes in certain cases, for example:

- An application where some global variables are accessed from a large number of locations. In this case they can be declared to be placed in memory with a smaller pointer type
- An application where all data, with the exception of one large chunk of data, fits into the region of one of the smaller memory types
- Data that must be placed at a specific memory location.

The IAR C/C++ Compiler for AVR provides memory attributes for placing data objects in the different memory spaces, and for the DATA and CODE space (flash) there are different memory attributes for placing data objects in different memory types, see *Using data memory attributes*, page 39.

Efficient usage of memory type attributes can significantly reduce the application size.

For details about the memory types, see *Memory types*, page 39.

USING THE BEST POINTER TYPE

The generic pointers, pointers declared `__generic`, can point to all memory spaces, which makes them simple and also tempting to use. However, they carry a cost in that special code is needed before each pointer access to check which memory a pointer points to and taking the appropriate actions. Use the smallest pointer type you can, and avoid any generic pointers unless necessary. For details about available pointer types, see *Pointer types*, page 237.

ANONYMOUS STRUCTS AND UNIONS

When a structure or union is declared without a name, it becomes anonymous. The effect is that its members will only be seen in the surrounding scope.

Anonymous structures are part of the C++ language; however, they are not part of the C standard. In the IAR C/C++ Compiler for AVR 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 203, for additional information.

Example

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

```
struct S
{
    char mTag;
    union
    {
        long mL;
        float mF;
    };
} St;

void F(void)
{
    St.mL = 5;
}
```

The member names must be unique in the surrounding scope. Having an anonymous struct or union at file scope, as a global, external, or static variable is also allowed. This could for instance be used for declaring I/O registers, as in this example:

```
__no_init volatile
union
{
    unsigned char IOPORT;
    struct
    {
        unsigned char way: 1;
        unsigned char out: 1;
    };
} @ 8;

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

This declares an I/O register byte `IOPORT` at address 8. The I/O register has 2 bits declared, `way` and `out`. Note that both the inner structure and the outer union are anonymous.

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

Controlling data and function placement in memory

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

- Memory models

Use the different compiler options for memory models to take advantage of the different addressing modes available for the microcontroller and thereby also place data objects in different parts of memory. For more information about memory models, see *Memory models*, page 38, and *Function storage*, page 47, respectively.

- Memory attributes

Use memory attributes to override the default addressing mode and placement of individual functions and data objects. For more information about memory attributes for data and functions, see *Using data memory attributes*, page 39, and *Using function memory attributes*, page 47, 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 to conform to external requirements, for example to populate interrupt vectors or other hardware tables. Note that it is not possible to use this notation for absolute placement of individual functions.

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

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

- The `--segment` option

Use the `--segment` option to place functions and/or data objects in named segments, which is useful, for example, if you want to direct them to different fast or slow memories. In contrast to using the `@` operator or the `#pragma location` directive, there are no restrictions on what types of variables that can be placed in named segments when using the `--segment` option. For more information about the `--segment` option, see *--segment*, page 222.

At compile time, data and functions are placed in different segments, see *Data segments*, page 57, and *Code segments*, page 65, respectively. At link time, one of the most important functions of the linker is to assign load addresses to the various segments used by the application. All segments, except for the segments holding absolute located data, are automatically allocated to memory according to the specifications of memory ranges in the linker configuration file, see *Placing segments in memory*, page 54.

DATA PLACEMENT AT AN ABSOLUTE LOCATION

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

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

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

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

Examples

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

```
__no_init volatile char alpha @ 0xFF2000; /* OK */
```

The next example contains two `const` declared objects. The first one is not initialized, and the second one is initialized to a specific value. Both objects are placed in ROM. This is useful for configuration parameters, which are accessible from an external

interface. Note that in the second case, the compiler is not obliged to actually read from the variable, because the value is known.

```
#pragma location=0xFF2002
__no_init const int beta;           /* OK */

const int gamma @ 0xFF2004 = 3;     /* OK */
```

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

These examples show incorrect usage:

```
int delta @ 0xFF2006;               /* Error, neither */
                                   /* "__no_init" nor "const". */
```

C++ considerations

In C++, module scoped `const` variables are static (module local), whereas in C they are global. This means that each module that declares a certain `const` variable will contain a separate variable with this name. If you link an application with several such modules all containing (via a header file), for instance, the declaration:

```
volatile const __no_init int x @ 0x100;    /* Bad in C++ */
```

the linker will report that more than one variable is located at address 0x100.

To avoid this problem and make the process the same in C and C++, you should declare these variables `extern`, for example:

```
/* The extern keyword makes x public. */
extern volatile const __no_init int x @ 0x100;
```

Note: C++ static member variables can be placed at an absolute address just like any other static variable.

DATA AND FUNCTION PLACEMENT IN SEGMENTS

The following methods 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.
- The `--segment` option can be used for placing variables and functions, which are parts of the whole compilation unit, in named segments.

C++ static member variables can be placed in named segments just like any other static variable.

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

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

The location of the segments can be controlled from the linker configuration file.

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

Examples of placing variables in named segments

In the following examples, a data object is placed in a user-defined segment.

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

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

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

As usual, you can use memory attributes to direct the variable to a non-default memory (and then also place the segment accordingly in the linker configuration file):

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

This example shows incorrect usage:

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

Examples of placing functions in named segments

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

void g(void) @ "MY_FUNCTIONS"
{
}

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

Specify a memory attribute to direct the function to a specific memory, and then modify the segment placement in the linker configuration file accordingly:

```
__farfunc void f(void) @ "MY_FARFUNC_FUNCTIONS";
```

Controlling compiler optimizations

The compiler performs many transformations on your application to generate the best possible code. Examples of such transformations are storing values in registers instead of memory, removing superfluous code, reordering computations in a more efficient order, and replacing arithmetic operations by cheaper operations.

The linker should also be considered an integral part of the compilation system, because some optimizations are performed by the linker. For instance, all unused functions and variables are removed and not included in the final output.

SCOPE FOR PERFORMED OPTIMIZATIONS

You can decide whether optimizations should be performed on your whole application or on individual files. By default, the same types of optimizations are used for an entire project, but you should consider using different optimization settings for individual files. For example, put code that must execute very quickly into a separate file and compile it for minimal execution time, and the rest of the code for minimal code size. This will give a small program, which is still fast enough where it matters.

You can also exclude individual functions from the performed optimizations. The `#pragma optimize` directive allows you to either lower the optimization level, or specify another type of optimization to be performed. See *optimize*, page 275, for information about the pragma directive.

MULTI-FILE COMPILATION UNITS

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

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

If the whole application is compiled as one compilation unit, it is very useful to make the compiler also discard unused public functions and variables before the interprocedural optimizations are performed. Doing this limits the scope of the

optimizations to functions and variables that are actually used. For more information, see `--discard_unused_publics`, page 201.

OPTIMIZATION LEVELS

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

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

Table 31: 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 170.

A high level of optimization might result in increased compile time, and will most likely also make debugging more difficult, because it is less clear how the generated code relates to the source code. For example, at the low, medium, and high optimization levels, variables do not live through their entire scope, which means processor registers used for storing variables can be reused immediately after they were last used. Due to this, the C-SPY Watch window might not be able to display the value of the variable throughout its scope. At any time, if you experience difficulties when debugging your code, try lowering the optimization level.

SPEED VERSUS SIZE

At the high optimization level, the compiler balances between size and speed optimizations. However, it is possible to fine-tune the optimizations explicitly for either size or speed. They only differ in what thresholds that are used; speed will trade size for speed, whereas size will trade speed for size. Note that one optimization sometimes enables other optimizations to be performed, and an application might in some cases become smaller even when optimizing for speed rather than size.

FINE-TUNING ENABLED TRANSFORMATIONS

At each optimization level you can disable some of the transformations individually. To disable a transformation, use either the appropriate option, for instance the command line option `--no_inline`, alternatively its equivalent in the IDE **Function inlining**, or the `#pragma optimize` directive. These transformations can be disabled individually:

- Common subexpression elimination
- Loop unrolling
- Function inlining
- Code motion
- Type-based alias analysis
- Static clustering
- 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.

For more information about the command line option, see `--no_cse`, page 213.

Function inlining

Function inlining means that a simple function, whose definition is known at compile time, is integrated into the body of its caller to eliminate the overhead of the call. This optimization, which is performed at optimization level High, normally reduces execution time, but the resulting code might be difficult to debug.

The compiler heuristically decides which functions to inline. Different heuristics are used when optimizing for speed, size, or when balancing between size and speed. Normally, code size does not increase when optimizing for size. To control the heuristics

for individual functions, use the `#pragma inline` directive or the Standard C `inline` keyword.

If you do not want to disable inlining for a whole module, use `#pragma inline=never` on an individual function instead.

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

For more information about the command line option, see `--no_inline`, page 214. For information about the pragma directive, see `inline`, page 272.

Code motion

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

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

For more information about the command line option, see `--no_code_motion`, page 213.

Type-based alias analysis

When two or more pointers reference the same memory location, these pointers are said to be *aliases* for each other. The existence of aliases makes optimization more difficult because it is not necessarily known at compile time whether a particular value is being changed.

Type-based alias analysis optimization assumes that all accesses to an object are performed using its declared type or as a `char` type. This assumption lets the compiler detect whether pointers can reference the same memory location or not.

Type-based alias analysis is performed at optimization level High. For application code conforming to standard C or C++ application code, this optimization can reduce code size and execution time. However, non-standard C or C++ code might result in the compiler producing code that leads to unexpected behavior. Therefore, it is possible to turn this optimization off.

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

For more information about the command line option, see `--no_tbaa`, page 215.

Example

```
short F(short *p1, long *p2)
{
    *p2 = 0;
    *p1 = 1;
```

```
    return *p2;
}
```

With type-based alias analysis, it is assumed that a write access to the `short` pointed to by `p1` cannot affect the `long` value that `p2` points to. Thus, it is known at compile time that this function returns 0. However, in non-standard-conforming C or C++ code these pointers could overlap each other by being part of the same union. If you use explicit casts, you can also force pointers of different pointer types to point to the same memory location.

Static clustering

When static clustering is enabled, static and global variables that are defined within the same module are arranged so that variables that are accessed in the same function are stored close to each other. This makes it possible for the compiler to use the same base pointer for several accesses.

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

For more information about the command line option, see `--no_clustering`, page 212.

Cross call

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

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

For more information about related command line options, see `--no_cross_call`, page 213, `--do_cross_call`, page 202, and `--cross_call_passes`, page 196.

Facilitating good code generation

This section contains hints on how to help the compiler generate good code, for example:

- Using efficient addressing modes
- Helping the compiler optimize
- Generating more useful error message.

WRITING OPTIMIZATION-FRIENDLY SOURCE CODE

The following is a list of programming techniques that will, when followed, enable the compiler to better optimize the application.

- Local variables—auto variables and parameters—are preferred over static or global variables. The reason is that the optimizer must assume, for example, that called functions can modify non-local variables. When the life spans for local variables end, the previously occupied memory can then be reused. Globally declared variables will occupy data memory during the whole program execution.
- Avoid taking the address of local variables using the `&` operator. This is inefficient for two main reasons. First, the variable must be placed in memory, and thus cannot be placed in a processor register. This results in larger and slower code. Second, the optimizer can no longer assume that the local variable is unaffected over function calls.
- Module-local variables—variables that are declared static—are preferred over global variables (non-static). Also avoid taking the address of frequently accessed static variables.
- The compiler is capable of inlining functions, see *Function inlining*, page 170. To maximize the effect of the inlining transformation, it is good practice to place the definitions of small functions called from more than one module in the header file rather than in the implementation file. Alternatively, you can use multi-file compilation. For more information, see *Multi-file compilation units*, page 168.
- Avoid using inline assembler. Instead, try writing the code in C or C++, use intrinsic functions, or write a separate module in assembler language. For more information, see *Mixing C and assembler*, page 117.

SAVING STACK SPACE AND RAM MEMORY

The following is a list of programming techniques that will, when followed, save memory and stack space:

- If stack space is limited, avoid long call chains and recursive functions.
- Avoid using large non-scalar types, such as structures, as parameters or return type. To save stack space, you should instead pass them as pointers or, in C++, as references.

FUNCTION PROTOTYPES

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

- Prototyped
- Kernighan & Ritchie C (K&R C)

Both styles are included in the C standard; however, it is recommended to use the prototyped style, since it makes it easier for the compiler to find problems in the code. Using the prototyped style will also make it possible to generate more efficient code, since type promotion (implicit casting) is not needed. The K&R style is only supported for compatibility reasons.

To make the compiler verify that all functions have proper prototypes, use the compiler option **Require prototypes** (`--require_prototypes`).

Prototyped style

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

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

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

Kernighan & Ritchie style

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

For example:

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

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

INTEGER TYPES AND BIT NEGATION

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

In some cases there might be warnings (for example, for constant conditional or pointless comparison), in others just a different result than what is expected. Under certain circumstances the compiler might warn only at higher optimizations, for example, if the compiler relies on optimizations to identify some instances of constant

conditionals. In this example an 8-bit character, a 16-bit integer, and two's complement is assumed:

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

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

PROTECTING SIMULTANEOUSLY ACCESSED VARIABLES

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

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

For sequences of accesses to variables that you do not want to be interrupted, use the `__monitor` keyword. This must be done for both write *and* read sequences, otherwise you might end up reading a partially updated variable. Accessing a small-sized `volatile` variable can be an atomic operation, but you should not rely on it unless you continuously study the compiler output. It is safer to use the `__monitor` keyword to ensure that the sequence is an atomic operation. For more information, see *__monitor*, page 257.

For more information about the `volatile` type qualifier and the rules for accessing volatile objects, see *Declaring objects volatile*, page 240.



Protecting the eeprom write mechanism

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

ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for several AVR devices are included in the IAR product installation. The header files are named `iodevice.h` and define the processor-specific special function registers (SFRs).

To enable bit definitions in IAR Embedded Workbench, select the option **General Options>System>Enable bit definitions in I/O include files**.

Note: Each header file contains one section used by the compiler, and one section used by the assembler.

SFRs with bitfields are declared in the header file. This example is from `iom128.h`:

```
__io union
{
    unsigned char PORTE;    /* The sfrb as 1 byte */
    struct
    {
        unsigned char PORTE_Bit0:1,
                        PORTE_Bit1:1,
                        PORTE_Bit2:1,
                        PORTE_Bit3:1,
                        PORTE_Bit4:1,
                        PORTE_Bit5:1,
                        PORTE_Bit6:1,
                        PORTE_Bit7:1;

    };
} @ 0x1F;
```

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 */
PORTE = 0x12;

/* Bitfield accesses */
PORTE_Bit0 = 1;
```

You can also use the header files as templates when you create new header files for other AVR devices. For information about the `@` operator, see *Located data*, page 65.

NON-INITIALIZED VARIABLES

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

The compiler supports the declaration of variables that will not be initialized, using the `__no_init` type modifier. They can be specified either as a keyword or using the `#pragma object_attribute` directive. The compiler places such variables in a

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

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

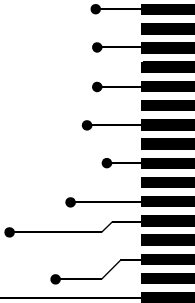
Variables declared using the `__no_init` keyword could, for example, be large input buffers or mapped to special RAM that keeps its content even when the application is turned off.

For information about the `__no_init` keyword, see page 259. Note that to use this keyword, language extensions must be enabled; see `-e`, page 203. For information about the `#pragma object_attribute`, see page 275.

Part 2. Reference information

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

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





External interface details

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

Invocation syntax

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

COMPILER INVOCATION SYNTAX

The invocation syntax for the compiler is:

```
iccavr [options] [sourcefile] [options]
```

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

```
iccavr prog.c --debug
```

The source file can be a C or C++ file, typically with the filename extension `c` or `cpp`, respectively. If no filename extension is specified, the file to be compiled must have the extension `c`.

Generally, the order of options on the command line, both relative to each other and to the source filename, is *not* significant. There is, however, one exception: when you use the `-I` option, the directories are searched in the same order as they are specified on the command line.

If you run the compiler from the command line without any arguments, the compiler version number and all available options including brief descriptions are directed to `stdout` and displayed on the screen.

PASSING OPTIONS

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

- Directly from the command line
Specify the options on the command line after the `iccavr` command, either before or after the source filename; see *Invocation syntax*, page 181.

- Via environment variables
The compiler automatically appends the value of the environment variables to every command line; see *Environment variables*, page 182.
- Via a text file, using the `-f` option; see *-f*, page 206.

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

ENVIRONMENT VARIABLES

These environment variables can be used with the compiler:

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

Table 32: Compiler environment variables

Include file search procedure

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

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

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

it searches these directories for the file to include:
 - 1 The directories specified with the `-I` option, in the order that they were specified, see *-I*, page 208.
 - 2 The directories specified using the `C_INCLUDE` environment variable, if any; see *Environment variables*, page 182.
 - 3 The automatically set up library system include directories. See *--clib*, page 195, *--dlib*, page 201, and *--dlib_config*, page 202.
- If the compiler encounters the name of an #include file in double quotes, for example:

```
#include "vars.h"
```

it searches the directory of the source file in which the #include statement occurs, and then performs the same sequence as for angle-bracketed filenames.

If there are nested `#include` files, the compiler starts searching the directory of the file that was last included, iterating upwards for each included file, searching the source file directory last. For example:

```
src.c in directory dir\src
#include "src.h"
...
src.h in directory dir\include
#include "config.h"
...
```

When `dir\exe` is the current directory, use this command for compilation:

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

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

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

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

For information about the syntax for including header files, see *Overview of the preprocessor*, page 293.

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 `.o`.
- Optional list files
Various kinds of list files can be specified using the compiler option `-l`, see *-l*, page 209. By default, these files will have the filename extension `.lst`.

- Optional preprocessor output files
A preprocessor output file is produced when you use the `--preprocess` option; by default, the file will have the filename extension `.i`.
- Diagnostic messages
Diagnostic messages are directed to the standard error stream and displayed on the screen, and printed in an optional list file. For more information about diagnostic messages, see *Diagnostics*, page 185.
- 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 184.
- Size information
Information about the generated amount of bytes for functions and data for each memory is directed to the standard output stream and displayed on the screen. Some of the bytes might be reported as *shared*.
Shared objects are functions or data objects that are shared between modules. If any of these occur in more than one module, only one copy is retained. For example, in some cases inline functions are not inlined, which means that they are marked as shared, because only one instance of each function will be included in the final application. This mechanism is sometimes also used for compiler-generated code or data not directly associated with a particular function or variable, and when only one instance is required in the final application.

Error return codes

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

These command line error codes are supported:

Code	Description
0	Compilation successful, but there might have been warnings.
1	Warnings were produced and the option <code>--warnings_affect_exit_code</code> was used.
2	Errors occurred.
3	Fatal errors occurred, making the compiler abort.
4	Internal errors occurred, making the compiler abort.

Table 33: Error return codes

Diagnostics

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

MESSAGE FORMAT

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

```
filename,linenumber level[tag]: message
```

with these elements:

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

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

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

SEVERITY LEVELS

The diagnostic messages are divided into different levels of severity:

Remark

A diagnostic message that is produced when the compiler finds a source code construct that can possibly lead to erroneous behavior in the generated code. Remarks are by default not issued, but can be enabled, see `--remarks`, page 221.

Warning

A diagnostic message that is produced when the compiler finds a potential programming error or omission which is of concern, but which does not prevent completion of the compilation. Warnings can be disabled by use of the command line option `--no_warnings`, see page 217.

Error

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

Fatal error

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

SETTING THE SEVERITY LEVEL

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

See *Summary of compiler options*, page 190, for information about the compiler options that are available for setting severity levels.

See the chapter *Pragma directives*, for information about the pragma directives that are available for setting severity levels.

INTERNAL ERROR

An internal error is a diagnostic message that signals that there was a serious and unexpected failure due to a fault in the compiler. It is produced using this form:

Internal error: *message*

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

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

Compiler options

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

Options syntax

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



See the *IAR Embedded Workbench® IDE Project Management and Building Guide* for information about the compiler options available in the IDE and how to set them.

TYPES OF OPTIONS

There are two *types of names* for command line options, *short* names and *long* names. Some options have both.

- A short option name consists of one character, and it can have parameters. You specify it with a single dash, for example `-e`
- A long option name consists of one or several words joined by underscores, and it can have parameters. You specify it with double dashes, for example `--char_is_signed`.

For information about the different methods for passing options, see *Passing options*, page 181.

RULES FOR SPECIFYING PARAMETERS

There are some general syntax rules for specifying option parameters. First, the rules depending on whether the parameter is *optional* or *mandatory*, and whether the option has a short or a long name, are described. Then, the rules for specifying filenames and directories are listed. Finally, the remaining rules are listed.

Rules for optional parameters

For options with a short name and an optional parameter, any parameter should be specified without a preceding space, for example:

`-O or -Oh`

For options with a long name and an optional parameter, any parameter should be specified with a preceding equal sign (=), for example:

```
--misrac2004=n
```

Rules for mandatory parameters

For options with a short name and a mandatory parameter, the parameter can be specified either with or without a preceding space, for example:

```
-I..src or -I ..src
```

For options with a long name and a mandatory parameter, the parameter can be specified either with a preceding equal sign (=) or with a preceding space, for example:

```
--diagnostics_tables=MyDiagnostics.lst
```

or

```
--diagnostics_tables MyDiagnostics.lst
```

Rules for options with both optional and mandatory parameters

For options taking both optional and mandatory parameters, the rules for specifying the parameters are:

- For short options, optional parameters are specified without a preceding space
- For long options, optional parameters are specified with a preceding equal sign (=)
- For short and long options, mandatory parameters are specified with a preceding space.

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

```
-lA MyList.lst
```

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

```
--preprocess=n PreprocOutput.lst
```

Rules for specifying a filename or directory as parameters

These rules apply for options taking a filename or directory as parameters:

- Options that take a filename as a parameter can optionally take a file path. The path can be relative or absolute. For example, to generate a listing to the file *List.lst* in the directory *..\listings*:

```
iccavr prog.c -l ..\listings\List.lst
```

- For options that take a filename as the destination for output, the parameter can be specified as a path without a specified filename. The compiler stores the output in that directory, in a file with an extension according to the option. The filename will be the same as the name of the compiled source file, unless a different name was specified with the option `-o`, in which case that name is used. For example:

```
iccavr prog.c -l ../listings\
```

The produced list file will have the default name `../listings\prog.lst`

- The *current directory* is specified with a period (`.`). For example:
- ```
iccavr prog.c -l .
```
- `/` can be used instead of `\` as the directory delimiter.
  - By specifying `-`, input files and output files can be redirected to the standard input and output stream, respectively. For example:

```
iccavr prog.c -l -
```

## Additional rules

These rules also apply:

- When an option takes a parameter, the parameter cannot start with a dash (`-`) followed by another character. Instead, you can prefix the parameter with two dashes; this example will create a list file called `-r`:

```
iccavr prog.c -l ---r
```

- For options that accept multiple arguments of the same type, the arguments can be provided as a comma-separated list (without a space), for example:

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

Alternatively, the option can be repeated for each argument, for example:

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

# Summary of compiler options

This table summarizes the compiler command line options:

| Command line option      | Description                                                                                                 |
|--------------------------|-------------------------------------------------------------------------------------------------------------|
| --64bit_doubles          | Forces the compiler to use 64-bit doubles                                                                   |
| --64k_flash              | Specifies a maximum of 64 Kbytes flash memory                                                               |
| --c89                    | Specifies the C89 dialect                                                                                   |
| --char_is_signed         | Treats char as signed                                                                                       |
| --char_is_unsigned       | Treats char as unsigned                                                                                     |
| --clib                   | Uses the system include files for the CLIB library                                                          |
| --cpu                    | Specifies a specific device                                                                                 |
| --cross_call_passes      | Cross call optimization                                                                                     |
| -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                                                                               |
| --disable_direct_mode    | Disables direct addressing mode                                                                             |
| --discard_unused_publics | Discards unused public symbols                                                                              |
| --dlib                   | Uses the system include files for the DLIB library                                                          |
| --dlib_config            | Uses the system include files for the DLIB library and determines which configuration of the library to use |
| --do_cross_call          | Forces the compiler to run the cross call optimizer.                                                        |
| -e                       | Enables language extensions                                                                                 |
| --ec++                   | Specifies Embedded C++                                                                                      |
| --eec++                  | Specifies Extended Embedded C++                                                                             |
| --eegr_address           | Defines the EECR address                                                                                    |
| --eeprom_size            | Specifies the EEPROM size                                                                                   |
| --enable_external_bus    | Adds code to enable external data bus                                                                       |

Table 34: Compiler options summary

| Command line option                           | Description                                                                                                                                                                           |
|-----------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>--enable_multibytes</code>              | Enables support for multibyte characters in source files                                                                                                                              |
| <code>--enhanced_core</code>                  | Enables enhanced instruction set                                                                                                                                                      |
| <code>--error_limit</code>                    | Specifies the allowed number of errors before compilation stops                                                                                                                       |
| <code>-f</code>                               | Extends the command line                                                                                                                                                              |
| <code>--force_switch_type</code>              | Forces the switch type                                                                                                                                                                |
| <code>--guard_calls</code>                    | Enables guards for function static variable initialization                                                                                                                            |
| <code>--header_context</code>                 | Lists all referred source files and header files                                                                                                                                      |
| <code>-I</code>                               | Specifies include file path                                                                                                                                                           |
| <code>--initializers_in_flash</code>          | Places aggregate initializers in flash memory                                                                                                                                         |
| <code>-l</code>                               | Creates a list file                                                                                                                                                                   |
| <code>--library_module</code>                 | Creates a library module                                                                                                                                                              |
| <code>--lock_regs</code>                      | Locks registers                                                                                                                                                                       |
| <code>-m</code>                               | Specifies a memory model                                                                                                                                                              |
| <code>--macro_positions_in_diagnostics</code> | Obtains positions inside macros in diagnostic messages                                                                                                                                |
| <code>--memory_model</code>                   | Specifies a memory model                                                                                                                                                              |
| <code>--mfc</code>                            | Enables multi-file compilation                                                                                                                                                        |
| <code>--misrac</code>                         | Enables error messages specific to MISRA-C:1998. This option is a synonym of <code>--misrac1998</code> and is only available for backwards compatibility.                             |
| <code>--misrac1998</code>                     | Enables error messages specific to MISRA-C:1998. See the <i>IAR Embedded Workbench® MISRA C:1998 Reference Guide</i> .                                                                |
| <code>--misrac2004</code>                     | Enables error messages specific to MISRA-C:2004. See the <i>IAR Embedded Workbench® MISRA C:2004 Reference Guide</i> .                                                                |
| <code>--misrac_verbose</code>                 | Enables verbose logging of MISRA C checking. See the <i>IAR Embedded Workbench® MISRA C:1998 Reference Guide</i> or the <i>IAR Embedded Workbench® MISRA C:2004 Reference Guide</i> . |
| <code>--module_name</code>                    | Sets the object module name                                                                                                                                                           |
| <code>--no_clustering</code>                  | Disables static clustering optimizations                                                                                                                                              |

Table 34: Compiler options summary (Continued)

| Command line option                       | Description                                                                                                |
|-------------------------------------------|------------------------------------------------------------------------------------------------------------|
| <code>--no_code_motion</code>             | Disables code motion optimization                                                                          |
| <code>--no_cross_call</code>              | Disables cross-call optimization                                                                           |
| <code>--no_cse</code>                     | Disables common subexpression elimination                                                                  |
| <code>--no_inline</code>                  | Disables function inlining                                                                                 |
| <code>--no_path_in_file_macros</code>     | Removes the path from the return value of the symbols <code>__FILE__</code> and <code>__BASE_FILE__</code> |
| <code>--no_rampd</code>                   | Uses <code>RAMPZ</code> instead of <code>RAMPD</code>                                                      |
| <code>--no_static_destruction</code>      | Disables destruction of C++ static variables at program exit                                               |
| <code>--no_system_include</code>          | Disables the automatic search for system include files                                                     |
| <code>--no_tbaa</code>                    | Disables type-based alias analysis                                                                         |
| <code>--no_typedefs_in_diagnostics</code> | Disables the use of typedef names in diagnostics                                                           |
| <code>--no_ubrof_messages</code>          | Minimizes object file size                                                                                 |
| <code>--no_unroll</code>                  | Disables loop unrolling                                                                                    |
| <code>--no_warnings</code>                | Disables all warnings                                                                                      |
| <code>--no_wrap_diagnostics</code>        | Disables wrapping of diagnostic messages                                                                   |
| <code>-O</code>                           | Sets the optimization level                                                                                |
| <code>-o</code>                           | Sets the object filename. Alias for <code>--output</code> .                                                |
| <code>--omit_types</code>                 | Excludes type information                                                                                  |
| <code>--only_stdout</code>                | Uses standard output only                                                                                  |
| <code>--output</code>                     | Sets the object filename                                                                                   |
| <code>--predef_macros</code>              | Lists the predefined symbols.                                                                              |
| <code>--preinclude</code>                 | Includes an include file before reading the source file                                                    |
| <code>--preprocess</code>                 | Generates preprocessor output                                                                              |
| <code>--public_equ</code>                 | Defines a global named assembler label                                                                     |
| <code>-r</code>                           | Generates debug information. Alias for <code>--debug</code> .                                              |
| <code>--relaxed_fp</code>                 | Relaxes the rules for optimizing floating-point expressions                                                |
| <code>--remarks</code>                    | Enables remarks                                                                                            |
| <code>--require_prototypes</code>         | Verifies that functions are declared before they are defined                                               |

Table 34: Compiler options summary (Continued)



| Command line option                          | Description                                            |
|----------------------------------------------|--------------------------------------------------------|
| --root_variables                             | Specifies variables as __root                          |
| -s                                           | Optimizes for speed.                                   |
| --segment                                    | Changes a segment name                                 |
| --separate_cluster_for_initialized_variables | Separates initialized and non-initialized variables    |
| --silent                                     | Sets silent operation                                  |
| --spmcr_address                              | Defines the SPMCR address                              |
| --strict                                     | Checks for strict compliance with Standard C/C++       |
| --string_literals_in_flash                   | Puts "string" in the __nearflash or __farflash segment |
| --system_include_dir                         | Specifies the path for system include files            |
| --use_c++_inline                             | Uses C++ inline semantics in C99                       |
| -v                                           | Specifies the processor variant                        |
| --version1_calls                             | Uses the ICCA90 calling convention                     |
| --vla                                        | Enables C99 VLA support                                |
| --warnings_affect_exit_code                  | Warnings affect exit code                              |
| --warnings_are_errors                        | Warnings are treated as errors                         |
| -y                                           | Places constants and literals                          |
| -z                                           | Optimizes for size                                     |
| --zero_register                              | Specifies register R15 as the zero register            |

Table 34: Compiler options summary (Continued)

## Descriptions of compiler options

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



Note that if you use the options page **Extra Options** to specify specific command line options, the IDE does not perform an instant check for consistency problems like conflicting options, duplication of options, or use of irrelevant options.

# --64bit\_doubles

|             |                                                                                                             |
|-------------|-------------------------------------------------------------------------------------------------------------|
| Syntax      | --64bit_doubles                                                                                             |
| Description | Use this option to force the compiler to use 64-bit doubles instead of 32-bit doubles which is the default. |
| See also    | <i>Floating-point types</i> , page 235.                                                                     |



To set related options, choose:  
**Project>Options>General Options>Target**

# --64k\_flash

|             |                                                                                                                                                                                                                                                                                                                                                                                                      |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | --64k_flash                                                                                                                                                                                                                                                                                                                                                                                          |
| Description | <p>This option tells the compiler that the intended target processor does not have more than 64 Kbytes program flash memory (small flash), and that the AVR core therefore does not have the <code>RAMPZ</code> register or the <code>ELPM</code> instruction.</p> <p>This option can only be used together with the <code>-v2</code>, <code>-v3</code>, and <code>-v4</code> processor options.</p> |



**Project>Options>General Options>Target (No RAMPZ register)**

# --c89

|             |                                                                                                                                                              |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | --c89                                                                                                                                                        |
| Description | <p>Use this option to enable the C89 C dialect instead of Standard C.</p> <p><b>Note:</b> This option is mandatory when the MISRA C checking is enabled.</p> |
| See also    | <i>C language overview</i> , page 135.                                                                                                                       |



**Project>Options>General Options>Language>C dialect>C89**

## --char\_is\_signed

Syntax

--char\_is\_signed

Description

By default, the compiler interprets the plain `char` type as unsigned. Use this option to make the compiler interpret the plain `char` type as signed instead. This can be useful when you, for example, want to maintain compatibility with another compiler.

**Note:** The runtime library is compiled without the `--char_is_signed` option and cannot be used with code that is compiled with this option.



**Project>Options>C/C++ Compiler>Language>Plain ‘char’ is**

## --char\_is\_unsigned

Syntax

--char\_is\_unsigned

Description

Use this option to make the compiler interpret the plain `char` type as unsigned. This is the default interpretation of the plain `char` type.



**Project>Options>C/C++ Compiler>Language>Plain ‘char’ is**

## --clib

Syntax

--clib

Description

Use this option to use the system header files for the CLIB library; the compiler will automatically locate the files and use them when compiling.

**Note:** The CLIB library is used by default: To use the DLIB library, use the `--dlib` or the `--dlib_config` option instead.

See also


`--dlib`, page 201 and `--dlib_config`, page 202.




To set related options, choose:

**Project>Options>General Options>Library Configuration**

**--cpu**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                             |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------|
| Syntax      | <code>--cpu=processor</code>                                                                                                                                                                                                                                                                                                                                                                                                                            |                             |
| Parameters  | <i>processor</i>                                                                                                                                                                                                                                                                                                                                                                                                                                        | Specifies a specific device |
| Description | <p>The compiler supports different processor variants. Use this option to select a specific processor variant for which the code will be generated.</p> <p>Note that to specify the processor, you can use either the <code>--cpu</code> option or the <code>-v</code> option. The <code>--cpu</code> option is, however, more precise because it contains more information about the intended target than the more generic <code>-v</code> option.</p> |                             |
| See also    | <code>-v</code> , page 226, <i>Processor configuration</i> , page 30.                                                                                                                                                                                                                                                                                                                                                                                   |                             |
|             |  To set related options, choose:<br><b>Project&gt;Options&gt;General Options&gt;Processor configuration</b>                                                                                                                                                                                                                                                            |                             |

**--cross\_call\_passes**

|             |                                                                                                                                                                                                                                                                                                                                |                                                                        |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------|
| Syntax      | <code>--cross_call_passes=N</code>                                                                                                                                                                                                                                                                                             |                                                                        |
| Parameters  | <i>N</i>                                                                                                                                                                                                                                                                                                                       | The number of times to run the cross call optimizer, which can be 1–5. |
| Description | <p>Use this option to decrease the <code>RSTACK</code> usage by running the cross-call optimizer. The default is to run it until no more improvements can be made.</p> <p><b>Note:</b> Use this option if you have a target processor with a hardware stack or a small internal return stack segment, <code>RSTACK</code>.</p> |                                                                        |
| See also    | <code>--no_cross_call</code> , page 213                                                                                                                                                                                                                                                                                        |                                                                        |
|             |  To set related options, choose:<br><b>Project&gt;Options&gt;C/C++ Compiler&gt;Optimizations</b>                                                                                                                                            |                                                                        |

-D

|             |                                                                                                                                                      |                                      |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------|
| Syntax      | <code>-D symbol [=value]</code>                                                                                                                      |                                      |
| Parameters  | <i>symbol</i>                                                                                                                                        | The name of the preprocessor symbol  |
|             | <i>value</i>                                                                                                                                         | 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 <code>-D</code> has the same effect as a <code>#define</code> statement at the top of the source file:                                    |                                      |
|             | <code>-Dsymbol</code>                                                                                                                                |                                      |
|             | is equivalent to:                                                                                                                                    |                                      |
|             | <code>#define symbol 1</code>                                                                                                                        |                                      |
|             | To get the equivalence of:                                                                                                                           |                                      |
|             | <code>#define FOO</code>                                                                                                                             |                                      |
|             | specify the <code>=</code> sign but nothing after, for example:                                                                                      |                                      |
|             | <code>-DFOO=</code>                                                                                                                                  |                                      |



**Project>Options>C/C++ Compiler>Preprocessor>Defined symbols**

--debug, -r

|             |                                                                                                                                                                                         |  |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| Syntax      | <code>--debug</code><br><code>-r</code>                                                                                                                                                 |  |
| Description | Use the <code>--debug</code> or <code>-r</code> option to make the compiler include information in the object modules required by the IAR C-SPY® Debugger and other symbolic debuggers. |  |
|             | <b>Note:</b> Including debug information will make the object files larger than otherwise.                                                                                              |  |



**Project>Options>C/C++ Compiler>Output>Generate debug information**

**--dependencies**

|                                                                                                                                               |                                                                                        |                               |
|-----------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|-------------------------------|
| Syntax                                                                                                                                        | <code>--dependencies=[<i>i</i> <i>m</i>]</code> { <i>filename</i>   <i>directory</i> } |                               |
| Parameters                                                                                                                                    | <i>i</i> (default)                                                                     | Lists only the names of files |
|                                                                                                                                               | <i>m</i>                                                                               | Lists in makefile style       |
| For information about specifying a filename or a directory, see <i>Rules for specifying a filename or directory as parameters</i> , page 188. |                                                                                        |                               |

**Description** Use this option to make the compiler list the names of all source and header files opened for input into a file with the default filename extension *i*.

**Example** If `--dependencies` or `--dependencies=i` is used, the name of each opened input file, including the full path, if available, is output on a separate line. For example:

```
c:\iar\product\include\stdio.h
d:\myproject\include\file.h
```

If `--dependencies=m` is used, the output is in makefile style. For each input file, one line containing a makefile dependency rule is produced. Each line consists of the name of the object file, a colon, a space, and the name of an input file. For example:

```
file.r90: c:\iar\product\include\stdio.h
file.r90: d:\myproject\include\file.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:

```
%.r90 : %.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      | <code>--diag_error=tag[, tag, ...]</code>                                                                                                                                                                                                                                                |                                                                          |
| Parameters  | <i>tag</i>                                                                                                                                                                                                                                                                               | The number of a diagnostic message, for example the message number Pe117 |
| Description | Use this option to reclassify certain diagnostic messages as errors. An error indicates a violation of the C or C++ language rules, of such severity that object code will not be generated. The exit code will be non-zero. This option may be used more than once on the command line. |                                                                          |



**Project>Options>C/C++ Compiler>Diagnostics>Treat these as errors**

**--diag\_remark**

|             |                                                                                                                                                                                                                                                                                                                                                                                                             |                                                                          |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------|
| Syntax      | <code>--diag_remark=tag[, tag, ...]</code>                                                                                                                                                                                                                                                                                                                                                                  |                                                                          |
| Parameters  | <i>tag</i>                                                                                                                                                                                                                                                                                                                                                                                                  | 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.<br><br><b>Note:</b> By default, remarks are not displayed; use the <code>--remarks</code> option to display them. |                                                                          |



**Project>Options>C/C++ Compiler>Diagnostics>Treat these as remarks**

**--diag\_suppress**

|            |                                              |                                                                          |
|------------|----------------------------------------------|--------------------------------------------------------------------------|
| Syntax     | <code>--diag_suppress=tag[, tag, ...]</code> |                                                                          |
| Parameters | <i>tag</i>                                   | The number of a diagnostic message, for example the message number Pe117 |

**Description** Use this option to suppress certain diagnostic messages. These messages will not be displayed. This option may be used more than once on the command line.



**Project>Options>C/C++ Compiler>Diagnostics>Suppress these diagnostics**

**--diag\_warning**

**Syntax** `--diag_warning=tag[, tag, ...]`

**Parameters**

|            |                                                                          |
|------------|--------------------------------------------------------------------------|
| <i>tag</i> | The number of a diagnostic message, for example the message number Pe826 |
|------------|--------------------------------------------------------------------------|

**Description** Use this option to reclassify certain diagnostic messages as warnings. A warning indicates an error or omission that is of concern, but which will not cause the compiler to stop before compilation is completed. This option may be used more than once on the command line.



**Project>Options>C/C++ Compiler>Diagnostics>Treat these as warnings**

**--diagnostics\_tables**

**Syntax** `--diagnostics_tables {filename|directory}`

**Parameters** For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 188.

**Description** Use this option to list all possible diagnostic messages in a named file. This can be convenient, for example, if you have used a pragma directive to suppress or change the severity level of any diagnostic messages, but forgot to document why.


This option cannot be given together with other options.




This option is not available in the IDE.




## --disable\_direct\_mode

|             |                                                                                                                                                                          |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--disable_direct_mode</code>                                                                                                                                       |
| Description | Use this option to prevent the compiler from generating the direct addressing mode instructions <code>LDS</code> and <code>STS</code> .                                  |
|             |  To set this option, use <b>Project&gt;Options&gt;C/C++ Compiler&gt;Extra Options</b> . |

## --discard\_unused\_publics

|             |                                                                                                                                                                                                                                                                            |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--discard_unused_publics</code>                                                                                                                                                                                                                                      |
| Description | Use this option to discard unused public functions and variables when compiling with the <code>--mfc</code> compiler option.<br><br><b>Note:</b> Do not use this option only on parts of the application, as necessary symbols might be removed from the generated output. |
| See also    | <code>--mfc</code> , page 212 and <i>Multi-file compilation units</i> , page 168.                                                                                                                                                                                          |
|             |  <b>Project&gt;Options&gt;C/C++ Compiler&gt;Discard unused publics</b>                                                                                                                    |

## --dlib

|             |                                                                                                                                                                                                                                                                                  |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--dlib</code>                                                                                                                                                                                                                                                              |
| Description | Use this option to use the system header files for the DLIB library; the compiler will automatically locate the files and use them when compiling.<br><br><b>Note:</b> The DLIB library is used by default: To use the CLIB library, use the <code>--clib</code> option instead. |
| See also    | <code>--dlib_config</code> , page 202, <code>--no_system_include</code> , page 215, <code>--system_include_dir</code> , page 225, and <code>--clib</code> , page 195.                                                                                                            |
|             |  To set related options, choose:<br><b>Project&gt;Options&gt;General Options&gt;Library Configuration</b>                                                                                     |

**--dlib\_config**





|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |                                                                                                                                                                                                                                                              |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--dlib_config filename.h config</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |                                                                                                                                                                                                                                                              |
| Parameters  | <i>filename</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        | A DLIB configuration header file. For information about specifying a filename, see <i>Rules for specifying a filename or directory as parameters</i> , page 188.                                                                                             |
|             | <i>config</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          | The default configuration file for the specified configuration will be used. Choose between:<br>none, no configuration will be used<br>normal, the normal library configuration will be used (default)<br>full, the full library configuration will be used. |
| Description | <p>Each runtime library has a corresponding library configuration file. Use this option to explicitly specify which library configuration file to use, either by specifying an explicit file or by specifying a library configuration—in which case the default file for that library configuration will be used. Make sure that you specify a configuration that corresponds to the library you are using. If you do not specify this option, the default library configuration file will be used.</p> <p>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 <code>avr\lib</code>. For examples and information about prebuilt runtime libraries, see <i>Using a prebuilt library</i>, page 73.</p> <p>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 <i>Building and using a customized library</i>, page 82.</p> <p><b>Note:</b> This option only applies to the IAR DLIB runtime environment.</p> |                                                                                                                                                                                                                                                              |



To set related options, choose:  
**Project>Options>General Options>Library Configuration**

**--do\_cross\_call**

|             |                                                                                                                                                                                        |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--do_cross_call</code>                                                                                                                                                           |
| Description | Use this option to force the compiler to run the cross call optimizer, regardless of the optimization level. The cross call optimizer is otherwise only run at high size optimization. |

|                |                                                                                                                                                                                                                                                                                                                                                                               |
|----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| See also       | -O, page 217                                                                                                                                                                                                                                                                                                                                                                  |
|                |  To set related options, choose:<br><b>Project&gt;Options&gt;C/C++ Compiler&gt;Optmization</b>                                                                                                                                                                                               |
| <b>-e</b>      |                                                                                                                                                                                                                                                                                                                                                                               |
| Syntax         | -e                                                                                                                                                                                                                                                                                                                                                                            |
| Description    | <p>In the command line version of the compiler, language extensions are disabled by default. If you use language extensions such as extended keywords and anonymous structs and unions in your source code, you must use this option to enable them.</p> <p><b>Note:</b> The <code>-e</code> option and the <code>--strict</code> option cannot be used at the same time.</p> |
| See also       | <i>Enabling language extensions</i> , page 137.                                                                                                                                                                                                                                                                                                                               |
|                |  <b>Project&gt;Options&gt;C/C++ Compiler&gt;Language&gt;Standard with IAR extensions</b><br><b>Note:</b> By default, this option is selected in the IDE.                                                                                                                                     |
| <b>--ec++</b>  |                                                                                                                                                                                                                                                                                                                                                                               |
| Syntax         | --ec++                                                                                                                                                                                                                                                                                                                                                                        |
| Description    | <p>In the compiler, the default language is C. If you use Embedded C++, you must use this option to set the language the compiler uses to Embedded C++.</p>                                                                                                                                                                                                                   |
|                |  <b>Project&gt;Options&gt;C/C++ Compiler&gt;Language&gt;Embedded C++</b>                                                                                                                                                                                                                   |
| <b>--eec++</b> |                                                                                                                                                                                                                                                                                                                                                                               |
| Syntax         | --eec++                                                                                                                                                                                                                                                                                                                                                                       |
| Description    | <p>In the compiler, the default language is C. If you take advantage of Extended Embedded C++ features like namespaces or the standard template library in your source code, you must use this option to set the language the compiler uses to Extended Embedded C++.</p>                                                                                                     |
| See also       | <i>Extended Embedded C++</i> , page 144.                                                                                                                                                                                                                                                                                                                                      |
|                |  <b>Project&gt;Options&gt;C/C++ Compiler&gt;Language&gt;Extended Embedded C++</b>                                                                                                                                                                                                          |

**--eecr\_address**

|             |                                                                                                                                                                                                                                                                                                                                     |                                                     |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------|
| Syntax      | <code>--eecr_address address</code>                                                                                                                                                                                                                                                                                                 |                                                     |
| Parameters  | <i>address</i>                                                                                                                                                                                                                                                                                                                      | The value of the EECR address. The default is 0x1C. |
| Description | <p>If you use the <code>-v</code> processor option, the <code>--eecr_address</code> option can be used for modifying the value of the EECR address.</p> <p>If you use the <code>--cpu</code> processor option, the <code>--eecr_address</code> option is implicitly set, which means you should not use these options together.</p> |                                                     |
| See also    | <code>-v</code> , page 226 and <code>--cpu</code> , page 196.                                                                                                                                                                                                                                                                       |                                                     |



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

**--eeprom\_size**

|             |                                                                                                                                                                                                                           |                                           |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------|
| Syntax      | <code>--eeprom_size=N</code>                                                                                                                                                                                              |                                           |
| Parameters  | <i>N</i>                                                                                                                                                                                                                  | The size of the EEPROM in bytes, 0–65536. |
| Description | <p>Use this option to enable the <code>__eeprom</code> extended keyword by specifying the size of the built-in EEPROM.</p> <p>To use the <code>__eeprom</code> extended keyword, language extensions must be enabled.</p> |                                           |
| See also    | <i>Function storage</i> , page 47, <code>-e</code> , page 203 and <i>language</i> , page 273.                                                                                                                             |                                           |



To set related options, choose:  
**Project>Options>C/C++ Compiler>Code**

**--enable\_external\_bus**

|             |                                                                                                                                                                                                                                                                |  |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| Syntax      | <code>--enable_external_bus</code>                                                                                                                                                                                                                             |  |
| Description | <p>Use this option to make the compiler add the special <code>__require</code> statement which makes XLINK include the code in <code>startup.s90</code> that enables the external data bus. Use this option if you intend to place RSTACK in external RAM.</p> |  |

**Note:** The code in `cstartup.s90` that enables the external data bus is preferably placed in `low_level_init` instead.

See also

*The return address stack*, page 63.



**Project>Options>General Options>System**

## --enable\_multibytes

Syntax

`--enable_multibytes`

Description

By default, multibyte characters cannot be used in C or C++ source code. Use this option to make multibyte characters in the source code be interpreted according to the host computer's default setting for multibyte support.

Multibyte characters are allowed in C and C++ style comments, in string literals, and in character constants. They are transferred untouched to the generated code.



**Project>Options>C/C++ Compiler>Language>Enable multibyte support**

## --enhanced\_core

Syntax

`--enhanced_core`

Description

Use this option to allow the compiler to generate instructions from the enhanced instruction set that is available in some AVR devices, for example ATmega161.


The enhanced instruction set consists of these instructions:

MUL  
 MOVW  
 MULS  
 MULSU  
 FMUL  
 FMULS  
 FMULSU  
 LPM Rd, Z  
 LPM Rd, Z+  
 ELPM Rd, Z  
 ELPM Rd, Z+  
 SPM




**Project>Options>General Options>Enhanced core**

**--error\_limit**

|             |                                                                                                                                                                  |                                                                                                                            |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--error_limit=<i>n</i></code>                                                                                                                              |                                                                                                                            |
| Parameters  | <i>n</i>                                                                                                                                                         | The number of errors before the compiler stops the compilation. <i>n</i> must be a positive integer; 0 indicates no limit. |
| Description | Use the <code>--error_limit</code> option to specify the number of errors allowed before the compiler stops the compilation. By default, 100 errors are allowed. |                                                                                                                            |
|             |                                                                                 | This option is not available in the IDE.                                                                                   |

**-f**

|              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |                                                                                        |
|--------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| Syntax       | <code>-f <i>filename</i></code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |                                                                                        |
| Parameters   | For information about specifying a filename, see <i>Rules for specifying a filename or directory as parameters</i> , page 188.                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                        |
| Descriptions | Use this option to make the compiler read command line options from the named file, with the default filename extension <code>.xcl</code> .<br><br>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.<br><br>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 <b>Project&gt;Options&gt;C/C++ Compiler&gt;Extra Options</b> . |

**--force\_switch\_type**

|            |                                          |                                |
|------------|------------------------------------------|--------------------------------|
| Syntax     | <code>--force_switch_type={0 1 2}</code> |                                |
| Parameters | 0                                        | Library call with switch table |
|            | 1                                        | Inline code with switch table  |
|            | 2                                        | Inline compare/jump logic      |

## Description

Use this option to set the switch type.



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

**-f**

## Syntax

`-f filename`

## Parameters

For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 188.

## Descriptions

Use this option to make the compiler read command line options from the named file, with the default filename extension `.xcl`.

In the command file, you format the items exactly as if they were on the command line itself, except that you may use multiple lines, because the newline character acts just as a space or tab character.

Both C and C++ style comments are allowed in the file. Double quotes behave in the same way as in the Microsoft Windows command line environment.



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

**--guard\_calls**

## Syntax

`--guard_calls`

## Description

Use this option to enable guards for function static variable initialization. This option should be used in a threaded C++ environment.

## See also

*Managing a multithreaded environment*, page 166.



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

**--header\_context**

## Syntax

`--header_context`

## Description

Occasionally, to find the cause of a problem it is necessary to know which header file that was included from which source line. Use this option to list, for each diagnostic

message, not only the source position of the problem, but also the entire include stack at that point.



This option is not available in the IDE.

**-I**

|             |                                                                                                                                          |                                                 |  |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------|--|
| Syntax      | <code>-I path</code>                                                                                                                     |                                                 |  |
| Parameters  | <code>path</code>                                                                                                                        | The search path for <code>#include</code> files |  |
| Description | Use this option to specify the search paths for <code>#include</code> files. This option can be used more than once on the command line. |                                                 |  |
| See also    | <i>Include file search procedure</i> , page 182.                                                                                         |                                                 |  |



**Project>Options>C/C++ Compiler>Preprocessor>Additional include directories**

**--initializers\_in\_flash**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |  |  |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|--|
| Syntax      | <code>--initializers_in_flash</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |  |  |
| Description | <p>Use this option to place aggregate initializers in flash memory. These initializers are otherwise placed in the external segments <code>_C</code> or in the flash segments <code>_F</code> if the compiler option <code>-y</code> was also specified.</p> <p>An aggregate initializer—an array or a struct—is constant data that is copied to the stack dynamically at runtime, in this case every time a function is entered.</p> <p>The drawback of placing data in flash memory is that it takes more time to copy it; the advantage is that it does not occupy memory in the data space.</p> <p>Local variables with aggregate initializers are copied from the segments:</p> |  |  |

| Data            | Default             | <code>-y</code>     | <code>--initializers_in_flash</code> |
|-----------------|---------------------|---------------------|--------------------------------------|
| auto aggregates | <code>NEAR_C</code> | <code>NEAR_F</code> | <code>NEAR_F</code>                  |

Table 35: Accessing variables with aggregate initializers



Example

```
void Func ()
{
 char buf[4] = { 'l', 'd', 'g', 't' };
 ...
}
```

See also

-y, page 229 and *Initialization of local aggregates at function invocation*, page 60.



To set related options, choose:

**Project>Options>C/C++ Compiler>Code**

-1

Syntax

```
-1 [a|A|b|B|c|C|D] [N] [H] {filename|directory}
```

Parameters

|             |                                                                                                                                                                                                                                           |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| a (default) | Assembler list file                                                                                                                                                                                                                       |
| A           | Assembler list file with C or C++ source as comments                                                                                                                                                                                      |
| b           | Basic assembler list file. This file has the same contents as a list file produced with -1a, 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 -1A, except that no extra compiler generated information (runtime model attributes, call frame information, frame size information) is included * |
| c           | C or C++ list file                                                                                                                                                                                                                        |
| C (default) | C or C++ list file with assembler source as comments                                                                                                                                                                                      |
| D           | C or C++ list file with assembler source as comments, but without instruction offsets and hexadecimal byte values                                                                                                                         |
| N           | No diagnostics in file                                                                                                                                                                                                                    |
| H           | Include source lines from header files in output. Without this option, only source lines from the primary source file are included                                                                                                        |

**\* This makes the list file less useful as input to the assembler, but more useful for reading by a human.**

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 188.

**Description** Use this option to generate an assembler or C/C++ listing to a file. Note that this option can be used one or more times on the command line.



To set related options, choose:  
**Project>Options>C/C++ Compiler>List**

**--library\_module**

**Syntax** --library\_module

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



**Project>Options>C/C++ Compiler>Output>Module type>Library Module**

**--lock\_regs**

**Syntax** --lock\_regs *N*

**Parameters** *N* The number of registers to lock, 0–12.

**Description** Use this option to lock registers that are to be used for global register variables. When you use this option, the registers R15 and downwards will be locked.

To maintain module consistency, make sure to lock the same number of registers in all modules.


**Note:** Locking more than nine registers might cause linking to fail. Because the pre-built libraries delivered with the product do not lock any registers, a library function might potentially use any of the lockable registers. Any such resource conflict between locked registers and compiler-used registers will be reported at link time.

If you need to lock any registers in your code, the library must therefore be rebuilt with the same set of locked registers.




To set related options, choose:  
**Project>Options>C/C++ Compiler>Code**

## --macro\_positions\_in\_diagnostics

|                                                                                   |                                                                                                                                                              |
|-----------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax                                                                            | <code>--macro_positions_in_diagnostics</code>                                                                                                                |
| Description                                                                       | Use this option to obtain position references inside macros in diagnostic messages. This is useful for detecting incorrect source code constructs in macros. |
|  | To set this option, use <b>Project&gt;Options&gt;C/C++ Compiler&gt;Extra Options</b> .                                                                       |

## --memory\_model, -m

|             |                                                                                                                                                                                                                                                                                                                                                                                                                             |  |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| Syntax      | <code>--memory_model={tiny t small s large l huge h}</code><br><code>-m{tiny t small s large l&lt;huge h}</code>                                                                                                                                                                                                                                                                                                            |  |
| Parameters  | <div> <div>tiny, t</div> <div>Specifies the Tiny memory model</div> </div> <div> <div>small, s</div> <div>Specifies the Small memory model</div> </div> <div> <div>large, l</div> <div>Specifies the Large memory model</div> </div> <div> <div>huge, h</div> <div>Specifies the Huge memory model</div> </div>                                                                                                             |  |
| Description | <p>Use this option to specify the memory model, for which the code is to be generated.</p> <p>By default, the compiler generates code for the Tiny memory model for all processor options except <code>-v4</code> and <code>-v6</code> where the Small memory model is the default.</p> <p>Use the <code>-m</code> or the <code>--memory_model</code> option if you want to generate code for a different memory model.</p> |  |
| Example     | <p>To generate code for the Large memory model, give the command:</p> <pre>iccavr filename -ml</pre> <p>or:</p> <pre>iccavr filename --memory_model=large</pre>                                                                                                                                                                                                                                                             |  |
| See also    | <div> <div>Memory models, page 38</div> <div></div> </div> <p>To set related options, choose:</p> <p><b>Project&gt;Options&gt;General Options&gt;Memory model</b></p>                                                                                                                                                                    |  |

**--mfc**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | --mfc                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| Description | <p>Use this option to enable <i>multi-file compilation</i>. This means that the compiler compiles one or several source files specified on the command line as one unit, which enhances interprocedural optimizations.</p> <p><b>Note:</b> The compiler will generate one object file per input source code file, where the first object file contains all relevant data and the other ones are empty. If you want only the first file to be produced, use the <code>-o</code> compiler option and specify a certain output file.</p> |
| Example     | <code>iccavr myfile1.c myfile2.c myfile3.c --mfc</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| See also    | <code>--discard_unused_publics</code> , page 201, <code>--output</code> , <code>-o</code> , page 218, and <i>Multi-file compilation units</i> , page 168.                                                                                                                                                                                                                                                                                                                                                                             |



**Project>Options>C/C++ Compiler>Multi-file compilation**

**--module\_name**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                           |             |                                |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------|--------------------------------|
| Syntax      | --module_name= <i>name</i>                                                                                                                                                                                                                                                                                                                                                                                                                |             |                                |
| Parameters  | <table><tr><td><i>name</i></td><td>An explicit object module name</td></tr></table>                                                                                                                                                                                                                                                                                                                                                       | <i>name</i> | An explicit object module name |
| <i>name</i> | An explicit object module name                                                                                                                                                                                                                                                                                                                                                                                                            |             |                                |
| Description | <p>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.</p> <p>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.</p> |             |                                |



**Project>Options>C/C++ Compiler>Output>Object module name**

**--no\_clustering**

|             |                                                             |
|-------------|-------------------------------------------------------------|
| Syntax      | --no_clustering                                             |
| Description | Use this option to disable static clustering optimizations. |

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

See also

*Static clustering*, page 172.



**Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Static clustering**

## --no\_code\_motion

Syntax

--no\_code\_motion

Description

Use this option to disable code motion optimizations.

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

See also

*Code motion*, page 171.



**Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Code motion**

## --no\_cross\_call

Syntax

--no\_cross\_call

Description

Use this option to disable the cross-call optimization. This is highly recommended if your target processor has a hardware stack or a small internal return stack segment, `RSTACK`, because this option reduces the usage of `RSTACK`.

This optimization is performed at size optimization, level High. Note that, although the optimization can drastically reduce the code size, this optimization increases the execution time.

See also

--cross\_call\_passes, page 196, *Cross call*, page 172.



**Project>Options>C/C++ Compiler>Optimizations>Enabled optimizations>Cross call**

## --no\_cse

Syntax

--no\_cse

Description

Use this option to disable common subexpression elimination.

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

See also

*Common subexpression elimination*, page 170.



**Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Common subexpression elimination**

## --no\_inline

Syntax

--no\_inline

Description

Use this option to disable function inlining.

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

See also

*Function inlining*, page 170.



**Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Function inlining**

## --no\_path\_in\_file\_macros

Syntax

--no\_path\_in\_file\_macros

Description

Use this option to exclude the path from the return value of the predefined preprocessor symbols `__FILE__` and `__BASE_FILE__`.

See also

*Descriptions of predefined preprocessor symbols*, page 294.



This option is not available in the IDE.

## --no\_rampd

Syntax

--no\_rampd

Description

Use this option to make the compiler use the `RAMPZ` register instead of `RAMPD`. This option corresponds to the instructions `LDS` and `STS`.

Note that this option is only useful on processor variants with more than 64 Kbytes data (`-v4` and `-v6`).



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

## --no\_static\_destruction

|             |                                                                                                                                                                                                                             |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--no_static_destruction</code>                                                                                                                                                                                        |
| Description | <p>Normally, the compiler emits code to destroy C++ static variables that require destruction at program exit. Sometimes, such destruction is not needed.</p> <p>Use this option to suppress the emission of such code.</p> |



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

## --no\_system\_include

|             |                                                                                                                                                                                                                                                                |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--no_system_include</code>                                                                                                                                                                                                                               |
| Description | <p>By default, the compiler automatically locates the system include files. Use this option to disable the automatic search for system include files. In this case, you might need to set up the search path by using the <code>-I</code> compiler option.</p> |
| See also    | <code>--clib</code> , page 195, <code>--dlib</code> , page 201, <code>--dlib_config</code> , page 202, and <code>--system_include_dir</code> , page 225.                                                                                                       |



**Project>Options>C/C++ Compiler>Preprocessor>Ignore standard include directories**

## --no\_tbaa

|             |                                                       |
|-------------|-------------------------------------------------------|
| Syntax      | <code>--no_tbaa</code>                                |
| Description | Use this option to disable type-based alias analysis. |
| See also    | <i>Type-based alias analysis</i> , page 171.          |



**Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Type-based alias analysis**

# --no\_typedefs\_in\_diagnostics

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--no_typedefs_in_diagnostics</code>                                                                                                                                                                                                                                                                                                                                                                                                                               |
| Description | Use this option to disable the use of typedef names in diagnostics. Normally, when a type is mentioned in a message from the compiler, most commonly in a diagnostic message of some kind, the typedef names that were used in the original declaration are used whenever they make the resulting text shorter.                                                                                                                                                         |
| Example     | <pre>typedef int (*MyPtr)(char const *);<br/>MyPtr p = "foo";</pre> <p>will give an error message like this:</p> <pre>Error[Pe144]: a value of type "char *" cannot be used to<br/>initialize an entity of type "MyPtr"</pre> <p>If the <code>--no_typedefs_in_diagnostics</code> option is used, the error message will be like this:</p> <pre>Error[Pe144]: a value of type "char *" cannot be used to<br/>initialize an entity of type "int (*)(char const *)"</pre> |



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

# --no\_ubrof\_messages


|             |                                                                                                                                                                                                                                                                     |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--no_ubrof_messages</code>                                                                                                                                                                                                                                    |
| Description | Use this option to minimize the size of your application object file by excluding messages from the UBROF files. A file size decrease of up to 60% can be expected. Note that the XLINK diagnostic messages will, however, be less useful when you use this option. |




To set related options, choose:  
**Project>Options>C/C++ Compiler>Output>No error messages in output files**



--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. Corresponds to the option -s9.      |
|             | hs                                                                                                                                                                                                                                                                                                                                                                                                                                                                | High, favoring speed                                |
|             | hz                                                                                                                                                                                                                                                                                                                                                                                                                                                                | High, favoring size. Corresponds to the option -z9. |
|             | <b>*The most important difference between None and Low is that at None, all non-static variables will live during their entire scope.</b>                                                                                                                                                                                                                                                                                                                         |                                                     |
| 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.<br><br>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 168, *-O*, page 217, and *-z*, page 229.



**Project>Options>C/C++ Compiler>Optimizations**

## --omit\_types

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



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

## --only\_stdout

|             |                                                                                                                                                                                              |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | --only_stdout                                                                                                                                                                                |
| Description | Use this option to make the compiler use the standard output stream ( <code>stdout</code> ) also for messages that are normally directed to the error output stream ( <code>stderr</code> ). |



This option is not available in the IDE.


## --output, -o

|             |                                                                                                                                                                                                                                                                |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | --output {filename directory}<br>-o {filename directory}                                                                                                                                                                                                       |
| Parameters  | For information about specifying a filename or a directory, see <i>Rules for specifying a filename or directory as parameters</i> , page 188.                                                                                                                  |
| Description | By default, the object code output produced by the compiler is located in a file with the same name as the source file, but with the extension <code>.o</code> . Use this option to explicitly specify a different output filename for the object code output. |




This option is not available in the IDE.

## --predef\_macros

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--predef_macros {filename directory}</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| Parameters  | For information about specifying a filename, see <i>Rules for specifying a filename or directory as parameters</i> , page 188.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| Description | <p>Use this option to list the predefined symbols. When using this option, make sure to also use the same options as for the rest of your project.</p> <p>If a filename is specified, the compiler stores the output in that file. If a directory is specified, the compiler stores the output in that directory, in a file with the <code>predef</code> filename extension.</p> <p>Note that this option requires that you specify a source file on the command line.</p> <div>  This option is not available in the IDE. </div> |

## --preinclude

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--preinclude includefile</code>                                                                                                                                                                                                                                                                                                                                                                                                                |
| Parameters  | For information about specifying a filename, see <i>Rules for specifying a filename or directory as parameters</i> , page 188.                                                                                                                                                                                                                                                                                                                       |
| Description | <p>Use this option to make the compiler read the specified include file before it starts to read the source file. This is useful if you want to change something in the source code for the entire application, for instance if you want to define a new symbol.</p> <div>  <b>Project&gt;Options&gt;C/C++ Compiler&gt;Preprocessor&gt;Preinclude file</b> </div> |

## --preprocess

|                |                                                                                                                                                                                                                                                                                                                                                                             |                |                   |                |                 |                |                                        |
|----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------|-------------------|----------------|-----------------|----------------|----------------------------------------|
| Syntax         | <code>--preprocess [= [c] [n] [1]] {filename directory}</code>                                                                                                                                                                                                                                                                                                              |                |                   |                |                 |                |                                        |
| Parameters     | <table> <tr> <td><code>c</code></td><td>Preserve comments</td></tr> <tr> <td><code>n</code></td><td>Preprocess only</td></tr> <tr> <td><code>1</code></td><td>Generate <code>#line</code> directives</td></tr> </table> <p>For information about specifying a filename or a directory, see <i>Rules for specifying a filename or directory as parameters</i>, page 188.</p> | <code>c</code> | Preserve comments | <code>n</code> | Preprocess only | <code>1</code> | Generate <code>#line</code> directives |
| <code>c</code> | Preserve comments                                                                                                                                                                                                                                                                                                                                                           |                |                   |                |                 |                |                                        |
| <code>n</code> | Preprocess only                                                                                                                                                                                                                                                                                                                                                             |                |                   |                |                 |                |                                        |
| <code>1</code> | Generate <code>#line</code> directives                                                                                                                                                                                                                                                                                                                                      |                |                   |                |                 |                |                                        |

Description

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



**Project>Options>C/C++ Compiler>Preprocessor>Preprocessor output to file**

## --public\_equ

Syntax

`--public_equ symbol[=value]`

Parameters

|               |                                                   |
|---------------|---------------------------------------------------|
| <i>symbol</i> | The name of the assembler symbol to be defined    |
| <i>value</i>  | An optional value of the defined assembler symbol |

Description

This option is equivalent to defining a label in assembler language using the `EQU` directive and exporting it using the `PUBLIC` directive. This option can be used more than once on the command line.



This option is not available in the IDE.

## --relaxed\_fp

Syntax

`--relaxed_fp`

Description

Use this option to allow the compiler to relax the language rules and perform more aggressive optimization of floating-point expressions. This option improves performance for floating-point expressions that fulfill these conditions:

- The expression consists of both single- and double-precision values
- The double-precision values can be converted to single precision without loss of accuracy
- The result of the expression is converted to single precision.

Note that performing the calculation in single precision instead of double precision might cause a loss of accuracy.

Example

```
float F(float a, float b)
{
 return a + b * 3.0;
}
```

The C standard states that `3.0` in this example has the type `double` and therefore the whole expression should be evaluated in `double` precision. However, when the

`--relaxed_fp` option is used, 3.0 will be converted to `float` and the whole expression can be evaluated in `float` precision.



**Project>Options>General Options>Language>Relaxed floating-point precision**

## --remarks

Syntax

`--remarks`

Description

The least severe diagnostic messages are called remarks. A remark indicates a source code construct that may cause strange behavior in the generated code. By default, the compiler does not generate remarks. Use this option to make the compiler generate remarks.

See also

*Severity levels*, page 185.



**Project>Options>C/C++ Compiler>Diagnostics>Enable remarks**

## --require\_prototypes

Syntax

`--require_prototypes`

Description

Use this option to force the compiler to verify that all functions have proper prototypes. Using this option means that code containing any of the following will generate an error:

- A function call of a function with no declaration, or with a Kernighan & Ritchie C declaration
- A function definition of a public function with no previous prototype declaration
- An indirect function call through a function pointer with a type that does not include a prototype.



**Project>Options>C/C++ Compiler>Language>Require prototypes**

## --root\_variables

Syntax

`--root_variables`

Description

Use this option to apply the `__root` extended keyword to all global and static variables. This will make sure that the variables are not removed by the IAR XLINK Linker.

**Note:** The `--root_variables` option is always available, even if you do not specify the compiler option `-e`, language extensions.



To set related options, choose:

**Project>Options>C/C++ Compiler>Code>Force generation of all global and static variables**

**-S**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                               |                             |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------|
| Syntax      | <code>-s [2   3   6   9]</code>                                                                                                                                                                                                                                                                                                                                                                                                                               |                             |
| Parameters  | 2                                                                                                                                                                                                                                                                                                                                                                                                                                                             | None* (Best debug support)  |
|             | 3 (default)                                                                                                                                                                                                                                                                                                                                                                                                                                                   | Low*                        |
|             | 6                                                                                                                                                                                                                                                                                                                                                                                                                                                             | Medium                      |
|             | 9                                                                                                                                                                                                                                                                                                                                                                                                                                                             | High (Maximum optimization) |
|             | <b>*The most important difference between <code>-s2</code> and <code>-s3</code> is that at level 2, all non-static variables will live during their entire scope.</b>                                                                                                                                                                                                                                                                                         |                             |
| Description | Use this option to make the compiler optimize the code for maximum execution speed. If no optimization option is specified, the optimization level 3 is used by default.<br><br>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.<br><br><b>Note:</b> The <code>-s</code> and <code>-z</code> options cannot be used at the same time. |                             |
| See also    | <code>-O</code> , page 217                                                                                                                                                                                                                                                                                                                                                                                                                                    |                             |



**Project>Options>C/C++ Compiler>Optimizations>Speed**

**--segment**

|             |                                                                                                                                                                                        |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--segment __memory_attribute=NEWSEGMENTNAME</code>                                                                                                                               |
| Description | The compiler places functions and data objects into named segments which are referred to by the IAR XLINK Linker. Use the <code>--segment</code> option to place all functions or data |

objects declared with the `__memory_attribute` in segments with names that begin with `NEWSEGMENTNAME`.

This is useful if you want to place your code or data in different address ranges and you find the `@` notation, alternatively the `#pragma location` directive, insufficient. Note that any changes to the segment names require corresponding modifications in the linker configuration file.

|          |                                                                                                                                                                                     |
|----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Example1 | This command places the <code>__near int a;</code> defined variable in the <code>MYDATA_Z</code> segment:<br><br><code>--segment __near=MYDATA</code>                               |
| Example2 | This command names the segment that contains interrupt vectors to <code>MYINTS</code> instead of the default name <code>INTVEC</code> :<br><br><code>--segment intvec=MYINTS</code> |
| See also | <i>Controlling data and function placement in memory</i> , page 164 and <i>Summary of extended keywords</i> , page 249.                                                             |



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

## **--separate\_cluster\_for\_initialized\_variables**

|             |                                                                                                                                                                                             |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--separate_cluster_for_initialized_variables</code>                                                                                                                                   |
| Description | Use this option to separate initialized and non-initialized variables when using variable clustering. The option makes the <code>*_ID</code> segments smaller but can generate larger code. |



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


## **--silent**

|             |                                                                                                                                                                                                                                                                                                     |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--silent</code>                                                                                                                                                                                                                                                                               |
| 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).<br><br>This option does not affect the display of error and warning messages. |




This option is not available in the IDE.


### **--spmcr\_address**

|             |                                                                                   |                                                                                        |
|-------------|-----------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| Syntax      | <code>--spmcr_address address</code>                                              |                                                                                        |
| Parameters  | <i>address</i>                                                                    | The SPMCR address, where 0x37 is the default.                                          |
| Description | Use this option to set the SPMCR address.                                         |                                                                                        |
|             |  | To set this option, use <b>Project&gt;Options&gt;C/C++ Compiler&gt;Extra Options</b> . |

### **--strict**

|             |                                                                                                                                                                                              |                                                                                           |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------|
| Syntax      | <code>--strict</code>                                                                                                                                                                        |                                                                                           |
| Description | By default, the compiler accepts a relaxed superset of Standard C and C++. Use this option to ensure that the source code of your application instead conforms to strict Standard C and C++. |                                                                                           |
|             | <b>Note:</b> The <code>-e</code> option and the <code>--strict</code> option cannot be used at the same time.                                                                                |                                                                                           |
| See also    | <i>Enabling language extensions</i> , page 137.                                                                                                                                              |                                                                                           |
|             |                                                                                                             | <b>Project&gt;Options&gt;C/C++ Compiler&gt;Language&gt;Language conformance&gt;Strict</b> |

### **--string\_literals\_in\_flash**

|             |                                                                                                                                                                                                                                                                                                                                                             |                                                                                        |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| Syntax      | <code>--string_literals_in_flash</code>                                                                                                                                                                                                                                                                                                                     |                                                                                        |
| Description | Use this option to put "string" in the <code>__nearflash</code> or <code>__farflash</code> segment depending on the processor option.<br><br>When this option is used, library functions taking a string literal as a parameter will no longer be type-compatible. Use the <code>*_P</code> library function variants (for example <code>printf_P</code> ). |                                                                                        |
| See also    | <i>AVR-specific library functions</i> , page 310                                                                                                                                                                                                                                                                                                            |                                                                                        |
|             |                                                                                                                                                                                                                                                                          | To set this option, use <b>Project&gt;Options&gt;C/C++ Compiler&gt;Extra Options</b> . |



## --system\_include\_dir

|             |                                                                                                                                                                                                                                                             |                                                                                                                                                                  |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>--system_include_dir path</code>                                                                                                                                                                                                                      |                                                                                                                                                                  |
| Parameters  | <i>path</i>                                                                                                                                                                                                                                                 | The path to the system include files. For information about specifying a path, see <i>Rules for specifying a filename or directory as parameters</i> , page 188. |
| Description | By default, the compiler automatically locates the system include files. Use this option to explicitly specify a different path to the system include files. This might be useful if you have not installed IAR Embedded Workbench in the default location. |                                                                                                                                                                  |
| See also    | <code>--clib</code> , page 195, <code>--dlib</code> , page 201, <code>--dlib_config</code> , page 202, and <code>--no_system_include</code> , page 215.                                                                                                     |                                                                                                                                                                  |



This option is not available in the IDE.

## --use\_c++\_inline

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |  |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| Syntax      | <code>--use_c++_inline</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |  |
| Description | <p>Standard C uses slightly different semantics for the <code>inline</code> keyword than C++ does. Use this option to get C++ semantics when compiling a Standard C source code file.</p> <p>The main difference in semantics is that in Standard C you cannot in general simply supply an inline definition in a header file. You need to supply an external definition in one of the compilation units, possibly by designating the inline definition as being external in that compilation unit.</p> |  |



**Project>Options>C/C++ Compiler>Language>C dialect>C99>C++ inline semantics**

**-v**

Syntax `-v{0|1|2|3|4|5|6}`

Parameters

| Parameter   | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0 (default) | The code space is physically limited to 8 Kbytes, and the <code>RCALL/RJMP</code> instructions are used for reaching the code space. Interrupt vectors are 2 bytes long. The compiler assumes that the index registers <code>X</code> , <code>Y</code> , and <code>Z</code> are eight bits wide when accessing the built-in SRAM. It also assumes that it is not possible to attach any external memory to the microcontroller and that it therefore should not generate any constant segment in data space ( <code>_C</code> segment). Instead the compiler adds an implicit <code>-y</code> command line option. It will also try to place all aggregate initializers in flash memory, that is, the implicit <code>--initializers_in_flash</code> option is also added to the command line. Relative function calls are made.                                                                                                                                                          |
| 1           | The code space is physically limited to 8 Kbytes, and the <code>RCALL/RJMP</code> instructions are used for reaching the code space. Interrupt vectors are 2 bytes long. The compiler assumes that it is possible to have external memory and will therefore generate <code>_C</code> segments. Relative function calls are made.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| 2           | The code space is physically limited to 128 Kbytes, and, if possible, the <code>RCALL/RJMP</code> instructions are used for reaching the code space. If that is not possible, <code>CALL/JMP</code> is used. Function calls through function pointers use <code>ICALL/IJMP</code> . 2 bytes are used for all function pointers. Interrupt vectors are 4 bytes long. The compiler assumes that the index registers <code>X</code> , <code>Y</code> , and <code>Z</code> are eight bits wide when accessing the built-in SRAM. It also assumes that it is not possible to attach any external memory to the microcontroller and that it should therefore not generate any constant segment in data space ( <code>_C</code> segment). Instead the compiler adds an implicit <code>-y</code> command line option. It will also try to place all aggregate initializers in flash memory, that is, the implicit <code>--initializers_in_flash</code> option is also added to the command line. |
| 3           | The code space is physically limited to 128 Kbytes, and, if possible, the <code>RCALL/RJMP</code> instructions are used for reaching the code space. If that is not possible, <code>CALL/JMP</code> is used. Function calls through function pointers use <code>ICALL/IJMP</code> . 2 bytes are used for all function pointers. Interrupt vectors are 4 bytes long. The compiler assumes that it is possible to have external memory and will therefore generate <code>_C</code> segments.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |

| Parameter | Description                                                                                                                                                                                                                                                                                                                                                                                                                           |
|-----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 4         | The code space is physically limited to 128 Kbytes, and, if possible, the RCALL/RJMP instructions are used for reaching the code space. If that is not possible, CALL/JMP is used. Function calls through function pointers use ICALL/IJMP. 2 bytes are used for all function pointers. Interrupt vectors are 4 bytes long. The compiler assumes that it is possible to have external memory and will therefore generate _C segments. |
| 5         | Allows function calls through farfunc function pointers to cover the entire 8 Mbyte code space by using EICALL/EIJMP. 3 bytes are used for function pointers. Interrupt vectors are 4 bytes long. The compiler assumes that it is possible to have external memory and will therefore generate _C segments.                                                                                                                           |
| 6         | Allows function calls through farfunc function pointers to cover the entire 8 Mbyte code space by using EICALL/EIJMP. 3 bytes are used for function pointers. Interrupt vectors are 4 bytes long. The compiler assumes that it is possible to have external memory and will therefore generate _C segments.                                                                                                                           |

Description Use this option to select the processor device for which the code is to be generated.

See also `--cpu`, page 196 and *Processor configuration*, page 30.



To set related options, choose:  
**Project>Options>General Options>Processor configuration**

**--version1\_calls**

Syntax `--version1_calls`

Description Use this option to make all functions and function calls use the calling convention of the A90 IAR Compiler, ICCA90, which is described in *Calling convention*, page 123.



To change the calling convention of a single function, use the `__version_1` function type attribute. See `__version_1`, page 263 for detailed information.

See also *Function storage*, page 47




**Project>Options>C/C++ Compiler>Code>Use ICCA90 1.x calling convention**


**--vla**

|             |                                                                                                                                                                                                                                                                                                                                                                           |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | --vla                                                                                                                                                                                                                                                                                                                                                                     |
| Description | Use this option to enable support for C99 variable length arrays. This option requires Standard C and cannot be used together with the --c89 compiler option.<br> <b>Note:</b> --vla should not be used together with the longjmp library function, as that can lead to memory leakages. |
| See also    | C language overview, page 135.<br> <b>Project&gt;Options&gt;C/C++ Compiler&gt;Language&gt;C dialect&gt;Allow VLA</b>                                                                                                                                                                     |

**--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.<br> 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.<br><br><b>Note:</b> 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 200.<br> <b>Project&gt;Options&gt;C/C++ Compiler&gt;Diagnostics&gt;Treat all warnings as errors</b>                                                                                                                                                                                                                            |

-y


|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | -y                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| Description | <p>Use this option to override the default placement of constants and literals.</p> <p><i>Without</i> this option, constants and literals are placed in an external <code>const</code> segment, <code>segmentbasename_C</code>. <i>With</i> this option, constants and literals will instead be placed in the initialized <code>segmentbasename_I</code> data segments that are copied from the <code>segmentbasename_ID</code> segments by the system startup code.</p> <p>Note that <code>-y</code> is implicit in the tiny memory model.</p> <p>This option can be combined with the option <code>--initializers_in_flash</code>.</p> |
| See also    | <code>--initializers_in_flash</code> , page 208                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |



To set related options, choose:  
**Project>Options>C/C++ Compiler>Code**

-z

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                           |  |   |                            |             |      |   |        |   |                             |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|---|----------------------------|-------------|------|---|--------|---|-----------------------------|
| Syntax      | <code>-z [2   3   6   9]</code>                                                                                                                                                                                                                                                                                                                                                                                                                           |  |   |                            |             |      |   |        |   |                             |
| Parameters  | <table><tr><td>2</td><td>None* (Best debug support)</td></tr><tr><td>3 (default)</td><td>Low*</td></tr><tr><td>6</td><td>Medium</td></tr><tr><td>9</td><td>High (Maximum optimization)</td></tr></table> <p><b>*The most important difference between <code>-z2</code> and <code>-z3</code> is that at level 2, all non-static variables will live during their entire scope.</b></p>                                                                     |  | 2 | None* (Best debug support) | 3 (default) | Low* | 6 | Medium | 9 | High (Maximum optimization) |
| 2           | None* (Best debug support)                                                                                                                                                                                                                                                                                                                                                                                                                                |  |   |                            |             |      |   |        |   |                             |
| 3 (default) | Low*                                                                                                                                                                                                                                                                                                                                                                                                                                                      |  |   |                            |             |      |   |        |   |                             |
| 6           | Medium                                                                                                                                                                                                                                                                                                                                                                                                                                                    |  |   |                            |             |      |   |        |   |                             |
| 9           | High (Maximum optimization)                                                                                                                                                                                                                                                                                                                                                                                                                               |  |   |                            |             |      |   |        |   |                             |
| Description | <p>Use this option to make the compiler optimize the code for minimum size. If no optimization option is specified, the optimization level 3 is used by default.</p> <p>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.</p> <p><b>Note:</b> The <code>-s</code> and <code>-z</code> options cannot be used at the same time.</p> |  |   |                            |             |      |   |        |   |                             |
| See also    | <code>-O</code> , page 217                                                                                                                                                                                                                                                                                                                                                                                                                                |  |   |                            |             |      |   |        |   |                             |




Project>Options>C/C++ Compiler>Optimizations>Size



**Project>Options>C/C++ Compiler>Optimizations>Size**

**--zero\_register**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | --zero_register                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| Description | <p>Use this option to make the compiler use register R15 as zero register, which means that register R15 is assumed to always contain zero.</p> <p>This option can in some cases reduce the size of the generated code, especially in the Large memory model. The option might be incompatible with the supplied runtime libraries. The linker will issue a link-time error if any incompatibilities arise.</p> <div> To set this option, use <b>Project&gt;Options&gt;C/C++ Compiler&gt;Extra Options</b>.</div> |

# 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. To decrease the alignment requirements on the structure and its members, use `#pragma pack`.

All data types must have a size that is a multiple of their alignment. Otherwise, only the first element of an array would be guaranteed to be placed in accordance with the alignment requirements. This means that the compiler might add pad bytes at the end of the structure.

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

### ALIGNMENT ON THE AVR MICROCONTROLLER

The AVR microcontroller does not have any alignment restrictions.

# Byte order

The AVR microcontroller stores data in little-endian byte order.

In the little-endian byte order, the *least* significant byte is stored at the lowest address in memory. The *most* significant byte is stored at the highest address.

In the big-endian byte order, the *most* significant byte is stored at the lowest address in memory. The *least* significant byte is stored at the highest address. It might be necessary to use the `#pragma bitfields=reversed` directive to be compatible with code for other compilers and I/O register definitions of some devices, see *Bitfields*, page 233.

# Basic data types

The compiler supports both all Standard C basic data types and some additional types.

## INTEGER TYPES

This table gives the size and range of each integer data type:

| Data type          | Size    | Range                                  | Alignment |
|--------------------|---------|----------------------------------------|-----------|
| bool               | 8 bits  | 0 to 1                                 | 1         |
| char               | 8 bits  | 0 to 255                               | 1         |
| signed char        | 8 bits  | -128 to 127                            | 1         |
| unsigned char      | 8 bits  | 0 to 255                               | 1         |
| signed short       | 16 bits | -32768 to 32767                        | 1         |
| unsigned short     | 16 bits | 0 to 65535                             | 1         |
| signed int         | 16 bits | -32768 to 32767                        | 1         |
| unsigned int       | 16 bits | 0 to 65535                             | 1         |
| signed long        | 32 bits | -2 <sup>31</sup> to 2 <sup>31</sup> -1 | 1         |
| unsigned long      | 32 bits | 0 to 2 <sup>32</sup> -1                | 1         |
| signed long long   | 64 bits | -2 <sup>63</sup> to 2 <sup>63</sup> -1 | 1         |
| unsigned long long | 64 bits | 0 to 2 <sup>64</sup> -1                | 1         |

Table 36: Integer types

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

## Bool

The `bool` data type is supported by default in the C++ language. If you have enabled language extensions, the `bool` type can also be used in C source code if you include the file `stdbool.h`. This will also enable the boolean values `false` and `true`.



## The long long type

The `long long` data type is supported with one restriction: The CLIB runtime library does not support the `long long` type

## The enum type

The compiler will use the smallest type required to hold `enum` constants, preferring signed rather than unsigned.

When IAR Systems language extensions are enabled, and in C++, the `enum` constants and types can also be of the type `long`, `unsigned long`, `long long`, or `unsigned long long`.

To make the compiler use a larger type than it would automatically use, define an `enum` constant with a large enough value. For example:

```
/* Disables usage of the char type for enum */
enum Cards{Spade1, Spade2,
 DontUseChar=257};
```

## The char type

The `char` type is by default unsigned in the compiler, but the `--char_is_signed` compiler option allows you to make it signed. Note, however, that the library is compiled with the `char` type as unsigned.

## The wchar\_t type

The `wchar_t` data type is an integer type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locals.

The `wchar_t` data type is supported by default in the C++ language. To use the `wchar_t` type also in C source code, you must include the file `stddef.h` from the runtime library.

**Note:** The IAR CLIB Library has only rudimentary support for `wchar_t`.

## Bitfields

In Standard C, `int`, `signed int`, and `unsigned int` can be used as the base type for integer bitfields. In standard C++, and in C when language extensions are enabled in the compiler, any integer or enumeration type can be used as the base type. It is implementation defined whether a plain integer type (`char`, `short`, `int`, etc) results in a signed or unsigned bitfield.

In the IAR C/C++ Compiler for AVR, plain integer types are treated as signed.

Bitfields in expressions are treated as `int` if `int` can represent all values of the bitfield. Otherwise, they are treated as the bitfield base type.

Each bitfield is placed into a container of its base type from the least significant bit to the most significant bit. If the last container is of the same type and has enough bits available, the bitfield is placed into this container, otherwise a new container is allocated.

If you use the directive `#pragma bitfield=reversed`, bitfields are placed from the most significant bit to the least significant bit in each container. See *bitfields*, page 267.

### Example

Assume this example:

```
struct BitfieldExample
{
 uint32_t a : 12;
 uint16_t b : 3;
 uint16_t c : 7;
 uint8_t d;
};
```

To place the first bitfield, `a`, the compiler allocates a 32-bit container at offset 0 and puts `a` into the least significant 12 bits of the container.

To place the second bitfield, `b`, a new container is allocated at offset 4, because the type of the bitfield is not the same as that of the previous one. `b` is placed into the least significant three bits of this container.

The third bitfield, `c`, has the same type as `b` and fits into the same container.

The fourth member, `d`, is allocated into the byte at offset 6. `d` cannot be placed into the same container as `b` and `c` because it is not a bitfield, it is not of the same type, and it would not fit.

When using reverse order, each bitfield is instead placed starting from the most significant bit of its container.

This is the layout of `bitfield_example`:

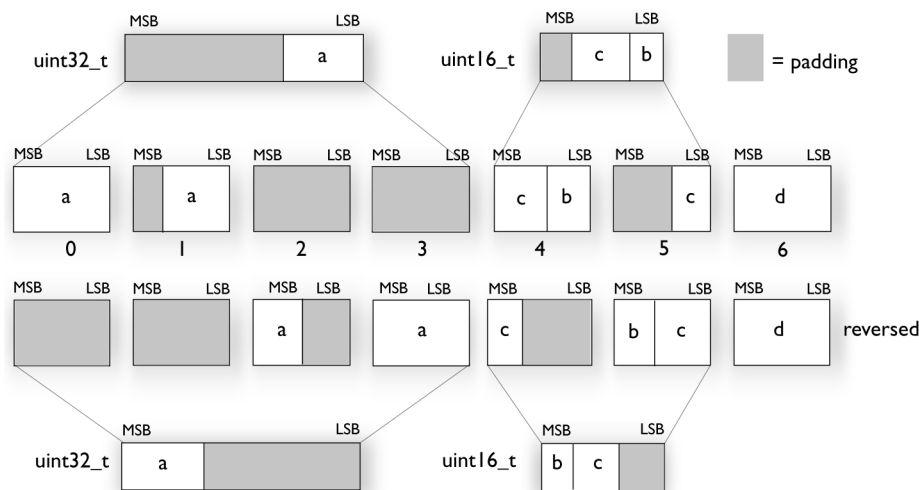


Figure 5: Layout of `bitfield_example` for disjoint types

## FLOATING-POINT TYPES

In the IAR C/C++ Compiler for AVR, floating-point values are represented in standard IEEE 754 format. The sizes for the different floating-point types are:

| Type        | Size if <code>--double=32</code> | Size if <code>--double=64</code> |
|-------------|----------------------------------|----------------------------------|
| float       | 32 bits                          | 32 bits                          |
| double      | 32 bits (default)                | 64 bits                          |
| long double | 32 bits                          | 64 bits                          |

Table 37: Floating-point types

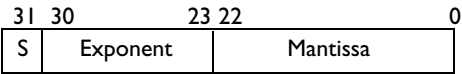
**Note:** The size of `double` and `long double` depends on the `--64bit_doubles` option, see `--64bit_doubles`, page 194. The type `long double` uses the same precision as `double`.

## Floating-point environment

Exception flags are not supported. The `feraiseexcept` function does not raise any exceptions.

32-bit floating-point format

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



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

The value of the number is:

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

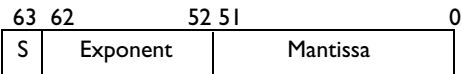
The range of the number is at least:

$\pm 1.18E-38$  to  $\pm 3.39E+38$

The precision of the float operators (+, -, \*, and /) is approximately 7 decimal digits.

64-bit floating-point format

The representation of a 64-bit floating-point number as an integer is:



The exponent is 11 bits, and the mantissa is 52 bits.

The value of the number is:

$(-1)^S * 2^{(Exponent-1023)} * 1.Mantissa$

The range of the number is at least:

$\pm 2.23E-308$  to  $\pm 1.79E+308$

The precision of the float operators (+, -, \*, and /) is approximately 15 decimal digits.

Representation of special floating-point numbers

This list describes the representation of special floating-point numbers:

- Zero is represented by zero mantissa and exponent. The sign bit signifies positive or negative zero.
- Infinity is represented by setting the exponent to the highest value and the mantissa to zero. The sign bit signifies positive or negative infinity.

- Not a number (NaN) is represented by setting the exponent to the highest positive value and the mantissa to a non-zero value. The value of the sign bit is ignored.
- Subnormal numbers are used for representing values smaller than what can be represented by normal values. The drawback is that the precision will decrease with smaller values. The exponent is set to 0 to signify that the number is denormalized, even though the number is treated as if the exponent was 1. Unlike normal numbers, denormalized numbers do not have an implicit 1 as the most significant bit (the MSB) of the mantissa. The value of a denormalized number is:

$$(-1)^S * 2^{(1-BIAS)} * 0.Mantissa$$

where BIAS is 127 and 1023 for 32-bit and 64-bit floating-point values, respectively.

**Note:** The IAR CLIB Library does not fully support the special cases of floating-point numbers, such as infinity, NaN, and subnormal numbers. A library function which gets one of these special cases of floating-point numbers as an argument might behave unexpectedly.

## Pointer types

The compiler has two basic types of pointers: function pointers and data pointers.

### FUNCTION POINTERS

The size of function pointers is always 16 or 24 bits, and they can address the entire memory. The internal representation of a function pointer is the actual address it refers to divided by two.

These function pointers are available:

| Keyword                 | Address range | Pointer size | Description                                                                                            |
|-------------------------|---------------|--------------|--------------------------------------------------------------------------------------------------------|
| <code>__nearfunc</code> | 0–0x1FFFE     | 2 bytes      | Can be called from any part of the code memory, but must reside in the first 128 Kbytes of that space. |
| <code>__farfunc</code>  | 0–0x7FFFFE    | 3 bytes      | No restrictions on code placement.                                                                     |

Table 38: Function pointers

### DATA POINTERS

Data pointers have three sizes: 8, 16, or 24 bits. These data pointers are available:

| Keyword             | Pointer size | Memory space | Index type  | Address range |
|---------------------|--------------|--------------|-------------|---------------|
| <code>__tiny</code> | 1 byte       | Data         | signed char | 0x0–0xFF      |

Table 39: Data pointers

| Keyword                  | Pointer size       | Memory space | Index type                | Address range                                                                                                                                                                                                     |
|--------------------------|--------------------|--------------|---------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>__near</code>      | 2 bytes            | Data         | signed int                | 0x0-0xFFFF                                                                                                                                                                                                        |
| <code>__far</code>       | 3 bytes            | Data         | signed int                | 0x0-0xFFFFFFFF<br>(16-bit arithmetics)                                                                                                                                                                            |
| <code>__huge</code>      | 3 bytes            | Data         | signed long               | 0x0-0xFFFFFFFF                                                                                                                                                                                                    |
| <code>__tinyflash</code> | 1 byte             | Code         | signed char               | 0x0-0xFF                                                                                                                                                                                                          |
| <code>__flash</code>     | 2 bytes            | Code         | signed int                | 0x0-0xFFFF                                                                                                                                                                                                        |
| <code>__farflash</code>  | 3 bytes            | Code         | signed int                | 0x0-0xFFFFFFFF<br>(16-bit arithmetics)                                                                                                                                                                            |
| <code>__hugeflash</code> | 3 bytes            | Code         | signed long               | 0x0-0xFFFFFFFF                                                                                                                                                                                                    |
| <code>__eeprom</code>    | 1 byte             | EEPROM       | signed char               | 0x0-0xFF                                                                                                                                                                                                          |
| <code>__eeprom</code>    | 2 bytes            | EEPROM       | signed int                | 0x0-0xFFFF                                                                                                                                                                                                        |
| <code>__generic</code>   | 2 bytes<br>3 bytes | Data/Code    | signed int<br>signed long | The most significant bit (MSB) determines whether <code>__generic</code> points to CODE (1) or DATA (0). The small generic pointer is generated for the processor options <code>-v0</code> and <code>-v1</code> . |

Table 39: Data pointers (Continued)

CASTING

Casts between pointers have these characteristics:

- Casting a *value* of an integer type to a pointer of a smaller type is performed by truncation
- Casting a *value* of an integer type to a pointer of a larger type is performed by zero extension
- Casting a *pointer type* to a smaller integer type is performed by truncation
- Casting a *pointer type* to a larger integer type is performed via casting to the largest possible pointer that fits in the integer
- Casting a *data pointer* to a function pointer and vice versa is illegal
- Casting a *function pointer* to an integer type gives an undefined result
- Casting from a smaller pointer to a larger pointer is performed by zero extension
- Casting from a larger pointer to a smaller pointer gives an undefined result.

size\_t

size\_t is the unsigned integer type required to hold the maximum size of an object. This table shows the typedef of size\_t depending on the processor option:

| Generic processor option    | Typedef       |
|-----------------------------|---------------|
| -v0 and -v1                 | unsigned int  |
| -v2, -v3, -v4, -v5, and -v6 | unsigned long |

Table 40: size\_t typedef

When using the Large or Huge memory model, the typedef for size\_t is unsigned long int.

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. This table shows the typedef of ptrdiff\_t depending on the processor option:

| Generic processor option    | Typedef     |
|-----------------------------|-------------|
| -v0 and -v1                 | signed int  |
| -v2, -v3, -v4, -v5, and -v6 | signed long |

Table 41: ptrdiff\_t typedef

**Note:** Subtracting the start address of an object from the end address can yield a negative value, because the object can be larger than what the ptrdiff\_t can represent. See this example:

```
char buff[60000]; /* Assuming ptrdiff_t is a 16-bit */
char *p1 = buff; /* signed integer type. */
char *p2 = buff + 60000;
ptrdiff_t diff = p2 - p1;
```

intptr\_t

intptr\_t is a signed integer type large enough to contain a void \*. In the IAR C/C++ Compiler for AVR, the size of intptr\_t is long int when using the Large or Huge memory model and int in all other cases.

uintptr\_t

uintptr\_t is equivalent to intptr\_t, with the exception that it is unsigned.

---

# Structure types

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

## ALIGNMENT

The `struct` and `union` types inherit the alignment requirements of their members. In addition, the size of a `struct` is adjusted to allow arrays of aligned structure objects.

## GENERAL LAYOUT

Members of a `struct` are always allocated in the order specified in the declaration. Each member is placed in the `struct` according to the specified alignment (offsets).

### Example

```
struct First
{
 char c;
 short s;
} s;
```

This diagram shows the layout in memory:

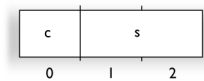


Figure 6: Structure layout

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

---

# Type qualifiers

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

## DECLARING OBJECTS VOLATILE

By declaring an object `volatile`, the compiler is informed that the value of the object can change beyond the compiler's control. The compiler must also assume that any



accesses can have side effects—thus all accesses to the `volatile` object must be preserved.

There are three main reasons for declaring an object `volatile`:

- Shared access; the object is shared between several tasks in a multitasking environment
- Trigger access; as for a memory-mapped SFR where the fact that an access occurs has an effect
- Modified access; where the contents of the object can change in ways not known to the compiler.

### Definition of access to volatile objects

The C standard defines an abstract machine, which governs the behavior of accesses to `volatile` declared objects. In general and in accordance to the abstract machine:

- The compiler considers each read and write access to an object declared `volatile` as an access
- The unit for the access is either the entire object or, for accesses to an element in a composite object—such as an array, struct, class, or union—the element. For example:

```
char volatile a;
a = 5; /* A write access */
a += 6; /* First a read then a write access */
```

- An access to a bitfield is treated as an access to the underlying type
- Adding a `const` qualifier to a `volatile` object will make write accesses to the object impossible. However, the object will be placed in RAM as specified by the C standard.

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

### Rules for accesses

In the IAR C/C++ Compiler for AVR, accesses to `volatile` declared objects are subject to these rules:

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

The compiler adheres to these rules for all 8-bit types.

These object types are treated in a special way:

| Type of object                             | Treated as                                                                                                                                                                                   |
|--------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Global register variables                  | Treated as non-triggering volatiles                                                                                                                                                          |
| __io declared variables located in 0x-0x1F | An assignment to a bitfield always generates a write access, in some cases also a read access. If only one bit is updated—set or cleared—the <code>sci/cbi</code> instructions are executed. |

Table 42: Type of volatile accesses treated in a special way

For all combinations of object types not listed, only the rule that states that all accesses are preserved applies.

DECLARING OBJECTS VOLATILE AND CONST

If you declare a `volatile` object `const`, it will be write-protected but it will still be stored in RAM memory as the C standard specifies.

To store the object in read-only memory instead, but still make it possible to access it as a `const volatile` object, follow this example:

```
/* Header */
extern int const xVar;
#define x (*(int const volatile *) &xVar)

/* Source that uses x */
int DoSomething()
{
 return x;
}

/* Source that defines x */
#pragma segment = "FLASH"
int const xVar @ "FLASH" = 6;
```

The segment `FLASH` contains the initializers. They must be flashed manually when the application starts up.

Thereafter, the initializers can be reflashed with other values at any time.

DECLARING OBJECTS CONST

The `const` type qualifier is used for indicating that a data object, accessed directly or via a pointer, is non-writable. A pointer to `const` declared data can point to both constant and non-constant objects. It is good programming practice to use `const`

declared pointers whenever possible because this improves the compiler's possibilities to optimize the generated code and reduces the risk of application failure due to erroneously modified data.

Static and global objects declared `const` are allocated in ROM. If no ROM is available in the `DATA` memory space, use the compiler option `-y` to control the placement of constants. For more information, see `-y`, page 229.

In C++, objects that require runtime initialization cannot be placed in ROM.

---

## Data types in C++

In C++, all plain C data types are represented in the same way as described earlier in this chapter. However, if any Embedded C++ features are used for a type, no assumptions can be made concerning the data representation. This means, for example, that it is not supported to write assembler code that accesses class members.



# Extended keywords

This chapter describes the extended keywords that support specific features of the AVR 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 AVR microcontroller. There are two types of attributes—*type attributes* and *object attributes*:

- Type attributes affect the *external functionality* of the data object or function
- Object attributes affect the *internal functionality* of the data object or function.

The syntax for the keywords differs slightly depending on whether it is a type attribute or an object attribute, and whether it is applied to a data object or a function.

For information about how to use attributes to modify data, see the chapter *Data storage*. For information about how to use attributes to modify functions, see the chapter *Functions*. For more information about each attribute, see *Descriptions of extended keywords*, page 250.

**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 203 for more information.

### TYPE ATTRIBUTES

Type attributes define how a function is called, or how a data object is accessed. This means that if you use a type attribute, it must be specified both when a function or data object is defined and when it is declared.

You can either place the type attributes explicitly in your declarations, or use the pragma directive `#pragma type_attribute`.

Type attributes can be further divided into *memory type attributes* and *general type attributes*. Memory type attributes are referred to as simply *memory attributes* in the rest of the documentation.

## Memory attributes

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

- Available *function memory attributes*: `__near_func` and `__far_func`
- Available *data memory attributes*: `__tiny`, `__near`, `__far`, `__huge`, `__regvar`, `__eeprom`, `__tinyflash`, `__flash`, `__farflash`, `__hugeflash`, `__generic`, `__io`, and `__ext_io`.

Data objects, functions, and destinations of pointers or C++ references always have a memory attribute. If no attribute is explicitly specified in the declaration or by the pragma directive `#pragma type_attribute`, an appropriate default attribute is used. You can specify one memory attribute for each level of pointer indirection.

## General type attributes

These general type attributes are available:

- *Function type attributes* affect how the function should be called: `__interrupt`, `__task`, and `__version_1`
- *Data type attributes*: `const` and `volatile`.

You can specify as many type attributes as required for each level of pointer indirection.

For more information about the type qualifiers `const` and `volatile`, see *Type qualifiers*, page 240.

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

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

Note that the attribute affects both identifiers.

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

```
#pragma type_attribute=__far
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 42.

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

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

and

__far char b;
char __far *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`:

|                             |                                                                      |
|-----------------------------|----------------------------------------------------------------------|
| <code>int __far * p;</code> | The <code>int</code> object is located in <code>__far</code> memory. |
| <code>int * __far p;</code> | The pointer is located in <code>__far</code> memory.                 |
| <code>__far int * p;</code> | The pointer is located in <code>__far</code> memory.                 |

### Syntax for type attributes on functions

The syntax for using type attributes on functions differs slightly from the syntax of type attributes on data objects. For functions, the attribute must be placed either in front of the return type, or in parentheses, for example:

```
__interrupt void my_handler(void);

or

void (__interrupt my_handler)(void);
```

This declaration of `my_handler` is equivalent with the previous one:

```
#pragma type_attribute=__interrupt
void my_handler(void);
```

### Syntax for type attributes on function pointers

To declare a function pointer, use this syntax:

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

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

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

```
typedef __farfunc 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

Normally, object attributes affect the internal functionality of functions and data objects, but not directly how the function is called or how the data is accessed. This means that an object attribute does not normally need to be present in the declaration of an object. Any exceptions to this rule are noted in the description of the attribute.

These object attributes are available:

- Object attributes that can be used for variables: `__no_init` and `__no_runtime_init`
- Object attributes that can be used for functions and variables: `location`, `@`, and `__root`
- Object attributes that can be used for functions: `__intrinsic`, `__monitor`, `__nested`, `__noreturn`, and `vector`.

You can specify as many object attributes as required for a specific function or data object.

For more information about `location` and `@`, see *Controlling data and function placement in memory*, page 164. For more information about `vector`, see *vector*, page 282.



### Syntax for object attributes

The object attribute must be placed in front of the type. For example, to place `myarray` in memory that is not initialized at startup:

```
__no_init int myarray[10];
```

The `#pragma object_attribute` directive can also be used. This declaration is equivalent to the previous one:

```
#pragma object_attribute=__no_init
int myarray[10];
```

**Note:** Object attributes cannot be used in combination with the `typedef` keyword.

## Summary of extended keywords

This table summarizes the extended keywords:

| Extended keyword         | Description                                                                                          |
|--------------------------|------------------------------------------------------------------------------------------------------|
| <code>__eeprom</code>    | Controls the storage of data objects in code memory space                                            |
| <code>__ext_io</code>    | Controls the storage of data objects in I/O memory space<br>Supports I/O instructions; used for SFRs |
| <code>__far</code>       | Controls the storage of data objects in data memory space                                            |
| <code>__farflash</code>  | Controls the storage of data objects in code memory space                                            |
| <code>__farfunc</code>   | Controls the storage of functions in code memory space                                               |
| <code>__flash</code>     | Controls the storage of data objects in code memory space                                            |
| <code>__generic</code>   | Declares a generic pointer                                                                           |
| <code>__huge</code>      | Controls the storage of data objects in data memory space                                            |
| <code>__hugeflash</code> | Controls the storage of data objects in code memory space                                            |
| <code>__interrupt</code> | Supports interrupt functions                                                                         |
| <code>__intrinsic</code> | Reserved for compiler internal use only                                                              |
| <code>__io</code>        | Controls the storage of data objects in I/O memory space<br>Supports I/O instructions; used for SFRs |
| <code>__monitor</code>   | Supports atomic execution of a function                                                              |
| <code>__near</code>      | Controls the storage of data objects in data memory space                                            |
| <code>__nearfunc</code>  | Controls the storage of functions in code memory space                                               |
| <code>__nested</code>    | Implements a nested interrupt                                                                        |
| <code>__no_init</code>   | Supports non-volatile memory                                                                         |

Table 43: Extended keywords summary

| Extended keyword               | Description                                                                       |
|--------------------------------|-----------------------------------------------------------------------------------|
| <code>__no_runtime_init</code> | Declares initialized variables that are not initialized at runtime.               |
| <code>__noreturn</code>        | Informs the compiler that the function will not return                            |
| <code>__raw</code>             | Prevents saving used registers in interrupt functions                             |
| <code>__regvar</code>          | Places a data object in a register                                                |
| <code>__root</code>            | Ensures that a function or variable is included in the object code even if unused |
| <code>__task</code>            | Relaxes the rules for preserving registers                                        |
| <code>__tiny</code>            | Controls the storage of data objects in data memory space                         |
| <code>__tinyflash</code>       | Controls the storage of data objects in code memory space                         |
| <code>__version_1</code>       | Specifies the old calling convention; available for backward compatibility        |

Table 43: Extended keywords summary (Continued)

## Descriptions of extended keywords

These sections give detailed information about each extended keyword.

### `__eeprom`

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | Follows the generic syntax rules for memory type attributes that can be used on data objects, see <i>Type attributes</i> , page 245.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| Description | <p>The <code>__eeprom</code> memory attribute overrides the default storage of variables given by the selected memory model and places individual initialized and non-initialized variables in the built-in EEPROM of the AVR microcontroller. These variables can be used like any other variable and provide a convenient way to access the built-in EEPROM.</p> <p>You can also override the supplied support functions to make the <code>__eeprom</code> memory attribute access an EEPROM or flash memory that is placed externally but not on the normal data bus, for example on an I2C bus.</p> <p>To do this, you must write a new EEPROM library routines file and include it in your project. The new file must use the same interface as the <code>eeprom.s90</code> file in the <code>avr\src\lib</code> directory (visible register use, the same entry points and the same semantics).</p> <p>Variables declared <code>__eeprom</code> are initialized only when a downloadable linker output file is downloaded to the system, and not every time the system startup code is executed.</p> |

You can also use the `__eeprom` attribute to create a pointer explicitly pointing to an object located in the EEPROM memory.

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Storage information | <ul style="list-style-type: none"> <li>● Memory space: EEPROM memory space</li> <li>● Address range: 0-0xFF (255 bytes) if <code>--eeprom_size ≤ 256</code> bytes, and 0-0xFFFF (64 Kbytes) if <code>--eeprom_size &gt; 256</code> bytes.</li> <li>● Maximum object size: 255 bytes if <code>--eeprom_size ≤ 256</code> bytes, and 65535 bytes if <code>--eeprom_size &gt; 256</code> bytes.</li> <li>● Pointer size: 1byte if <code>--eeprom_size ≤ 256</code> bytes, and 2 bytes if <code>--eeprom_size &gt; 256</code> bytes.</li> </ul> |
| Example             | <code>__eeprom int x;</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| See also            | <i>Memory types</i> , page 39 and <code>--eeprom_size</code> , page 204.                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |

## `__ext_io`

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax              | Follows the generic syntax rules for memory type attributes that can be used on data objects, see <i>Type attributes</i> , page 245.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| Description         | <p>The <code>__ext_io</code> memory attribute overrides the default storage of variables given by the selected memory model and allows individual objects to be accessed by use of the special I/O instructions in the AVR microcontroller. The <code>__ext_io</code> memory attribute implies that the object is <code>__no_init</code> and <code>volatile</code>.</p> <p>Your application may access the AVR I/O system by using the memory-mapped internal special function registers (SFRs). To access the AVR I/O system efficiently, the <code>__ext_io</code> memory attribute should be included in the code.</p> |
| Storage information | <ul style="list-style-type: none"> <li>● Memory space: Data memory space</li> <li>● Address range: 0x100-0xFFFF (64 Kbytes)</li> <li>● Maximum object size: 4 bytes.</li> <li>● Pointer size: Pointers not allowed.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                            |
| Example             | <code>__ext_io int x;</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| See also            | <i>Memory types</i> , page 39.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |

## **\_\_far**

|                     |                                                                                                                                                                                                                                                                                                                                                                                             |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax              | Follows the generic syntax rules for memory type attributes that can be used on data objects, see <i>Type attributes</i> , page 245.                                                                                                                                                                                                                                                        |
| Description         | The <code>__far</code> memory attribute overrides the default storage of variables given by the selected memory model and places individual variables and constants in far memory. You can also use the <code>__far</code> attribute to create a pointer explicitly pointing to an object located in the far memory.                                                                        |
| Storage information | <ul style="list-style-type: none"> <li>● Memory space: Data memory space</li> <li>● Address range: 0–0xFFFF (64 Kbytes)</li> <li>● Maximum object size: 32 Kbytes. An object cannot cross a 64-Kbyte boundary.</li> <li>● Pointer size: 2 bytes. Arithmetics is only performed on the two lower bytes, except comparison which is always performed on the entire 24-bit address.</li> </ul> |
| Example             | <code>__far int x;</code>                                                                                                                                                                                                                                                                                                                                                                   |
| See also            | <i>Memory types</i> , page 39.                                                                                                                                                                                                                                                                                                                                                              |

## **\_\_farflash**

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax              | Follows the generic syntax rules for memory type attributes that can be used on data objects, see <i>Type attributes</i> , page 245.                                                                                                                                                                                                                                                                                                                                                                                                                                |
| Description         | <p>The <code>__farflash</code> memory attribute overrides the default storage of variables given by the selected memory model and places individual variables and constants in flash (code) memory. You can also use the <code>__farflash</code> attribute to create a pointer explicitly pointing to an object located in the flash (code) memory.</p> <p><b>Note:</b> It is preferable to declare flash objects as <code>const</code>. The <code>__farflash</code> memory attribute is only available for AVR chips with more than 64 Kbytes of flash memory.</p> |
| Storage information | <ul style="list-style-type: none"> <li>● Memory space: Code memory space</li> <li>● Address range: 0–0x7FFFFFFF (8 Mbytes)</li> <li>● Maximum object size: 32 Kbytes. An object cannot cross a 64-Kbyte boundary.</li> <li>● Pointer size: 3 bytes. Arithmetics is only performed on the two lower bytes, except comparison which is always performed on the entire 24-bit address.</li> </ul>                                                                                                                                                                      |
| Example             | <code>__farflash int x;</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |

See also *Memory types*, page 39.

## **\_\_farfunc**

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax              | Follows the generic syntax rules for memory type attributes that can be used on functions, see <i>Type attributes</i> , page 245.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| Description         | <p>The <code>__farfunc</code> memory attribute overrides the default storage of functions given by the selected code model and places individual functions in code memory. You can also use the <code>__farfunc</code> attribute to create a pointer explicitly pointing to a function located in <code>FARCODE</code> memory (<code>0-0xFFFFF</code>).</p> <p>The default code pointer for the <code>-v5</code> and <code>-v6</code> processor options is <code>__farfunc</code>, and it only affects the size of the function pointers.</p> <p>Note that pointers with function memory attributes have restrictions in implicit and explicit casts when casting between pointers and also when casting between pointers and integer types.</p> <p>It is possible to call a <code>__nearfunc</code> function from a <code>__farfunc</code> function and vice versa. Only the size of the function pointer is affected.</p> |
| Storage information | <ul style="list-style-type: none"> <li>● Memory space: Code memory space</li> <li>● Address range: <code>0-0x7FFFFE</code> (8 Mbytes)</li> <li>● Pointer size: 3 bytes</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| Example             | <code>__farfunc void myfunction(void);</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| See also            | <i>Function storage</i> , page 47.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |

## **\_\_flash**

|             |                                                                                                                                                                                                                                                                                                                                                    |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | Follows the generic syntax rules for memory type attributes that can be used on data objects, see <i>Type attributes</i> , page 245.                                                                                                                                                                                                               |
| Description | <p>The <code>__flash</code> memory attribute overrides the default storage of variables given by the selected memory model and places individual variables and constants in flash (code) memory.</p> <p>Note that it is preferable to declare flash objects as constant. The <code>__flash</code> keyword is available for all AVR processors.</p> |

Because the AVR microcontrollers have only a small amount of on-board RAM, this memory should not be wasted on data that could be stored in flash memory (of which there is much more). However, because of the architecture of the processor, a default pointer cannot access the flash memory. The `__flash` keyword is used to identify that the constant should be stored in flash memory.

A header file called `pgmspace.h` is installed in the `avr\inc` directory, to provide some standard C library functions for taking strings stored in flash memory as arguments.

You can also use the `__flash` attribute to create a pointer explicitly pointing to an object located in the flash (code) memory.

**Note:** The `__flash` option cannot be used in a reduced ATtiny core.

#### Storage information

- Memory space: Code memory space
- Address range: 0–0xFFFF (64 Kbytes)
- Maximum object size: 32 Kbytes.
- Pointer size: 2 bytes.

#### Examples

A program defines a couple of strings that are stored in flash memory:

```
__flash char str1[] = "Message 1";
__flash char str2[] = "Message 2";
```

The program creates a `__flash` pointer to point to one of these strings, and assigns it to `str1`:

```
char __flash *msg;
msg=str1;
```

Using the `strcpy_P` function declared in `pgmspace.h`, a string in flash memory can be copied to another string in RAM as follows:

```
strcpy_P(dest, msg);
```

#### See also

*Memory types*, page 39.

## `__generic`

#### Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 245.

#### Description

The `__generic` pointer type attribute declares a generic pointer that can point to objects in both code and data space. The size of the generic pointer depends on which processor option is used.

The most significant bit (MSB) determines whether `__generic` points to Code (MSB=1) or Data (MSB=0).

It is not possible to place objects in a generic memory area, only to point to it. When using generic pointers, make sure that objects that have been declared `__far` and `__huge` are located in the range `0x0-0x7FFFFFFF`. Objects may still be placed in the entire data address space, but a generic pointer cannot point to objects in the upper half of the data address space.

The `__generic` keyword cannot be used with the `#pragma type_attribute` directive for a pointer.

Access through a `__generic` pointer is implemented with inline code. Because this type of access is slow and generates a lot of code, `__generic` pointers should be avoided when possible.

#### Storage information

- Memory space: Data or code memory space
- Address range: `0-0x7FFF` (32 Kbytes) for the processor options `-v0` and `-v1`, `0-07FFFFFF` (8 Mbytes) for the processor options `-v2` to `-v6`
- Generic pointer size: 1 + 15 bits when the processor options `-v0` and `-v1` are used. 1 + 23 bits when the processor options `-v2` to `-v6` are used.

#### See also

*Memory types*, page 39.

## `__huge`

#### Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 245.

#### Description

The `__huge` memory attribute overrides the default storage of variables given by the selected memory model and places individual variables and constants in huge memory. You can also use the `__huge` attribute to create a pointer explicitly pointing to an object located in the huge memory.

#### Storage information

- Memory space: Data memory space
- Address range: `0-0xFFFFFFFF` (16 Mbytes)
- Maximum object size: 16 Mbytes.
- Pointer size: 3 bytes.

#### Example

```
__huge int x;
```

#### See also

*Memory types*, page 39.

## **\_\_hugeflash**

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax              | Follows the generic syntax rules for memory type attributes that can be used on data objects, see <i>Type attributes</i> , page 245.                                                                                                                                                                                                                                                                                                                                                                                                                                |
| Description         | <p>The <code>__hugeflash</code> memory attribute overrides the default storage of variables given by the selected memory model and places individual variables and constants in flash (code) memory. You can also use the <code>__hugeflash</code> attribute to create a pointer explicitly pointing to an object located in the flash (code) memory.</p> <p>Note that it is preferable to declare flash objects as constant. The <code>__hugeflash</code> memory attribute is only available for AVR microcontrollers with at least 64 Kbytes of flash memory.</p> |
| Storage information | <ul style="list-style-type: none"> <li>● Memory space: Code memory space</li> <li>● Address range: 0–0x7FFFFFFF (8 Mbytes)</li> <li>● Maximum object size: 8 Mbytes.</li> <li>● Pointer size: 3 bytes.</li> </ul>                                                                                                                                                                                                                                                                                                                                                   |
| Example             | <code>__hugeflash int x;</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| See also            | <i>Memory types</i> , page 39.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |

## **\_\_interrupt**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | Follows the generic syntax rules for type attributes that can be used on functions, see <i>Type attributes</i> , page 245.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| Description | <p>The <code>__interrupt</code> keyword specifies interrupt functions. To specify one or several interrupt vectors, use the <code>#pragma vector</code> directive. The range of the interrupt vectors depends on the device used. It is possible to define an interrupt function without a vector, but then the compiler will not generate an entry in the interrupt vector table.</p> <p>An interrupt function must have a <code>void</code> return type and cannot have any parameters.</p> <p>The header file <code>iodevice.h</code>, where <i>device</i> corresponds to the selected device, contains predefined names for the existing interrupt vectors.</p> |
| Example     | <pre>#pragma vector=0x14 __interrupt void my_interrupt_handler(void);</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| See also    | <i>Interrupt functions</i> , page 48, <i>vector</i> , page 282, <i>INTVEC</i> , page 327.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |



## **\_\_intrinsic**

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

## **\_\_io**

**Syntax** Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 245.

**Description** The `__io` memory attribute overrides the default storage of variables given by the selected memory model and places individual variables and constants in I/O memory.

**Storage information**

- Memory space: Data memory space
- Address range: 0–0xFFFF (64 Kbytes)
- Maximum object size: 8 bytes (64 bits).
- Pointer size: Pointers not allowed.

**Example** `__io int x;`

**See also** *Memory types*, page 39.

## **\_\_monitor**

**Syntax** Follows the generic syntax rules for object attributes that can be used on functions, see *Object attributes*, page 248.

**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 49. For information about the intrinsic functions, see `__disable_interrupt`, page 285, `__enable_interrupt`, page 285, `__get_interrupt_state`, page 287, and `__set_interrupt_state`, page 290, respectively.

## **\_\_near**

|                     |                                                                                                                                                                                                                                                                                                                          |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax              | Follows the generic syntax rules for memory type attributes that can be used on data objects, see <i>Type attributes</i> , page 245.                                                                                                                                                                                     |
| Description         | The <code>__near</code> memory attribute overrides the default storage of variables given by the selected memory model and places individual variables and constants in near memory. You can also use the <code>__near</code> attribute to create a pointer explicitly pointing to an object located in the near memory. |
| Storage information | <ul style="list-style-type: none"> <li>● Memory space: Data memory space</li> <li>● Address range: 0–0xFFFF (64 Kbytes)</li> <li>● Maximum object size: 32 Kbytes.</li> <li>● Pointer size: 2 bytes.</li> </ul>                                                                                                          |
| Example             | <code>__near int x;</code>                                                                                                                                                                                                                                                                                               |
| See also            | <i>Memory types</i> , page 39.                                                                                                                                                                                                                                                                                           |

## **\_\_nearfunc**

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax              | Follows the generic syntax rules for memory type attributes that can be used on functions, see <i>Type attributes</i> , page 245.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| Description         | <p>The <code>__nearfunc</code> memory attribute overrides the default storage of functions given by the selected code model and places individual functions in near memory.</p> <p>Functions declared <code>__nearfunc</code> can be called from the entire code memory area, but must reside in the first 128 Kbytes of the code memory.</p> <p>The default for the <code>-v0</code> to <code>-v4</code> processor options is <code>__nearfunc</code>, and it only affects the size of the function pointers.</p> <p>Note that pointers with function memory attributes have restrictions in implicit and explicit casts when casting between pointers and also when casting between pointers and integer types.</p> <p>It is possible to call a <code>__nearfunc</code> function from a <code>__farfunc</code> function and vice versa. Only the size of the function pointer is affected.</p> <p>You can also use the <code>__nearfunc</code> attribute to create a pointer explicitly pointing to a function located in CODE memory (0–0xFFFF).</p> |
| Storage information | <ul style="list-style-type: none"> <li>● Memory space: Code memory space</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |

- Address range: 0–0x1FFFE (128 Kbytes)
- Pointer size: 2 bytes

Example `__nearfunc void myfunction(void);`

See also *Function storage*, page 47.

## **\_\_nested**

Syntax Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 245.

Description Use the `__nested` keyword to implement a nested interrupt, in other words a new interrupt can be served inside an interrupt function. A nested interrupt service routine acts like a normal interrupt service routine except that it sets the interrupt enable bit before any registers are saved.

Example `__nested __interrupt void myInterruptFunction()`

## **\_\_no\_init**

Syntax Follows the generic syntax rules for object attributes, see *Object attributes*, page 248.

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];`

## **\_\_no\_runtime\_init**

Syntax Follows the generic syntax rules for object attributes, see *Object attributes*, page 248.

Description The `__no_runtime_init` keyword is used for declaring initialized variables for which the initialization is performed when programming the device. These variables are not initialized at runtime during system startup.

**Note:** The `__no_runtime_init` keyword cannot be used in combination with the `typedef` keyword.

Example `__no_runtime_init int myarray[10];`

## **\_\_noreturn**

|             |                                                                                                                                                                                                                                                                                                        |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | Follows the generic syntax rules for object attributes, see <i>Object attributes</i> , page 248.                                                                                                                                                                                                       |
| Description | The <code>__noreturn</code> keyword can be used on a function to inform the compiler that the function will not return. If you use this keyword on such functions, the compiler can optimize more efficiently. Examples of functions that do not return are <code>abort</code> and <code>exit</code> . |
| Example     | <pre>__noreturn void terminate(void);</pre>                                                                                                                                                                                                                                                            |

## **\_\_raw**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                    |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | Follows the generic syntax rules for object attributes, see <i>Object attributes</i> , page 248.                                                                                                                                                                                                                                                                                                                   |
| Description | <p>This keyword prevents saving used registers in interrupt functions.</p> <p>Interrupt functions preserve the contents of all used processor registers at function entrance and restore them at exit. However, for some very special applications, it can be desirable to prevent the registers from being saved at function entrance. This can be accomplished by the use of the keyword <code>__raw</code>.</p> |
| Example     | <pre>__raw __interrupt void my_interrupt_function()</pre>                                                                                                                                                                                                                                                                                                                                                          |

## **\_\_regvar**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | Follows the generic syntax rules for type attributes that can be used on data, see <i>Type attributes</i> , page 245.                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| Description | <p>The <code>__regvar</code> extended keyword is used for declaring that a <i>global</i> or <i>static</i> variable should be placed permanently in the specified register or registers. The registers R4–R15 can be used for this purpose, provided that they have been locked with the <code>--lock_regs</code> compiler option.</p> <p><b>Note:</b></p> <ul style="list-style-type: none"> <li>• An object declared <code>__regvar</code> cannot have an initial value.</li> <li>• The <code>__regvar</code> option cannot be used in a reduced ATtiny core.</li> </ul> |
| Example     | <pre>__regvar __no_init int counter @ 14;</pre> <p>This will create the 16-bit integer variable <code>counter</code>, which will always be available in R15:R14. At least two registers must be locked.</p>                                                                                                                                                                                                                                                                                                                                                               |

|                     |                                                                                                                                                                                                                     |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Storage information | <ul style="list-style-type: none"> <li>● Memory space: Data memory space</li> <li>● Address range: 4–F</li> <li>● Maximum object size: 4 bytes (32 bits).</li> <li>● Pointer size: Pointers not allowed.</li> </ul> |
| See also            | <code>--memory_model</code> , <code>-m</code> , page 211, <code>--lock_regs</code> , page 210, and <i>Preserved registers</i> , page 126.                                                                           |

## **\_\_root**

|             |                                                                                                                                                                                                                                                               |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | Follows the generic syntax rules for object attributes, see <i>Object attributes</i> , page 248.                                                                                                                                                              |
| Description | A function or variable with the <code>__root</code> attribute is kept whether or not it is referenced from the rest of the application, provided its module is included. Program modules are always included and library modules are only included if needed. |
| Example     | <pre>__root int myarray[10];</pre>                                                                                                                                                                                                                            |
| See also    | For more information about modules, segments, and the link process, see the <i>IAR Linker and Library Tools Reference Guide</i> .                                                                                                                             |

## **\_\_task**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | Follows the generic syntax rules for type attributes that can be used on functions, see <i>Type attributes</i> , page 245.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| Description | <p>This keyword allows functions to relax the rules for preserving registers. Typically, the keyword is used on the start function for a task in an RTOS.</p> <p>By default, functions save the contents of used preserved registers on the stack upon entry, and restore them at exit. Functions that are declared <code>__task</code> do not save all registers, and therefore require less stack space.</p> <p>Because a function declared <code>__task</code> can corrupt registers that are needed by the calling function, you should only use <code>__task</code> on functions that do not return or call such a function from assembler code.</p> <p>The function <code>main</code> can be declared <code>__task</code>, unless it is explicitly called from the application. In real-time applications with more than one task, the root function of each task can be declared <code>__task</code>.</p> |
| Example     | <pre>__task void my_handler(void);</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |

## **\_\_tiny**

|                     |                                                                                                                                                                                                                                                                                                                          |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax              | Follows the generic syntax rules for memory type attributes that can be used on data objects, see <i>Type attributes</i> , page 245.                                                                                                                                                                                     |
| Description         | The <code>__tiny</code> memory attribute overrides the default storage of variables given by the selected memory model and places individual variables and constants in tiny memory. You can also use the <code>__tiny</code> attribute to create a pointer explicitly pointing to an object located in the tiny memory. |
| Storage information | <ul style="list-style-type: none"> <li>● Memory space: Data memory space</li> <li>● Address range: 0–0xFF (256 bytes)</li> <li>● Maximum object size: 127 bytes.</li> <li>● Pointer size: 1 byte.</li> </ul>                                                                                                             |
| Example             | <code>__tiny int x;</code>                                                                                                                                                                                                                                                                                               |
| See also            | <i>Memory types</i> , page 39.                                                                                                                                                                                                                                                                                           |

## **\_\_tinyflash**

|                     |                                                                                                                                                                                                                                                                                                                                                    |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax              | Follows the generic syntax rules for memory type attributes that can be used on data objects, see <i>Type attributes</i> , page 245.                                                                                                                                                                                                               |
| Description         | The <code>__tinyflash</code> memory attribute overrides the default storage of variables given by the selected memory model and places individual variables and constants in flash (code) memory. You can also use the <code>__tinyflash</code> attribute to create a pointer explicitly pointing to an object located in the flash (code) memory. |
| Storage information | <ul style="list-style-type: none"> <li>● Memory space: Code memory space</li> <li>● Address range: 0–0xFF (256 bytes)</li> <li>● Maximum object size: 127 bytes.</li> <li>● Pointer size: 1 byte.</li> </ul>                                                                                                                                       |
| Example             | <code>__tinyflash int x;</code>                                                                                                                                                                                                                                                                                                                    |
| See also            | <i>Memory types</i> , page 39.                                                                                                                                                                                                                                                                                                                     |

## **\_\_version\_1**

|             |                                                                                                                                                                                                                                                                                                                                                                |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | Follows the generic syntax rules for type attributes that can be used on functions, see <i>Type attributes</i> , page 245.                                                                                                                                                                                                                                     |
| Description | The <code>__version_1</code> keyword is available for backward compatibility of the interface for calling assembler routines from C. It makes a function use the calling convention of the A90 IAR C Compiler instead of the default calling convention. The <code>__version_1</code> calling convention is preferred when calling assembler functions from C. |
| Example     | <pre>__version_1 int func(int arg1, double arg2)</pre>                                                                                                                                                                                                                                                                                                         |
| See also    | <i>Calling convention</i> , page 123.                                                                                                                                                                                                                                                                                                                          |





# Pragma directives

This chapter describes the pragma directives of the compiler.

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

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

The pragma directives are always enabled in the compiler.

---

## Summary of pragma directives

This table lists the pragma directives of the compiler that can be used either with the `#pragma` preprocessor directive or the `_Pragma()` preprocessor operator:

| Pragma directive                     | Description                                            |
|--------------------------------------|--------------------------------------------------------|
| <code>basic_template_matching</code> | Makes a template function fully memory-attribute aware |
| <code>bitfields</code>               | Controls the order of bitfield members                 |
| <code>constseg</code>                | Places constant variables in a named segment           |
| <code>data_alignment</code>          | Gives a variable a higher (more strict) alignment      |
| <code>dataseg</code>                 | Places variables in a named segment                    |
| <code>diag_default</code>            | Changes the severity level of diagnostic messages      |
| <code>diag_error</code>              | Changes the severity level of diagnostic messages      |
| <code>diag_remark</code>             | Changes the severity level of diagnostic messages      |
| <code>diag_suppress</code>           | Suppresses diagnostic messages                         |
| <code>diag_warning</code>            | Changes the severity level of diagnostic messages      |
| <code>error</code>                   | Signals an error while parsing                         |
| <code>include_alias</code>           | Specifies an alias for an include file                 |
| <code>inline</code>                  | Controls inlining of a function                        |
| <code>language</code>                | Controls the IAR Systems language extensions           |

*Table 44: Pragma directives summary*

| Pragma directive      | Description                                                                                                |
|-----------------------|------------------------------------------------------------------------------------------------------------|
| location              | Specifies the absolute address of a variable, or places groups of functions or variables in named segments |
| message               | Prints a message                                                                                           |
| object_attribute      | Changes the definition of a variable or a function                                                         |
| optimize              | Specifies the type and level of an optimization                                                            |
| pack                  | Specifies the alignment of structures and union members                                                    |
| __printf_args         | Verifies that a function with a printf-style format string is called with the correct arguments            |
| required              | Ensures that a symbol that is needed by another symbol is included in the linked output                    |
| rtmodel               | Adds a runtime model attribute to the module                                                               |
| __scanf_args          | Verifies that a function with a scanf-style format string is called with the correct arguments             |
| section               | This directive is an alias for #pragma segment                                                             |
| segment               | Declares a segment name to be used by intrinsic functions                                                  |
| STDC CX_LIMITED_RANGE | Specifies whether the compiler can use normal complex mathematical formulas or not                         |
| STDC FENV_ACCESS      | Specifies whether your source code accesses the floating-point environment or not.                         |
| STDC FENV_ACCESS      | Specifies whether the compiler is allowed to contract floating-point expressions or not.                   |
| type_attribute        | Changes the declaration and definitions of a variable or function                                          |
| vector                | Specifies the vector of an interrupt or trap function                                                      |

Table 44: Pragma directives summary (Continued)

**Note:** For portability reasons, see also *Recognized pragma directives (6.10.6)*, page 340.

## Descriptions of pragma directives

This section gives detailed information about each pragma directive.

## basic\_template\_matching

|             |                                                                                                                                                                                                                                                                                                             |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma basic_template_matching</code>                                                                                                                                                                                                                                                                |
| Description | Use this pragma directive in front of a template function declaration to make the function fully memory-attribute aware, in the rare cases where this is useful. That template function will then match the template without the modifications, see <i>Templates and data memory attributes</i> , page 151. |
| Example     | <pre>#pragma basic_template_matching template&lt;typename T&gt; void fun(T *);  fun((int __near *) 0); /* Template parameter T becomes                         int __near */</pre>                                                                                                                          |

## bitfields

|             |                                                                                                                                                                                                                                                           |                                                                                         |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma bitfields={reversed default}</code>                                                                                                                                                                                                         |                                                                                         |
| Parameters  | <code>reversed</code>                                                                                                                                                                                                                                     | Bitfield members are placed from the most significant bit to the least significant bit. |
|             | <code>default</code>                                                                                                                                                                                                                                      | Bitfield members are placed from the least significant bit to the most significant bit. |
| Description | Use this pragma directive to control the order of bitfield members.                                                                                                                                                                                       |                                                                                         |
| Example     | <pre>#pragma bitfields=reversed /* Structure that uses reversed bitfields. */ struct S {     unsigned char  error : 1;     unsigned char  size  : 4;     unsigned short code  : 10; }; #pragma bitfields=default /* Restores to default setting. */</pre> |                                                                                         |
| See also    | <i>Bitfields</i> , page 233.                                                                                                                                                                                                                              |                                                                                         |

**constseg**

|             |                                                                                                                                                                                                                                                                        |                                                                                                                            |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------|
| Syntax      | #pragma constseg=[__memoryattribute ]{SEGMENT_NAME default}                                                                                                                                                                                                            |                                                                                                                            |
| Parameters  | <i>__memoryattribute</i>                                                                                                                                                                                                                                               | An optional memory attribute denoting in what memory the segment will be placed; if not specified, default memory is used. |
|             | <i>SEGMENT_NAME</i>                                                                                                                                                                                                                                                    | 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=__huge MY_CONSTANTS<br>const int factorySettings[] = {42, 15, -128, 0};<br>#pragma constseg=default                                                                                                                                                   |                                                                                                                            |

**data\_alignment**

|             |                                                                                                                                                                                                                                                                                                       |                                                          |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------|
| Syntax      | #pragma data_alignment= <i>expression</i>                                                                                                                                                                                                                                                             |                                                          |
| Parameters  | <i>expression</i>                                                                                                                                                                                                                                                                                     | A constant which must be a power of two (1, 2, 4, etc.). |
| Description | Use this pragma directive to give a variable a higher (more strict) alignment of the start address than it would otherwise have. This directive can be used on variables with static and automatic storage duration.                                                                                  |                                                          |
|             | When you use this directive on variables with automatic storage duration, there is an upper limit on the allowed alignment for each function, determined by the calling convention used.                                                                                                              |                                                          |
|             | <b>Note:</b> Normally, the size of a variable is a multiple of its alignment. The <code>data_alignment</code> directive only affects the alignment of the variable's start address, and not its size, and can thus be used for creating situations where the size is not a multiple of the alignment. |                                                          |

## dataseg

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma dataseg=[__memoryattribute ] {<i>SEGMENT_NAME</i> default}</code>                                                                                                                                                                                                                                                                                                                                                                |
| Parameters  | <div><div><code>__memoryattribute</code></div><div>An optional memory attribute denoting in what memory the segment will be placed; if not specified, default memory is used.</div></div> <div><div><code><i>SEGMENT_NAME</i></code></div><div>A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.</div></div> <div><div><code>default</code></div><div>Uses the default segment.</div></div> |
| Description | Use this pragma directive to place variables in a named segment. The segment name cannot be a segment name predefined for use by the compiler and linker. The variable will not be initialized at startup, and can for this reason not have an initializer, which means it must be declared <code>__no_init</code> . The setting remains active until you turn it off again with the <code>#pragma dataseg=default</code> directive.           |
| Example     | <pre>#pragma dataseg=__huge MY_SEGMENT __no_init char myBuffer[1000]; #pragma dataseg=default</pre>                                                                                                                                                                                                                                                                                                                                            |

## diag\_default

|             |                                                                                                                                                                                                                                                                               |                                                                           |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------|
| Syntax      | #pragma diag_default=tag[, tag, ...]                                                                                                                                                                                                                                          |                                                                           |
| Parameters  | <i>tag</i>                                                                                                                                                                                                                                                                    | The number of a diagnostic message, for example the message number Pe117. |
| Description | Use this pragma directive to change the severity level back to the default, or to the severity level defined on the command line by any of the options --diag_error, --diag_remark, --diag_suppress, or --diag_warnings, for the diagnostic messages specified with the tags. |                                                                           |
| See also    | <i>Diagnostics</i> , page 185.                                                                                                                                                                                                                                                |                                                                           |

**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 <code>error</code> for the specified diagnostics. |                                                                           |
| See also    | <i>Diagnostics</i> , page 185.                                                                              |                                                                           |

**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 <code>remark</code> for the specified diagnostic messages. |                                                                           |
| See also    | <i>Diagnostics</i> , page 185.                                                                                       |                                                                           |

**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    | <i>Diagnostics</i> , page 185.                                           |                                                                           |

## diag\_warning

|             |                                                                                                                       |                                                                           |
|-------------|-----------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------|
| Syntax      | #pragma diag_warning=tag[, tag, ...]                                                                                  |                                                                           |
| Parameters  | <i>tag</i>                                                                                                            | The number of a diagnostic message, for example the message number Pe826. |
| Description | Use this pragma directive to change the severity level to <code>warning</code> for the specified diagnostic messages. |                                                                           |
| See also    | <i>Diagnostics</i> , page 185.                                                                                        |                                                                           |

## error

|             |                                                                                                                                                                                                                                                                                                                                                       |                                             |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------|
| Syntax      | #pragma error <i>message</i>                                                                                                                                                                                                                                                                                                                          |                                             |
| Parameters  | <i>message</i>                                                                                                                                                                                                                                                                                                                                        | A string that represents the error message. |
| Description | Use this pragma directive to cause an error message when it is parsed. This mechanism is different from the preprocessor directive <code>#error</code> , because the <code>#pragma error</code> directive can be included in a preprocessor macro using the <code>_Pragma</code> form of the directive and only causes an error if the macro is used. |                                             |
| Example     | <pre>#if FOO_AVAILABLE #define FOO ... #else #define FOO _Pragma("error\Foo is not available") #endif</pre> <p>If <code>FOO_AVAILABLE</code> is zero, an error will be signaled if the <code>FOO</code> macro is used in actual source code.</p>                                                                                                      |                                             |

## include\_alias

|            |                                                                                                                                      |                                                                  |
|------------|--------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------|
| Syntax     | <pre>#pragma include_alias ("orig_header" , "subst_header") #pragma include_alias (&lt;orig_header&gt; , &lt;subst_header&gt;)</pre> |                                                                  |
| Parameters | <i>orig_header</i>                                                                                                                   | The name of a header file for which you want to create an alias. |
|            | <i>subst_header</i>                                                                                                                  | The alias for the original header file.                          |

|             |                                                                                                                                                                                                                                                                                                                                                                                                      |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | <p>Use this pragma directive to provide an alias for a header file. This is useful for substituting one header file with another, and for specifying an absolute path to a relative file.</p> <p>This pragma directive must appear before the corresponding <code>#include</code> directives and <code>subst_header</code> must match its corresponding <code>#include</code> directive exactly.</p> |
| Example     | <pre>#pragma include_alias (&lt;stdio.h&gt; , &lt;C:\MyHeaders\stdio.h&gt;)<br/>#include &lt;stdio.h&gt;</pre> <p>This example will substitute the relative file <code>stdio.h</code> with a counterpart located according to the specified path.</p>                                                                                                                                                |
| See also    | <i>Include file search procedure</i> , page 182.                                                                                                                                                                                                                                                                                                                                                     |

**inline**

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |              |                                                         |                     |                                                         |                    |                                                                                          |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------|---------------------------------------------------------|---------------------|---------------------------------------------------------|--------------------|------------------------------------------------------------------------------------------|
| Syntax              | <code>#pragma inline[=forced =never]</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |              |                                                         |                     |                                                         |                    |                                                                                          |
| Parameters          | <table><tr><td>No parameter</td><td>Has the same effect as the <code>inline</code> keyword.</td></tr><tr><td><code>forced</code></td><td>Disables the compiler's heuristics and forces inlining.</td></tr><tr><td><code>never</code></td><td>Disables the compiler's heuristics and makes sure that the function will not be inlined.</td></tr></table>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               | No parameter | Has the same effect as the <code>inline</code> keyword. | <code>forced</code> | Disables the compiler's heuristics and forces inlining. | <code>never</code> | Disables the compiler's heuristics and makes sure that the function will not be inlined. |
| No parameter        | Has the same effect as the <code>inline</code> keyword.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |              |                                                         |                     |                                                         |                    |                                                                                          |
| <code>forced</code> | Disables the compiler's heuristics and forces inlining.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |              |                                                         |                     |                                                         |                    |                                                                                          |
| <code>never</code>  | Disables the compiler's heuristics and makes sure that the function will not be inlined.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |              |                                                         |                     |                                                         |                    |                                                                                          |
| Description         | <p>Use <code>#pragma inline</code> to advise the compiler that the function whose declaration follows immediately after the directive should be expanded inline into the body of the calling function. Whether the inlining actually occurs is subject to the compiler's heuristics.</p> <p><code>#pragma inline</code> is similar to the C++ keyword <code>inline</code>. The difference is that the compiler uses C++ inline semantics for the <code>#pragma inline</code> directive, but uses the Standard C semantics for the <code>inline</code> keyword.</p> <p>Specifying <code>#pragma inline=never</code> disables the compiler's heuristics and makes sure that the function will not be inlined.</p> <p>Specifying <code>#pragma inline=forced</code> will force the compiler to treat the function declared immediately after the directive as a candidate for inlining, regardless of the compiler's heuristics. If the compiler fails to inline the candidate function for some reason, for example due to recursion, a warning message is emitted.</p> |              |                                                         |                     |                                                         |                    |                                                                                          |



Inlining is normally performed only on the High optimization level. Specifying `#pragma inline=forced` will enable inlining of the function in question also on the Medium optimization level.

See also *Function inlining*, page 170.

language

Syntax `#pragma language={extended|default|save|restore}`

Parameters

|                           |                                                                                                                                                                                                                                                                             |
|---------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>extended</code>     | Enables the IAR Systems language extensions from the first use of the pragma directive and onward.                                                                                                                                                                          |
| <code>default</code>      | From the first use of the pragma directive and onward, restores the settings for the IAR Systems language extensions to whatever that was specified by compiler options.                                                                                                    |
| <code>save restore</code> | Saves and restores, respectively, the IAR Systems language extensions setting around a piece of source code.<br>Each use of <code>save</code> must be followed by a matching <code>restore</code> in the same file without any intervening <code>#include</code> directive. |

Description Use this pragma directive to control the use of language extensions.

Example 1 At the top of a file that needs to be compiled with IAR Systems extensions enabled:

```
#pragma language=extended
/* The rest of the file. */
```

Example 2 Around a particular part of the source code that needs to be compiled with IAR Systems extensions enabled, but where the state before the sequence cannot be assumed to be the same as that specified by the compiler options in use:

```
#pragma language=save
#pragma language=extended
/* Part of source code. */
#pragma language=restore
```

See also `-e`, page 203 and `--strict`, page 224.

**location**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |                                                                                                      |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------|
| Syntax      | #pragma location={ <i>address</i>   <i>NAME</i> }                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |                                                                                                      |
| Parameters  | <i>address</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 | The absolute address of the global or static variable for which you want an absolute location.       |
|             | <i>NAME</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    | A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker. |
| Description | Use this pragma directive to specify the location—the absolute address—of the global or static variable whose declaration follows the pragma directive. The variables must be declared either <code>__no_init</code> or <code>const</code> . Alternatively, the directive can take a string specifying a segment for placing either a variable or a function whose declaration follows the pragma directive. Do not place variables declared as <code>__no_init</code> and variables declared as <code>const</code> in the same named segment. |                                                                                                      |
| Example     | <pre>#pragma location=0x2000 __no_init volatile char PORT1; /* PORT1 is located at address                                 0x2000 */  #pragma location="foo" char PORT1; /* PORT1 is located in segment foo */  /* A better way is to use a corresponding mechanism */ #define FLASH _Pragma("location=\"FLASH\"") ... FLASH int i; /* i is placed in the FLASH segment */</pre>                                                                                                                                                               |                                                                                                      |
| See also    | <i>Controlling data and function placement in memory</i> , page 164.                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |                                                                                                      |

**message**

|            |                                   |                                                                    |
|------------|-----------------------------------|--------------------------------------------------------------------|
| Syntax     | #pragma message( <i>message</i> ) |                                                                    |
| Parameters | <i>message</i>                    | The message that you want to direct to the standard output stream. |

|             |                                                                                                                         |
|-------------|-------------------------------------------------------------------------------------------------------------------------|
| Description | Use this pragma directive to make the compiler print a message to the standard output stream when the file is compiled. |
| Example:    | <pre>#ifdef TESTING #pragma message("Testing") #endif</pre>                                                             |

object\_attribute

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma object_attribute=object_attribute[,object_attribute,...]</code>                                                                                                                                                                                                                                                                                                                                                        |
| Parameters  | For information about object attributes that can be used with this pragma directive, see <i>Object attributes</i> , page 248.                                                                                                                                                                                                                                                                                                        |
| 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 <code>#pragma type_attribute</code> that specifies the storing and accessing of a variable or function, it is not necessary to specify an object attribute in declarations. |
| Example     | <pre>#pragma object_attribute=__no_init char bar;</pre>                                                                                                                                                                                                                                                                                                                                                                              |
| See also    | <i>General syntax rules for extended keywords</i> , page 245.                                                                                                                                                                                                                                                                                                                                                                        |

optimize

|            |                                                                                                                                                                                                                                                                                                       |
|------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax     | <code>#pragma optimize=[goal][level][no_optimization...]</code>                                                                                                                                                                                                                                       |
| Parameters | <div> <div>goal</div> <div>Choose between:<br/>balanced, optimizes balanced between speed and size<br/>size, optimizes for size<br/>speed, optimizes for speed.</div> </div> <div> <div>level</div> <div>Specifies the level of optimization; choose between none, low, medium, or high.</div> </div> |

|                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |                       |                                                                 |            |                                             |                   |                                      |
|-----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------|-----------------------------------------------------------------|------------|---------------------------------------------|-------------------|--------------------------------------|
|                       | <div><div><code>no_optimization</code></div><div>Disables one or several optimizations; choose between:<br/><code>no_code_motion</code>, disables code motion<br/><code>no_cse</code>, disables common subexpression elimination<br/><code>no_inline</code>, disables function inlining<br/><code>no_tbaa</code>, disables type-based alias analysis<br/><code>no_unroll</code>, disables loop unrolling.</div></div>                                                                                                                                                                                                                                                                                                                                                                                                                                           |                       |                                                                 |            |                                             |                   |                                      |
| Description           | <div><p>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.</p><p>The parameters <code>speed</code>, <code>size</code>, and <code>balanced</code> only have effect on the <code>high</code> 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.</p><p><b>Note:</b> If you use the <code>#pragma optimize</code> directive to specify an optimization level that is higher than the optimization level you specify using a compiler option, the pragma directive is ignored.</p></div> |                       |                                                                 |            |                                             |                   |                                      |
| Example               | <div><pre>#pragma optimize=speed int SmallAndUsedOften() {     /* Do something here. */ }  #pragma optimize=size int BigAndSeldomUsed() {     /* Do something here. */ }</pre></div>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |                       |                                                                 |            |                                             |                   |                                      |
| <b>pack</b>           |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |                       |                                                                 |            |                                             |                   |                                      |
| Syntax                | <div><pre>#pragma pack(<i>n</i>) #pragma pack() #pragma pack({push pop}[ ,<i>name</i>] [ ,<i>n</i>])</pre></div>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |                       |                                                                 |            |                                             |                   |                                      |
| Parameters            | <div><table><tr><td><code><i>n</i></code></td><td>Sets an optional structure alignment; one of: 1, 2, 4, 8, or 16</td></tr><tr><td>Empty list</td><td>Restores the structure alignment to default</td></tr><tr><td><code>push</code></td><td>Sets a temporary structure alignment</td></tr></table></div>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       | <code><i>n</i></code> | Sets an optional structure alignment; one of: 1, 2, 4, 8, or 16 | Empty list | Restores the structure alignment to default | <code>push</code> | Sets a temporary structure alignment |
| <code><i>n</i></code> | Sets an optional structure alignment; one of: 1, 2, 4, 8, or 16                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |                       |                                                                 |            |                                             |                   |                                      |
| Empty list            | Restores the structure alignment to default                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |                       |                                                                 |            |                                             |                   |                                      |
| <code>push</code>     | Sets a temporary structure alignment                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |                       |                                                                 |            |                                             |                   |                                      |

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                          |                                                                      |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------|
|             | <code>pop</code>                                                                                                                                                                                                                                                                                                                                                                                                                         | Restores the structure alignment from a temporarily pushed alignment |
|             | <code>name</code>                                                                                                                                                                                                                                                                                                                                                                                                                        | An optional pushed or popped alignment label                         |
| Description | <p>Use this pragma directive to specify the maximum alignment of <code>struct</code> and <code>union</code> members.</p> <p>The <code>#pragma pack</code> directive affects declarations of structures following the pragma directive to the next <code>#pragma pack</code> or the end of the compilation unit.</p> <p><b>Note:</b> This can result in significantly larger and slower code when accessing members of the structure.</p> |                                                                      |
| See also    | <i>Structure types</i> , page 240.                                                                                                                                                                                                                                                                                                                                                                                                       |                                                                      |

## \_\_printf\_args

|             |                                                                                                                                                                                                                                         |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma __printf_args</code>                                                                                                                                                                                                      |
| Description | Use this pragma directive on a function with a printf-style format string. For any call to that function, the compiler verifies that the argument to each conversion specifier (for example <code>%d</code> ) is syntactically correct. |
| Example     | <pre>#pragma __printf_args int printf(char const *,...);  void PrintNumbers(unsigned short x) {     printf("%d", x); /* Compiler checks that x is an integer */ }</pre>                                                                 |

## required

|             |                                                                                                                                                                                           |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma required=symbol</code>                                                                                                                                                      |
| Parameters  | <p><i>symbol</i></p> <p>Any statically linked function or variable.</p>                                                                                                                   |
| Description | Use this pragma directive to ensure that a symbol which is needed by a second symbol is included in the linked output. The directive must be placed immediately before the second symbol. |

Use the directive if the requirement for a symbol is not otherwise visible in the application, for example if a variable is only referenced indirectly through the segment it resides in.

Example

```
const char copyright[] = "Copyright by me";

#pragma required=copyright
int main()
{
 /* Do something here. */
}
```

Even if the copyright string is not used by the application, it will still be included by the linker and available in the output.

rtmodel

Syntax

```
#pragma rtmodel="key", "value"
```

Parameters

|         |                                                                                                                                                      |
|---------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| "key"   | A text string that specifies the runtime model attribute.                                                                                            |
| "value" | A text string that specifies the value of the runtime model attribute. Using the special value * is equivalent to not defining the attribute at all. |

Description

Use this pragma directive to add a runtime model attribute to a module, which can be used by the linker to check consistency between modules.

This pragma directive is useful for enforcing consistency between modules. All modules that are linked together and define the same runtime attribute key must have the same value for the corresponding key, or the special value \*. It can, however, be useful to state explicitly that the module can handle any runtime model.

A module can have several runtime model definitions.

**Note:** The predefined compiler runtime model attributes start with a double underscore. To avoid confusion, this style must not be used in the user-defined attributes.

Example

```
#pragma rtmodel="I2C", "ENABLED"
```

The linker will generate an error if a module that contains this definition is linked with a module that does not have the corresponding runtime model attributes defined.

See also

*Checking module consistency*, page 103.

## \_\_scanf\_args

|             |                                                                                                                                                                                                                                                         |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <pre>#pragma __scanf_args</pre>                                                                                                                                                                                                                         |
| 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     | <pre>#pragma __scanf_args int scanf(char const *,...);  int GetNumber() {     int nr;     scanf("%d", &amp;nr); /* Compiler checks that                        the argument is a                        pointer to an integer */     return nr; }</pre> |

## segment

|                          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |             |                          |                          |                                                                                                                              |              |                                                                                                                   |
|--------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------|--------------------------|--------------------------|------------------------------------------------------------------------------------------------------------------------------|--------------|-------------------------------------------------------------------------------------------------------------------|
| Syntax                   | <pre>#pragma segment="NAME" [__memoryattribute] [align] alias #pragma section="NAME" [__memoryattribute] [align]</pre>                                                                                                                                                                                                                                                                                                                                                                               |             |                          |                          |                                                                                                                              |              |                                                                                                                   |
| Parameters               | <table> <tr> <td><i>NAME</i></td><td>The name of the segment.</td></tr> <tr> <td><i>__memoryattribute</i></td><td>An optional memory attribute identifying the memory the segment will be placed in; if not specified, default memory is used.</td></tr> <tr> <td><i>align</i></td><td>Specifies an alignment for the segment part. The value must be a constant integer expression to the power of two.</td></tr> </table>                                                                          | <i>NAME</i> | The name of the segment. | <i>__memoryattribute</i> | An optional memory attribute identifying the memory the segment will be placed in; if not specified, default memory is used. | <i>align</i> | Specifies an alignment for the segment part. The value must be a constant integer expression to the power of two. |
| <i>NAME</i>              | The name of the segment.                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |             |                          |                          |                                                                                                                              |              |                                                                                                                   |
| <i>__memoryattribute</i> | An optional memory attribute identifying the memory the segment will be placed in; if not specified, default memory is used.                                                                                                                                                                                                                                                                                                                                                                         |             |                          |                          |                                                                                                                              |              |                                                                                                                   |
| <i>align</i>             | Specifies an alignment for the segment part. The value must be a constant integer expression to the power of two.                                                                                                                                                                                                                                                                                                                                                                                    |             |                          |                          |                                                                                                                              |              |                                                                                                                   |
| Description              | <p>Use this pragma directive to define a segment name that can be used by the segment operators <code>__segment_begin</code>, <code>__segment_end</code>, and <code>__segment_size</code>. All segment declarations for a specific segment must have the same memory type attribute and alignment.</p> <p>If an optional memory attribute is used, the return type of the segment operators <code>__segment_begin</code> and <code>__segment_end</code> is:</p> <pre>void __memoryattribute *.</pre> |             |                          |                          |                                                                                                                              |              |                                                                                                                   |

|          |                                                                                                                                                      |
|----------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| Example  | <code>#pragma segment="MYHUGE" __huge 4</code>                                                                                                       |
| See also | <i>Dedicated segment operators</i> , page 139. For more information about segments and segment parts, see the chapter <i>Placing code and data</i> . |

**STDC CX\_LIMITED\_RANGE**

|             |                                                                                                                                                                                                                                                                                                          |                                                    |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------|
| Syntax      | <code>#pragma STDC CX_LIMITED_RANGE {ON OFF DEFAULT}</code>                                                                                                                                                                                                                                              |                                                    |
| Parameters  | ON                                                                                                                                                                                                                                                                                                       | Normal complex mathematic formulas can be used.    |
|             | OFF                                                                                                                                                                                                                                                                                                      | Normal complex mathematic formulas cannot be used. |
|             | DEFAULT                                                                                                                                                                                                                                                                                                  | Sets the default behavior, that is OFF.            |
| Description | Use this pragma directive to specify that the compiler can use the normal complex mathematic formulas for $\times$ (multiplication), $/$ (division), and <code>abs</code> .<br><br><b>Note:</b> This directive is required by Standard C. The directive is recognized but has no effect in the compiler. |                                                    |

**STDC FENV\_ACCESS**

|             |                                                                                                                                                                             |                                                                                                                |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>#pragma STDC FENV_ACCESS {ON OFF DEFAULT}</code>                                                                                                                      |                                                                                                                |
| Parameters  | ON                                                                                                                                                                          | Source code accesses the floating-point environment. Note that this argument is not supported by the compiler. |
|             | OFF                                                                                                                                                                         | Source code does not access the floating-point environment.                                                    |
|             | DEFAULT                                                                                                                                                                     | Sets the default behavior, that is OFF.                                                                        |
| Description | Use this pragma directive to specify whether your source code accesses the floating-point environment or not.<br><br><b>Note:</b> This directive is required by Standard C. |                                                                                                                |



## STDC FP\_CONTRACT

|             |                                                                                                                                                               |                                                                                                                               |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | #pragma STDC FP_CONTRACT {ON OFF DEFAULT}                                                                                                                     |                                                                                                                               |
| Parameters  | ON                                                                                                                                                            | The compiler is allowed to contract floating-point expressions.                                                               |
|             | OFF                                                                                                                                                           | The compiler is not allowed to contract floating-point expressions. Note that this argument is not supported by the compiler. |
|             | DEFAULT                                                                                                                                                       | Sets the default behavior, that is ON.                                                                                        |
| Description | Use this pragma directive to specify whether the compiler is allowed to contract floating-point expressions or not. This directive is required by Standard C. |                                                                                                                               |
| Example     | #pragma STDC_FP_CONTRACT=ON                                                                                                                                   |                                                                                                                               |

## type\_attribute

|             |                                                                                                                                                                                                                                                                                                                                                                            |  |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| Syntax      | #pragma type_attribute=type_attribute[, type_attribute, ...]                                                                                                                                                                                                                                                                                                               |  |
| Parameters  | For information about type attributes that can be used with this pragma directive, see <i>Type attributes</i> , page 245.                                                                                                                                                                                                                                                  |  |
| Description | <p>Use this pragma directive to specify IAR-specific <i>type attributes</i>, which are not part of Standard C. Note however, that a given type attribute might not be applicable to all kind of objects.</p> <p>This directive affects the declaration of the identifier, the next variable, or the next function that follows immediately after the pragma directive.</p> |  |
| Example     | <p>In this example, an <code>int</code> object with the memory attribute <code>__near</code> is defined:</p> <pre>#pragma type_attribute=__near int x;</pre> <p>This declaration, which uses extended keywords, is equivalent:</p> <pre>__near int x;</pre>                                                                                                                |  |
| See also    | See the chapter <i>Extended keywords</i> for more information.                                                                                                                                                                                                                                                                                                             |  |

**vector**

|             |                                                                                                                                                                                                 |                                                        |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------|
| Syntax      | #pragma vector=vector1[, vector2, vector3, ...]                                                                                                                                                 |                                                        |
| Parameters  | <i>vectorN</i>                                                                                                                                                                                  | The vector number(s) of an interrupt or trap function. |
| Description | Use this pragma directive to specify the vector(s) of an interrupt or trap function whose declaration follows the pragma directive. Note that several vectors can be defined for each function. |                                                        |
| Example     | #pragma vector=0x14<br>__interrupt void my_handler(void);                                                                                                                                       |                                                        |

# Intrinsic functions

This chapter gives reference information about the intrinsic functions, a predefined set of functions available in the compiler.

The intrinsic functions provide direct access to low-level processor operations and can be very useful in, for example, time-critical routines. The intrinsic functions compile into inline code, either as a single instruction or as a short sequence of instructions.

---

## Summary of intrinsic functions

To use intrinsic functions in an application, include the header file `intrinsics.h`.

Note that the intrinsic function names start with double underscores, for example:

`__disable_interrupt`

This table summarizes the intrinsic functions:

| Intrinsic function                                      | Description                                         |
|---------------------------------------------------------|-----------------------------------------------------|
| <code>__delay_cycles</code>                             | Inserts a time delay                                |
| <code>__DES_decryption</code>                           | Decrypts according to Digital Encryption Standard   |
| <code>__DES_encryption</code>                           | Encrypts according to Digital Encryption Standard   |
| <code>__disable_interrupt</code>                        | Disables interrupts                                 |
| <code>__enable_interrupt</code>                         | Enables interrupts                                  |
| <code>__extended_load_program_memory</code>             | Returns one byte from code memory                   |
| <code>__fractional_multiply_signed</code>               | Generates an <code>FMULS</code> instruction         |
| <code>__fractional_multiply_signed_with_unsigned</code> | Generates an <code>FMULSU</code> instruction        |
| <code>__fractional_multiply_unsigned</code>             | Generates an <code>FMUL</code> instruction          |
| <code>__get_interrupt_state</code>                      | Returns the interrupt state                         |
| <code>__indirect_jump_to</code>                         | Generates an <code>IJMP</code> instruction          |
| <code>__insert_opcode</code>                            | Assigns a value to a processor register             |
| <code>__lac</code>                                      | Provides access to the <code>LAC</code> instruction |

*Table 45: Intrinsic functions summary*

| Intrinsic function                           | Description                                         |
|----------------------------------------------|-----------------------------------------------------|
| <code>__las</code>                           | Provides access to the <code>LAS</code> instruction |
| <code>__lat</code>                           | Provides access to the <code>LAT</code> instruction |
| <code>__load_program_memory</code>           | Returns one byte from program memory                |
| <code>__multiply_signed</code>               | Generates a <code>MULS</code> instruction           |
| <code>__multiply_signed_with_unsigned</code> | Generates a <code>MULSU</code> instruction          |
| <code>__multiply_unsigned</code>             | Generates a <code>MUL</code> instruction            |
| <code>__no_operation</code>                  | Inserts a <code>NOP</code> instruction              |
| <code>__require</code>                       | Sets a constant literal as required                 |
| <code>__restore_interrupt</code>             | Restores the interrupt flag                         |
| <code>__reverse</code>                       | Reverses the byte order of a value                  |
| <code>__save_interrupt</code>                | Saves the state of the interrupt flag               |
| <code>__set_interrupt_state</code>           | Restores the interrupt state                        |
| <code>__sleep</code>                         | Inserts a <code>SLEEP</code> instruction            |
| <code>__swap_nibbles</code>                  | Swaps bit 0-3 with bit 4-7                          |
| <code>__watchdog_reset</code>                | Inserts a watchdog reset instruction                |
| <code>__xch</code>                           | Provides access to the <code>XCH</code> instruction |

Table 45: Intrinsic functions summary (Continued)

## Descriptions of intrinsic functions

This section gives reference information about each intrinsic function.

### `__delay_cycles`

|             |                                                                                                                                                                                                                                                                                            |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>void __delay_cycles(unsigned long int);</code>                                                                                                                                                                                                                                       |
| Description | <p>Makes the compiler generate code that takes the given amount of cycles to perform, that is, it inserts a time delay that lasts the specified number of cycles.</p> <p>The specified value must be a constant integer expression and not an expression that is evaluated at runtime.</p> |

### `__DES_decryption`

|        |                                                                                                    |
|--------|----------------------------------------------------------------------------------------------------|
| Syntax | <code>unsigned long long __DES_decryption(unsigned long long data, unsigned long long key);</code> |
|--------|----------------------------------------------------------------------------------------------------|

where:

|             |                            |
|-------------|----------------------------|
| <i>data</i> | The data to decrypt        |
| <i>key</i>  | The key to decrypt against |

|             |                                                                                                                                                                                                                                                                                                      |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | Decrypts according to the Digital Encryption Standard (DES). <code>__DES_decryption</code> performs a DES decryption of <i>data</i> against <i>key</i> and returns the decrypted data.<br><br><b>Note:</b> This intrinsic function is available only for cores with support for the DES instruction. |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

**\_\_DES\_encryption**

|        |                                                                                                                |
|--------|----------------------------------------------------------------------------------------------------------------|
| Syntax | <pre>unsigned long long __DES_encryption(unsigned long long data, unsigned long long key);</pre> <p>where:</p> |
|--------|----------------------------------------------------------------------------------------------------------------|

|             |                            |
|-------------|----------------------------|
| <i>data</i> | The data to encrypt        |
| <i>key</i>  | The key to encrypt against |

|             |                                                                                                                                                                                                                                                                                         |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | Encrypts according to the Digital Encryption Standard (DES). <code>__DES_encryption</code> performs a DES encryption of <i>data</i> against <i>key</i> and returns the encrypted data.<br><br>This intrinsic function is available only for cores with support for the DES instruction. |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

**\_\_disable\_interrupt**

|        |                                            |
|--------|--------------------------------------------|
| Syntax | <pre>void __disable_interrupt(void);</pre> |
|--------|--------------------------------------------|

|             |                                                                    |
|-------------|--------------------------------------------------------------------|
| Description | Disables interrupts by inserting the <code>CLI</code> instruction. |
|-------------|--------------------------------------------------------------------|

**\_\_enable\_interrupt**

|        |                                           |
|--------|-------------------------------------------|
| Syntax | <pre>void __enable_interrupt(void);</pre> |
|--------|-------------------------------------------|

|             |                                                                   |
|-------------|-------------------------------------------------------------------|
| Description | Enables interrupts by inserting the <code>SEI</code> instruction. |
|-------------|-------------------------------------------------------------------|

## **\_\_extended\_load\_program\_memory**

|             |                                                                                                           |
|-------------|-----------------------------------------------------------------------------------------------------------|
| Syntax      | <code>unsigned char __extended_load_program_memory(unsigned char __farflash *);</code>                    |
| Description | Returns one byte from code memory.<br>Use this intrinsic function to access constant data in code memory. |

## **\_\_fractional\_multiply\_signed**

|             |                                                                                 |
|-------------|---------------------------------------------------------------------------------|
| Syntax      | <code>signed int __fractional_multiply_signed(signed char, signed char);</code> |
| Description | Generates an <code>FMULS</code> instruction.                                    |

## **\_\_fractional\_multiply\_signed\_with\_unsigned**

|             |                                                                                                 |
|-------------|-------------------------------------------------------------------------------------------------|
| Syntax      | <code>signed int __fractional_multiply_signed_with_unsigned(signed char, unsigned char);</code> |
| Description | Generates an <code>FMULSU</code> instruction.                                                   |

## **\_\_fractional\_multiply\_unsigned**

|             |                                                                                         |
|-------------|-----------------------------------------------------------------------------------------|
| Syntax      | <code>unsigned int __fractional_multiply_unsigned(unsigned char, unsigned char);</code> |
| Description | Generates an <code>FMUL</code> instruction.                                             |

## \_\_get\_interrupt\_state

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>__istate_t __get_interrupt_state(void);</code>                                                                                                                                                                                                                                                                                                                                                                                                                     |
| Description | Returns the global interrupt state. The return value can be used as an argument to the <code>__set_interrupt_state</code> intrinsic function, which will restore the interrupt state.                                                                                                                                                                                                                                                                                    |
| Example     | <pre>#include "intrinsics.h"  void CriticalFn() {     __istate_t s = __get_interrupt_state();     __disable_interrupt();      /* Do something here. */      __set_interrupt_state(s); }</pre> <p>The advantage of using this sequence of code compared to using <code>__disable_interrupt</code> and <code>__enable_interrupt</code> is that the code in this example will not enable any interrupts disabled before the call of <code>__get_interrupt_state</code>.</p> |

## \_\_indirect\_jump\_to

|             |                                                                                                                                                                                                                                                                                                                                        |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>void __indirect_jump_to(unsigned long);</code>                                                                                                                                                                                                                                                                                   |
| Description | Jumps to the address specified by the argument by using the <code>IJMP</code> or <code>EIJMP</code> instruction, depending on the generic processor option: <div style="margin-left: 20px;"> <code>IJMP</code>                      <code>-v0 to -v4</code><br/> <code>EIJMP</code>                    <code>-v5 and -v6</code> </div> |

## \_\_insert\_opcode

|             |                                                    |
|-------------|----------------------------------------------------|
| Syntax      | <code>void __insert_opcode(unsigned short);</code> |
| Description | Inserts a <code>DW unsigned</code> directive.      |

## \_\_lac

|        |                                                                   |
|--------|-------------------------------------------------------------------|
| Syntax | <code>unsigned char __lac(unsigned char, unsigned char *);</code> |
|--------|-------------------------------------------------------------------|

|             |                                                          |
|-------------|----------------------------------------------------------|
| Description | Provides access to the LAC (Load And Clear) instruction. |
|-------------|----------------------------------------------------------|

## **\_\_las**

|        |                                                                   |
|--------|-------------------------------------------------------------------|
| Syntax | <code>unsigned char __las(unsigned char, unsigned char *);</code> |
|--------|-------------------------------------------------------------------|

|             |                                                        |
|-------------|--------------------------------------------------------|
| Description | Provides access to the LAS (Load And Set) instruction. |
|-------------|--------------------------------------------------------|

## **\_\_lat**

|        |                                                                   |
|--------|-------------------------------------------------------------------|
| Syntax | <code>unsigned char __lat(unsigned char, unsigned char *);</code> |
|--------|-------------------------------------------------------------------|

|             |                                                           |
|-------------|-----------------------------------------------------------|
| Description | Provides access to the LAT (Load And Toggle) instruction. |
|-------------|-----------------------------------------------------------|

## **\_\_load\_program\_memory**

|        |                                                                            |
|--------|----------------------------------------------------------------------------|
| Syntax | <code>unsigned char __load_program_memory(unsigned char __flash *);</code> |
|--------|----------------------------------------------------------------------------|

|             |                                                                                                       |
|-------------|-------------------------------------------------------------------------------------------------------|
| Description | Returns one byte from code memory. The constants must be placed within the first 64 Kbytes of memory. |
|-------------|-------------------------------------------------------------------------------------------------------|

## **\_\_multiply\_signed**

|        |                                                                      |
|--------|----------------------------------------------------------------------|
| Syntax | <code>signed int __multiply_signed(signed char, signed char);</code> |
|--------|----------------------------------------------------------------------|

|             |                               |
|-------------|-------------------------------|
| Description | Generates a MULS instruction. |
|-------------|-------------------------------|

## **\_\_multiply\_signed\_with\_unsigned**

|        |                                                                                      |
|--------|--------------------------------------------------------------------------------------|
| Syntax | <code>signed int __multiply_signed_with_unsigned(signed char, unsigned char);</code> |
|--------|--------------------------------------------------------------------------------------|

|             |                                |
|-------------|--------------------------------|
| Description | Generates a MULSU instruction. |
|-------------|--------------------------------|

## **\_\_multiply\_unsigned**

|        |                                                                              |
|--------|------------------------------------------------------------------------------|
| Syntax | <code>unsigned int __multiply_unsigned(unsigned char, unsigned char);</code> |
|--------|------------------------------------------------------------------------------|

|             |                              |
|-------------|------------------------------|
| Description | Generates a MUL instruction. |
|-------------|------------------------------|



**\_\_no\_operation**

|             |                                         |
|-------------|-----------------------------------------|
| Syntax      | <code>void __no_operation(void);</code> |
| Description | Inserts a NOP instruction.              |

**\_\_require**

|             |                                      |
|-------------|--------------------------------------|
| Syntax      | <code>void __require(void *);</code> |
| Description | Sets a constant literal as required. |

One of the prominent features of the IAR XLINK Linker is its ability to strip away anything that is not needed. This is a very good feature because it reduces the resulting code size to a minimum. However, in some situations you may want to be able to explicitly include a piece of code or a variable even though it is not directly used.

The argument to `__require` could be a variable, a function name, or an exported assembler label. It must, however, be a constant literal. The label referred to will be treated as if it would be used at the location of the `__require` call.

|         |                                                                                                                           |
|---------|---------------------------------------------------------------------------------------------------------------------------|
| Example | In this example, the copyright message will be included in the generated binary file even though it is not directly used. |
|---------|---------------------------------------------------------------------------------------------------------------------------|

```
#include <intrinsics.h>
char copyright[] = "Copyright 2011 by XXXX";
void main(void)
{
 __require(copyright);
 [... the rest of the program ...]
}
```

**\_\_restore\_interrupt**

|             |                                                                |
|-------------|----------------------------------------------------------------|
| Syntax      | <code>void __restore_interrupt(unsigned char oldState);</code> |
| Description | Restores the interrupt flag to the specified state.            |

**Note:** The value of `oldState` must be the result of a call to the `__save_interrupt` intrinsic function.

## **\_\_reverse**

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>unsigned int __reverse(unsigned int);</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| Description | Reverses the byte order of the value given as parameter. Avoid using <code>__reverse</code> in complex expressions as it might introduce extra register copying.                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| Example     | <pre>signed int      __reverse( signed int); unsigned long   __reverse(unsigned long); signed long     __reverse( signed long); void __far *    __reverse(void __far *); /* Only on -v4 */                                            /* and -v6 */ void __huge *   __reverse(void __huge *); /* Only on -v4 */                                            /* and -v6 */ void __farflash * __reverse(void __farflash *); /* Only on -v2 through -v6 with &gt; 64k flash memory */ void __hugeflash * __reverse(void __hugeflash *); /* Only on -v2 through -v6 with &gt; 64k flash memory */</pre> |

## **\_\_save\_interrupt**

|             |                                                                                                                                                                                                     |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>unsigned char __save_interrupt(void);</code>                                                                                                                                                  |
| Description | Saves the state of the interrupt flag in the byte returned. This value can then be used for restoring the state of the interrupt flag with the <code>__restore_interrupt</code> intrinsic function. |
| Example     | <pre>unsigned char oldState;  oldState = __save_interrupt(); __disable_interrupt();  /* Critical section goes here */  __restore_interrupt(oldState);</pre>                                         |

## **\_\_set\_interrupt\_state**

|              |                                                                                                                                                                                                                                         |
|--------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax       | <code>void __set_interrupt_state(__istate_t);</code>                                                                                                                                                                                    |
| Descriptions | <p>Restores the interrupt state to a value previously returned by the <code>__get_interrupt_state</code> function.</p> <p>For information about the <code>__istate_t</code> type, see <code>__get_interrupt_state</code>, page 287.</p> |

**\_\_sleep**

Syntax `void __sleep(void);`

Description Inserts a *SLEEP* instruction.

**\_\_swap\_nibbles**

Syntax `unsigned char __swap_nibbles(unsigned char);`

Description Swaps bit 0-3 with bit 4-7 of the parameter and returns the swapped value.

**\_\_watchdog\_reset**

Syntax `void __watchdog_reset(void);`

Description Inserts a watchdog reset instruction.

**\_\_xch**

Syntax `unsigned char __xch(unsigned char, unsigned char *);`

Description Provides access to the *XCH* (Exchange) instruction.



# The preprocessor

This chapter gives a brief overview of the preprocessor, including reference information about the different preprocessor directives, symbols, and other related information.

---

## Overview of the preprocessor

The preprocessor of the IAR C/C++ Compiler for AVR adheres to Standard C. The compiler also makes these preprocessor-related features available to you:

- **Predefined preprocessor symbols**  
These symbols allow you to inspect the compile-time environment, for example the time and date of compilation. For more information, see *Descriptions of predefined preprocessor symbols*, page 294.
- **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 197.
- **Preprocessor extensions**  
There are several preprocessor extensions, for example many pragma directives; for more information, see the chapter *Pragma directives* in this guide. For information about the corresponding `_Pragma` operator and the other extensions related to the preprocessor, see *Descriptions of miscellaneous preprocessor extensions*, page 299.
- **Preprocessor output**  
Use the option `--preprocess` to direct preprocessor output to a named file, see *--preprocess*, page 219.

To specify a path for an include file, use forward slashes:

```
#include "mydirectory/myfile"
```

In source code, use forward slashes:

```
file = fopen("mydirectory/myfile", "rt");
```

Note that backslashes can also be used. In this case, use one in include file paths and two in source code strings.

# Descriptions of predefined preprocessor symbols

This table describes the predefined preprocessor symbols:

| Predefined symbol | Identifies                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|-------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| __BASE_FILE__     | A string that identifies the name of the base source file (that is, not the header file), being compiled. See also __FILE__, page 295, and <code>--no_path_in_file_macros</code> , page 214.                                                                                                                                                                                                                                                              |
| __BUILD_NUMBER__  | A unique integer that identifies the build number of the compiler currently in use. The build number does not necessarily increase with a compiler that is released later.                                                                                                                                                                                                                                                                                |
| __CORE__          | An integer that identifies the processor variant in use. The symbol corresponds to the processor option <code>-vn</code> in use.                                                                                                                                                                                                                                                                                                                          |
| __cplusplus       | An integer which is defined when the compiler runs in any of the C++ modes, otherwise it is undefined. When defined, its value is 199711L. This symbol can be used with <code>#ifdef</code> to detect whether the compiler accepts C++ code. It is particularly useful when creating header files that are to be shared by C and C++ code.*                                                                                                               |
| __CPU__           | A symbol that identifies the processor variant in use. This symbol expands to a number which corresponds to the processor option <code>-vn</code> in use.                                                                                                                                                                                                                                                                                                 |
| __DATE__          | A string that identifies the date of compilation, which is returned in the form "Mmm dd yyyy", for example "Oct 30 2010".*                                                                                                                                                                                                                                                                                                                                |
| __device__        | A symbol that identifies the processor variant in use. This symbol corresponds to the processor variant that you have specified with the <code>--cpu</code> compiler option. <i>device</i> corresponds exactly to the device name, except for FpSLic, which uses the predefined symbol __AT94Kxx__.<br>For example, the symbol is __AT90S2313__ when the <code>--cpu=2313</code> option is used, and __ATmega163__, when <code>--cpu=m163</code> is used. |
| __DOUBLE__        | An integer that identifies the size of the data type <code>double</code> . The symbols is defined to 32 or 64, depending on the setting of the option <code>--64bit_doubles</code> .                                                                                                                                                                                                                                                                      |

Table 46: Predefined symbols

| Predefined symbol                  | Identifies                                                                                                                                                                                                                                                                                                           |
|------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>__embedded_cplusplus</code>  | An integer which is defined to 1 when the compiler runs in any of the C++ modes, otherwise the symbol is undefined. This symbol can be used with <code>#ifdef</code> to detect whether the compiler accepts C++ code. It is particularly useful when creating header files that are to be shared by C and C++ code.* |
| <code>__FILE__</code>              | A string that identifies the name of the file being compiled, which can be both the base source file and any included header file. See also <code>__BASE_FILE__</code> , page 294, and <code>--no_path_in_file_macros</code> , page 214.*                                                                            |
| <code>__func__</code>              | A predefined string identifier that is initialized with the name of the function in which the symbol is used. This is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see -e, page 203. See also <code>__PRETTY_FUNCTION__</code> , page 297.             |
| <code>__FUNCTION__</code>          | A predefined string identifier that is initialized with the name of the function in which the symbol is used. This is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see -e, page 203. See also <code>__PRETTY_FUNCTION__</code> , page 297.             |
| <code>__HAS_EEPROM__</code>        | A symbol that determines whether there is internal EEPROM available or not. When this symbol is defined, there is internal EEPROM available. When this symbol is not defined, there is no internal EEPROM available.                                                                                                 |
| <code>__HAS_EIND__</code>          | A symbol that determines whether the instruction <code>EIND</code> is available or not. When this symbol is defined, the instruction <code>EIND</code> is available. When this symbol is not defined, the <code>EIND</code> instruction is not available.                                                            |
| <code>__HAS_ELPM__</code>          | A symbol that determines whether the instruction <code>ELPM</code> is available or not. When this symbol is defined, the instruction <code>ELPM</code> is available. When this symbol is not defined, the <code>ELPM</code> instruction is not available.                                                            |
| <code>__HAS_ENHANCED_CORE__</code> | A symbol that determines whether the enhanced core is used or not. The symbol reflects the <code>--enhanced_core</code> option and is defined when the enhanced core is used. When this symbol is not defined, the enhanced core is not used.                                                                        |

Table 46: Predefined symbols (Continued)

| Predefined symbol   | Identifies                                                                                                                                                                                                                                                                        |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| __HAS_FISCR__       | A symbol that determines whether the instruction <code>FISCR</code> is available or not. When this symbol is defined, the instruction <code>FISCR</code> is available. When this symbol is not defined, the <code>FISCR</code> instruction is not available.                      |
| __HAS_MUL__         | A symbol that determines whether the instruction <code>MUL</code> is available or not. When this symbol is defined, the instruction <code>MUL</code> is available. When this symbol is not defined, the <code>MUL</code> instruction is not available.                            |
| __HAS_RAMPD__       | A symbol that determines whether the register <code>RAMPD</code> is available or not. When this symbol is defined, the register <code>RAMPD</code> is available. When this symbol is not defined, the register <code>RAMPD</code> is not available.                               |
| __HAS_RAMPX__       | A symbol that determines whether the register <code>RAMPX</code> is available or not. When this symbol is defined, the register <code>RAMPX</code> is available. When this symbol is not defined, the register <code>RAMPX</code> is not available.                               |
| __HAS_RAMPY__       | A symbol that determines whether the register <code>RAMPY</code> is available or not. When this symbol is defined, the register <code>RAMPY</code> is available. When this symbol is not defined, the register <code>RAMPY</code> is not available.                               |
| __HAS_RAMPZ__       | A symbol that determines whether the register <code>RAMPZ</code> is available or not. When this symbol is defined, the register <code>RAMPZ</code> is available. When this symbol is not defined, the register <code>RAMPZ</code> is not available.                               |
| __IAR_SYSTEMS_ICC__ | An integer that identifies the IAR compiler platform. The current value is 9. Note that the number could be higher in a future version of the product. This symbol can be tested with <code>#ifdef</code> to detect whether the code was compiled by a compiler from IAR Systems. |
| __ICCAVR__          | An integer that is set to 1 when the code is compiled with the IAR C/C++ Compiler for AVR.                                                                                                                                                                                        |
| __LINE__            | An integer that identifies the current source line number of the file being compiled, which can be both the base source file and any included header file.*                                                                                                                       |
| __LITTLE_ENDIAN__   | An integer that is defined to 1 because the AVR byte order is little-endian.                                                                                                                                                                                                      |

Table 46: Predefined symbols (Continued)



| Predefined symbol   | Identifies                                                                                                                                                                                                                                                                                                                                                                            |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| __MEMORY_MODEL__    | An integer that identifies the memory model in use. The symbol reflects the <code>--memory_model</code> option and is defined to 1 for the Tiny memory model, 2 for the Small memory model, and 3 for the Large memory model.                                                                                                                                                         |
| __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 <code>"void func(char)"</code> . This symbol is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see <code>-e</code> , page 203. See also <code>__func__</code> , page 295. |
| __STDC__            | An integer that is set to 1, which means the compiler adheres to Standard C. This symbol can be tested with <code>#ifdef</code> to detect whether the compiler in use adheres to Standard C.*                                                                                                                                                                                         |
| __STDC_VERSION__    | An integer that identifies the version of the C standard in use. The symbol expands to <code>199901L</code> , unless the <code>--c89</code> compiler option is used in which case the symbol expands to <code>199409L</code> . This symbol does not apply in EC++ mode.*                                                                                                              |
| __SUBVERSION__      | An integer that identifies the subversion number of the compiler version number, for example 3 in 5.30.3.                                                                                                                                                                                                                                                                             |
| __TIME__            | A string that identifies the time of compilation in the form <code>"hh:mm:ss"</code> .*                                                                                                                                                                                                                                                                                               |

Table 46: Predefined symbols (Continued)

| Predefined symbol   | Identifies                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| __TID__             | <p>A symbol that expands to the target identifier which contains these parts:</p> <ul style="list-style-type: none"><li>• A target identifier (t) unique for each IAR compiler. For the AVR microcontroller, the target identifier is 90.</li><li>• The value (c) of the --cpu or -v option.</li><li>• The value (m) corresponding to the --memory_model option in use; where the value 1 corresponds to Tiny, the value 2 corresponds to Small, and the value 3 to Large.</li></ul> <p>The __TID__ value is constructed as:</p> <pre>((t &lt;&lt; 8)   (c &lt;&lt; 4)   m)</pre> <p>You can extract the values as follows:</p> <pre>t = (__TID__ &gt;&gt; 8) &amp; 0x7F; /* target id */ c = (__TID__ &gt;&gt; 4) &amp; 0x0F; /* cpu core */ m = __TID__ &amp; 0x0F;      /* memory model */</pre> <p>To find the value of the target identifier for the current compiler, execute:</p> <pre>printf("%ld", (__TID__ &gt;&gt; 8) &amp; 0x7F)</pre> <p><b>Note:</b> The use of __TID__ is not recommended. We recommend that you use the symbols __ICCAVR__ and __CORE__ instead.</p> |
| __TINY_AVR__        | <p>A symbol that determines whether the reduced tiny AVR core is used or not. When this symbol is defined, the reduced tiny AVR core is used. When this symbol is not defined, the reduced tiny AVR core is not used.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| __VER__             | <p>An integer that identifies the version number of the IAR compiler in use. The value of the number is calculated in this way: (100 * the major version number + the minor version number). For example, for compiler version 3.34, 3 is the major version number and 34 is the minor version number. Hence, the value of __VER__ is 334.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| __VERSION_1_CALLS__ | <p>A symbol that expands to 1 if the used calling convention is the old calling convention used in compiler version 1.x. If zero, the new calling convention is used. For more information about the calling conventions, see <i>Calling convention</i>, page 123.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| __XMEGA_CORE__      | <p>A symbol that determines whether the xmega core is used or not. When this symbol is defined, the xmega core is used. When this symbol is not defined, the xmega core is not used.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |

Table 46: Predefined symbols (Continued)

\* This symbol is required by Standard C.

## Descriptions of miscellaneous preprocessor extensions

This section gives reference information about the preprocessor extensions that are available in addition to the predefined symbols, pragma directives, and Standard C directives.

### NDEBUG

#### Description

This preprocessor symbol determines whether any assert macros you have written in your application shall be included or not in the built application.

If this symbol is not defined, all assert macros are evaluated. If the symbol is defined, all assert macros are excluded from the compilation. In other words, if the symbol is:

- **defined**, the assert code will *not* be included
- **not defined**, the assert code will be included

This means that if you write any assert code and build your application, you should define this symbol to exclude the assert code from the final application.

Note that the assert macro is defined in the `assert.h` standard include file.

#### See also

*Assert*, page 98.



In the IDE, the `NDEBUG` symbol is automatically defined if you build your application in the Release build configuration.

### #warning message

#### Syntax

`#warning message`

where *message* can be any string.

#### Description

Use this preprocessor directive to produce messages. Typically, this is useful for assertions and other trace utilities, similar to the way the Standard C `#error` directive is used. This directive is not recognized when the `--strict` compiler option is used.



# Library functions

This chapter gives an introduction to the C and C++ library functions. It also lists the header files used for accessing library definitions.

At the end of this chapter, all AVR-specific library functions are described.

For detailed reference information about the library functions, see the online help system.

---

## Library overview

The compiler provides two different libraries:

- IAR DLIB Library is a complete library, compliant with Standard C and C++. This library also supports floating-point numbers in IEEE 754 format and it can be configured to include different levels of support for locale, file descriptors, multibyte characters, et cetera.
- IAR CLIB Library is a light-weight library, which is not fully compliant with Standard C. Neither does it fully support floating-point numbers in IEEE 754 format or does it support C++.

Note that different customization methods are normally needed for these two libraries. For more information, see the chapter *The DLIB runtime environment* and *The CLIB runtime environment*, respectively.

For detailed information about the library functions, see the online documentation supplied with the product. There is also keyword reference information for the DLIB library functions. To obtain reference information for a function, select the function name in the editor window and press F1.

For more information about library functions, see the chapter *Implementation-defined behavior* in this guide.

## HEADER FILES

Your application program gains access to library definitions through header files, which it incorporates using the `#include` directive. The definitions are divided into several different header files, each covering a particular functional area, letting you include just those that are required.

It is essential to include the appropriate header file before making any reference to its definitions. Failure to do so can cause the call to fail during execution, or generate error or warning messages at compile time or link time.

## **LIBRARY OBJECT FILES**

Most of the library definitions can be used without modification, that is, directly from the library object files that are supplied with the product. For information about how to choose a runtime library, see *Basic project configuration*, page 29. The linker will include only those routines that are required—directly or indirectly—by your application.

## **ALTERNATIVE MORE ACCURATE LIBRARY FUNCTIONS**

The default implementation of `cos`, `sin`, `tan`, and `pow` is designed to be fast and small. As an alternative, there are versions designed to provide better accuracy. They are named `__iar_xxx_accuratef` for float variants of the functions and `__iar_xxx_accuratel` for long double variants of the functions, and where `xxx` is `cos`, `sin`, etc.

To use any of these more accurate versions, use the `-e` linker option.

## **REENTRANCY**

A function that can be simultaneously invoked in the main application and in any number of interrupts is reentrant. A library function that uses statically allocated data is therefore not reentrant.

Most parts of the DLIB library are reentrant, but the following functions and parts are not reentrant because they need static data:

- Heap functions—`malloc`, `free`, `realloc`, `calloc`, and the C++ operators `new` and `delete`
- Locale functions—`localeconv`, `setlocale`
- Multibyte functions—`mbrlen`, `mbrtowc`, `mbsrtowc`, `mbtowc`, `wcrtomb`, `wcsrtomb`, `wctomb`
- Rand functions—`rand`, `srand`
- Time functions—`asctime`, `localtime`, `gmtime`, `mktime`
- The miscellaneous functions `atexit`, `strerror`, `strtok`
- Functions that use files or the heap in some way. This includes `printf`, `sprintf`, `scanf`, `sscanf`, `getchar`, and `putchar`.

For the CLIB library, the `qsort` function and functions that use files in some way are non-reentrant. This includes `printf`, `scanf`, `getchar`, and `putchar`. However, the functions `sprintf` and `sscanf` are reentrant.

Functions that can set `errno` are not reentrant, because an `errno` value resulting from one of these functions can be destroyed by a subsequent use of the function before it is read. This applies to math and string conversion functions, among others.

Remedies for this are:

- Do not use non-reentrant functions in interrupt service routines
- Guard calls to a non-reentrant function by a mutex, or a secure region, etc.

## THE LONGJMP FUNCTION



A `longjmp` is in effect a jump to a previously defined `setjmp`. Any variable length arrays or C++ objects residing on the stack during stack unwinding will not be destroyed. This can lead to resource leaks or incorrect application behavior.

---

## IAR DLIB Library

The IAR DLIB Library provides most of the important C and C++ library definitions that apply to embedded systems. These are of the following types:

- Adherence to a free-standing implementation of Standard C. The library supports most of the hosted functionality, but you must implement some of its base functionality. For additional information, see the chapter *Implementation-defined behavior* in this guide.
- Standard C library definitions, for user programs.
- C++ library definitions, for user programs.
- `CSTARTUP`, the module containing the start-up code, see the chapter *The DLIB runtime environment* in this guide.
- Runtime support libraries; for example low-level floating-point routines.
- Intrinsic functions, allowing low-level use of AVR features. See the chapter *Intrinsic functions* for more information.
- Special compiler support for accessing strings in flash memory, see *AVR-specific library functions*, page 310.

In addition, the IAR DLIB Library includes some added C functionality, see *Added C functionality*, page 307.

## C HEADER FILES

This section lists the header files specific to the DLIB library C definitions. Header files may additionally contain target-specific definitions; these are documented in the chapter *Using C*.

This table lists the C header files:

| Header file | Usage                                                              |
|-------------|--------------------------------------------------------------------|
| assert.h    | Enforcing assertions when functions execute                        |
| complex.h   | Computing common complex mathematical functions                    |
| ctype.h     | Classifying characters                                             |
| errno.h     | Testing error codes reported by library functions                  |
| fenv.h      | Floating-point exception flags                                     |
| float.h     | Testing floating-point type properties                             |
| inttypes.h  | Defining formatters for all types defined in <code>stdint.h</code> |
| iso646.h    | Using Amendment 1— <code>iso646.h</code> 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 <code>bool</code> data type in C.             |
| stddef.h    | Defining several useful types and macros                           |
| stdint.h    | Providing integer characteristics                                  |
| stdio.h     | Performing input and output                                        |
| stdlib.h    | Performing a variety of operations                                 |
| string.h    | Manipulating several kinds of strings                              |
| tgmath.h    | Type-generic mathematical functions                                |
| time.h      | Converting between various time and date formats                   |
| uchar.h     | Unicode functionality (IAR extension to Standard C)                |
| wchar.h     | Support for wide characters                                        |
| wctype.h    | Classifying wide characters                                        |

Table 47: Traditional Standard C header files—DLIB

C++ HEADER FILES

This section lists the C++ header files.



### Embedded C++

This table lists the Embedded C++ header files:

| Header file               | Usage                                                                             |
|---------------------------|-----------------------------------------------------------------------------------|
| <code>complex</code>      | Defining a class that supports complex arithmetic                                 |
| <code>fstream</code>      | Defining several I/O stream classes that manipulate external files                |
| <code>iomanip</code>      | Declaring several I/O stream manipulators that take an argument                   |
| <code>ios</code>          | Defining the class that serves as the base for many I/O streams classes           |
| <code>iosfwd</code>       | Declaring several I/O stream classes before they are necessarily defined          |
| <code>iostream</code>     | Declaring the I/O stream objects that manipulate the standard streams             |
| <code>istream</code>      | Defining the class that performs extractions                                      |
| <code>new</code>          | Declaring several functions that allocate and free storage                        |
| <code>ostream</code>      | Defining the class that performs insertions                                       |
| <code>sstream</code>      | Defining several I/O stream classes that manipulate string containers             |
| <code>streambuf</code>    | Defining classes that buffer I/O stream operations                                |
| <code>string</code>       | Defining a class that implements a string container                               |
| <code>stringstream</code> | Defining several I/O stream classes that manipulate in-memory character sequences |

Table 48: Embedded C++ header files

### Extended Embedded C++ standard template library

The following table lists the Extended EC++ standard template library (STL) header files:

| Header file             | Description                                            |
|-------------------------|--------------------------------------------------------|
| <code>algorithm</code>  | Defines several common operations on sequences         |
| <code>deque</code>      | A deque sequence container                             |
| <code>functional</code> | Defines several function objects                       |
| <code>hash_map</code>   | A map associative container, based on a hash algorithm |
| <code>hash_set</code>   | A set associative container, based on a hash algorithm |
| <code>iterator</code>   | Defines common iterators, and operations on iterators  |
| <code>list</code>       | A doubly-linked list sequence container                |
| <code>map</code>        | A map associative container                            |
| <code>memory</code>     | Defines facilities for managing memory                 |
| <code>numeric</code>    | Performs generalized numeric operations on sequences   |

Table 49: Standard template library header files

| Header file | Description                             |
|-------------|-----------------------------------------|
| queue       | A queue sequence container              |
| set         | A set associative container             |
| slist       | A singly-linked list sequence container |
| stack       | A stack sequence container              |
| utility     | Defines several utility components      |
| vector      | A vector sequence container             |

Table 49: Standard template library header files (Continued)

Using Standard C libraries in C++

The C++ library works in conjunction with some of the header files from the Standard C library, sometimes with small alterations. The header files come in two forms—new and traditional—for example, `cassert` and `assert.h`.

This table shows the new header files:

| Header file | Usage                                                              |
|-------------|--------------------------------------------------------------------|
| cassert     | Enforcing assertions when functions execute                        |
| complex     | Computing common complex mathematical functions                    |
| cctype      | Classifying characters                                             |
| cerrno      | Testing error codes reported by library functions                  |
| cfenv       | Floating-point exception flags                                     |
| cfloat      | Testing floating-point type properties                             |
| cinttypes   | Defining formatters for all types defined in <code>stdint.h</code> |
| ciso646     | Using Amendment 1— <code>iso646.h</code> standard header           |
| climits     | Testing integer type properties                                    |
| clocale     | Adapting to different cultural conventions                         |
| cmath       | Computing common mathematical functions                            |
| csetjmp     | Executing non-local goto statements                                |
| csignal     | Controlling various exceptional conditions                         |
| cstdarg     | Accessing a varying number of arguments                            |
| cstdbool    | Adds support for the <code>bool</code> data type in C.             |
| cstddef     | Defining several useful types and macros                           |
| stdint      | Providing integer characteristics                                  |
| stdio       | Performing input and output                                        |

Table 50: New Standard C header files—DLIB

| Header file          | Usage                                            |
|----------------------|--------------------------------------------------|
| <code>cstdlib</code> | Performing a variety of operations               |
| <code>cstring</code> | Manipulating several kinds of strings            |
| <code>ctgmath</code> | Type-generic mathematical functions              |
| <code>ctime</code>   | Converting between various time and date formats |
| <code>cwchar</code>  | Support for wide characters                      |
| <code>cwctype</code> | Classifying wide characters                      |

Table 50: New Standard C header files—DLIB (Continued)

### LIBRARY FUNCTIONS AS INTRINSIC FUNCTIONS

Certain C library functions will under some circumstances be handled as intrinsic functions and will generate inline code instead of an ordinary function call, for example `memcpy`, `memset`, and `strcat`.

### ADDED C FUNCTIONALITY

The IAR DLIB Library includes some added C functionality.

The following include files provide these features:

- `fenv.h`
- `stdio.h`
- `stdlib.h`
- `string.h`
- `time.h`

#### **fenv.h**

In `fenv.h`, trap handling support for floating-point numbers is defined with the functions `fegettrapenable` and `fegettrapdisable`. No floating-point status flags are supported.

#### **stdio.h**

These functions provide additional I/O functionality:

|                     |                                                                                     |
|---------------------|-------------------------------------------------------------------------------------|
| <code>fdopen</code> | Opens a file based on a low-level file descriptor.                                  |
| <code>fileno</code> | Gets the low-level file descriptor from the file descriptor ( <code>FILE*</code> ). |
| <code>__gets</code> | Corresponds to <code>fgets</code> on <code>stdin</code> .                           |

|                            |                                                                 |
|----------------------------|-----------------------------------------------------------------|
| <code>getw</code>          | Gets a <code>wchar_t</code> character from <code>stdin</code> . |
| <code>putw</code>          | Puts a <code>wchar_t</code> character to <code>stdout</code> .  |
| <code>__ungetchar</code>   | Corresponds to <code>ungetc</code> on <code>stdout</code> .     |
| <code>__write_array</code> | Corresponds to <code>fwrite</code> on <code>stdout</code> .     |

## **string.h**

These are the additional functions defined in `string.h`:

|                          |                                                |
|--------------------------|------------------------------------------------|
| <code>strdup</code>      | Duplicates a string on the heap.               |
| <code>strcasecmp</code>  | Compares strings case-insensitive.             |
| <code>strncasecmp</code> | Compares strings case-insensitive and bounded. |
| <code>strnlen</code>     | Bounded string length.                         |

## **time.h**

There are two interfaces for using `time_t` and the associated functions `time`, `ctime`, `difftime`, `gmtime`, `localtime`, and `mktime`:

- The 32-bit interface supports years from 1900 up to 2035 and uses a 32-bit integer for `time_t`. The type and function have names like `__time32_t`, `__time32`, etc. This variant is mainly available for backwards compatibility.
- The 64-bit interface supports years from -9999 up to 9999 and uses a signed `long long` for `time_t`. The type and function have names like `__time64_t`, `__time64`, etc.

The interfaces are defined in the system header file `time.h`.

An application can use either interface, and even mix them by explicitly using the 32- or 64-bit variants. By default, the library and the header redirect `time_t`, `time` etc. to

the 32-bit variants. However, to explicitly redirect them to their 64-bit variants, define `_DLIB_TIME_USES_64` in front of the inclusion of `time.h` or `ctime`.

See also, *Time*, page 164.

## SYMBOLS USED INTERNALLY BY THE LIBRARY

The following symbols are used by the library, which means that they are visible in library source files, etc:

`__assignment_by_bitwise_copy_allowed`

This symbol determines properties for class objects.

`__constrange()`

Determines the allowed range for a parameter to an intrinsic function and that the parameter must be of type `const`.

`__construction_by_bitwise_copy_allowed`

This symbol determines properties for class objects.

`__has_constructor`, `__has_destructor`

These symbols determine properties for class objects and they function like the `sizeof` operator. The symbols are true when a class, base class, or member (recursively) has a user-defined constructor or destructor, respectively.

`__memory_of`

Determines the class memory. A class memory determines which memory a class object can reside in. This symbol can only occur in class definitions as a class memory.

**Note:** The symbols are reserved and should only be used by the library.

Use the compiler option `--predef_macros` to determine the value for any predefined symbols.

---

## IAR CLIB Library

The IAR CLIB Library provides most of the important C library definitions that apply to embedded systems. These are of the following types:

- Standard C library definitions available for user programs. These are documented in this chapter.

- The system startup code; see the chapter *The CLIB runtime environment* in this guide.
- Runtime support libraries; for example low-level floating-point routines.
- Intrinsic functions, allowing low-level use of AVR features. See the chapter *Intrinsic functions* for more information.
- Special compiler support for accessing strings in flash memory, see *AVR-specific library functions*, page 310.

**LIBRARY DEFINITIONS SUMMARY**

This table lists the header files specific to the CLIB library:

| Header file            | Description                                                                                                               |
|------------------------|---------------------------------------------------------------------------------------------------------------------------|
| <code>assert.h</code>  | Assertions                                                                                                                |
| <code>ctype.h*</code>  | Character handling                                                                                                        |
| <code>errno.h</code>   | Error return values                                                                                                       |
| <code>float.h</code>   | Limits and sizes of floating-point types                                                                                  |
| <code>iccbutl.h</code> | Low-level routines                                                                                                        |
| <code>limits.h</code>  | Limits and sizes of integral types                                                                                        |
| <code>math.h</code>    | Mathematics                                                                                                               |
| <code>setjmp.h</code>  | Non-local jumps                                                                                                           |
| <code>stdarg.h</code>  | Variable arguments                                                                                                        |
| <code>stdbool.h</code> | Adds support for the <code>bool</code> data type in C                                                                     |
| <code>stddef.h</code>  | Common definitions including <code>size_t</code> , <code>NULL</code> , <code>ptrdiff_t</code> , and <code>offsetof</code> |
| <code>stdio.h</code>   | Input/output                                                                                                              |
| <code>stdlib.h</code>  | General utilities                                                                                                         |
| <code>string.h</code>  | String handling                                                                                                           |

*Table 51: IAR CLIB Library header files*

**\* The functions `isxxx`, `toupper`, and `tolower` declared in the header file `ctype.h` evaluate their argument more than once. This is not according to the ISO/ANSI standard.**

**AVR-specific library functions**

This section lists the AVR-specific library functions declared in `pgmspace.h` that allow access to strings in flash memory. The `_P` functions exist in both the IAR CLIB Library and the IAR DLIB Library, but they allow access to strings in flash memory only. The

`_G` functions use the `__generic` pointer instead, which means that they allow access to both flash and data memory.

## SPECIFYING READ AND WRITE FORMATTERS

You can override default formatters for the functions `printf_P` and `scanf_P` by editing the linker command file. Note that it is not possible to use the IAR Embedded Workbench IDE for overriding the default formatter for the AVR-specific library routines.

To override the default `printf_P` formatter, type any of the following lines in your linker command file:

```
-e_small_write_P=_formatted_write_P
-e_medium_write_P=_formatted_write_P
```

To override the default `scanf_P` formatter, type the following line in your linker command file:

```
-e_medium_read_P=_formatted_read_P
```

## memcpy\_G

Syntax

```
int memcpy_G(const void *s1, const void __generic *s2, size_t
n);
```

Description

Identical to `memcpy` except that `s2` can be located in flash *or* data memory.

## memcpy\_G

Syntax

```
void * memcpy_G(void *s1, const void __generic *s2, size_t n);
```

Description

Identical to `memcpy` except that it copies string `s2` in flash *or* data memory to the location that `s1` points to in data memory.

## memcpy\_P

Syntax

```
void * memcpy_P(void *s1, PGM_P s2, size_t n);
```

Description

Identical to `memcpy` except that it copies string `s2` in flash memory to the location that `s1` points to in data memory. This function is available in both the CLIB and the DLIB library.

## printf\_P

|             |                                                                                                                                                                                                                                                                                             |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>int printf_P(PGM_P __format,...);</code>                                                                                                                                                                                                                                              |
| Description | Similar to <code>printf</code> except that the format string is in flash memory, not in data memory. This function is available in both the CLIB and the DLIB library. For information about how to override default formatter, see <i>Specifying read and write formatters</i> , page 311. |

## puts\_G

|             |                                                                                                            |
|-------------|------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>int puts_G(const char __generic *s);</code>                                                          |
| Description | Identical to <code>puts</code> except that the string to be written can be in flash <i>or</i> data memory. |

## puts\_P

|             |                                                                                                                                                                               |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>int puts_P(PGM_P __s);</code>                                                                                                                                           |
| Description | Identical to <code>puts</code> except that the string to be written is in flash memory, not in data memory. This function is available in both the CLIB and the DLIB library. |

## scanf\_P

|             |                                                                                                                                                                                                                                                                                                  |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>int scanf_P(PGM_P __format,...);</code>                                                                                                                                                                                                                                                    |
| Description | Identical to <code>scanf</code> except that the format string is in flash memory, not in data memory. This function is available in both the CLIB and the DLIB library. For information about how to override the default formatter, see <i>Specifying read and write formatters</i> , page 311. |

## sprintf\_P

|             |                                                                                                                                                                           |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>int sprintf_P(char *__s, PGM_P __format,...);</code>                                                                                                                |
| Description | Identical to <code>sprintf</code> except that the format string is in flash memory, not in data memory. This function is available in both the CLIB and the DLIB library. |



**sscanf\_P**

|             |                                                                                                                                                                          |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>int sscanf_P(const char *__s, PGM_P __format,...);</code>                                                                                                          |
| Description | Identical to <code>sscanf</code> except that the format string is in flash memory, not in data memory. This function is available in both the CLIB and the DLIB library. |

**strcat\_G**

|             |                                                                                                            |
|-------------|------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>char *strcat_G(char *s1, const char __generic *s2);</code>                                           |
| Description | Identical to <code>strcat</code> except that string <code>s2</code> can be in flash <i>or</i> data memory. |

**strcmp\_G**

|             |                                                                                                            |
|-------------|------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>int strcmp_G(const char *s1, const char __generic *s2);</code>                                       |
| Description | Identical to <code>strcmp</code> except that string <code>s2</code> can be in flash <i>or</i> data memory. |

**strcmp\_P**

|             |                                                                                                                                                                               |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>int strcmp_P(const char *s1, PGM_P s2);</code>                                                                                                                          |
| Description | Identical to <code>strcmp</code> except that string <code>s2</code> is in flash memory, not in data memory. This function is available in both the CLIB and the DLIB library. |

**strcpy\_G**

|             |                                                                                                                             |
|-------------|-----------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>char *strcpy_G(char *s1, const char __generic *s2);</code>                                                            |
| Description | Identical to <code>strcpy</code> except that the string <code>s2</code> being copied can be in flash <i>or</i> data memory. |

**strcpy\_P**

|             |                                                                                                                                                                                                |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>char * strcpy_P(char *s1, PGM_P s2);</code>                                                                                                                                              |
| Description | Identical to <code>strcpy</code> except that the string <code>s2</code> being copied is in flash memory, not in data memory. This function is available in both the CLIB and the DLIB library. |

**strerror\_P**

Syntax

`PGM_P strerror_P(int errnum);`

Description

Identical to `strerror` except that the string returned is in flash memory, not in data memory. This function is available in both the CLIB and the DLIB library.

**strlen\_G**

Syntax

`size_t strlen_G(const char __generic *s);`

Description

Identical to `strlen` except that the string being tested can be in flash *or* data memory.

**strlen\_P**

Syntax

`size_t strlen_P(PGM_P s);`

Description

Identical to `strlen` except that the string being tested is in flash memory, not in data memory. This function is available in both the CLIB and the DLIB library.

**strncat\_G**

Syntax

`char *strncat_G(char *s1, const char __generic *s2, size_t n);`

Description

Identical to `strncat` except that the string `s2` can be in flash *or* data memory.

**strncmp\_G**

Syntax

`int strncmp_G(const char *s1, const char __generic *s2, size_t n);`

Description

Identical to `strncmp` except that the string `s2` can be in flash *or* data memory.

**strncmp\_P**

Syntax

`int strncmp_P(const char *s1, PGM_P s2, size_t n);`

Description

Identical to `strncmp` except that the string `s2` is in flash memory, not in data memory. This function is available in both the CLIB and the DLIB library.

## strncpy\_G

|             |                                                                                                                        |
|-------------|------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>char *strncpy_G(char *s1, const char __generic *s2, size_t n);</code>                                            |
| Description | Identical to <code>strncpy</code> except that the source string <code>s2</code> can be in flash <i>or</i> data memory. |

## strncpy\_P

|             |                                                                                                                                                                                           |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <code>char * strncpy_P(char *s1, PGM_P s2, size_t n);</code>                                                                                                                              |
| Description | Identical to <code>strncpy</code> except that the source string <code>s2</code> is in flash memory, not in data memory. This function is available in both the CLIB and the DLIB library. |



# Segment reference

The compiler places code and data into named segments which are referred to by the IAR XLINK Linker. Details about the segments are required for programming assembler language modules, and are also useful when interpreting the assembler language output from the compiler.

For more information about segments, see the chapter *Placing code and data*.

---

## Summary of segments

The table below lists the segments that are available in the compiler:

| Segment  | Description                                                                                                                   |
|----------|-------------------------------------------------------------------------------------------------------------------------------|
| CHECKSUM | Holds the checksum generated by the linker.                                                                                   |
| CODE     | Holds the program code.                                                                                                       |
| CSTACK   | Holds the stack used by C or C++ programs.                                                                                    |
| DIFUNCT  | Holds pointers to code, typically C++ constructors, that should be executed by the system startup code before main is called. |
| EEPROM_I | Holds <code>__eeprom</code> static and global initialized variables.                                                          |
| EEPROM_N | Holds <code>__no_init __eeprom</code> static and global variables.                                                            |
| FARCODE  | Holds program code declared <code>__farfunc</code> .                                                                          |
| FAR_C    | Holds <code>__far</code> constant data.                                                                                       |
| FAR_F    | Holds static and global <code>__farflash</code> variables.                                                                    |
| FAR_HEAP | Holds the heap used for dynamically allocated data in far memory using the DLIB library.                                      |
| FAR_I    | Holds <code>__far</code> static and global initialized variables.                                                             |
| FAR_ID   | Holds initial values for <code>__far</code> static and global variables in FAR_I.                                             |
| FAR_N    | Holds <code>__no_init __far</code> static and global variables.                                                               |
| FAR_Z    | Holds zero-initialized <code>__far</code> static and global variables.                                                        |
| HEAP     | Holds the heap used for dynamically allocated data using the CLIB library.                                                    |
| HUGE_C   | Holds <code>__huge</code> constant data.                                                                                      |
| HUGE_F   | Holds static and global <code>__hugeflash</code> variables.                                                                   |

Table 52: Segment summary

| Segment   | Description                                                                                                                  |
|-----------|------------------------------------------------------------------------------------------------------------------------------|
| HUGE_HEAP | Holds the heap used for dynamically allocated data in huge memory using the DLIB library.                                    |
| HUGE_I    | Holds <code>__huge</code> static and global initialized variables.                                                           |
| HUGE_ID   | Holds initial values for <code>__huge</code> static and global variables in <code>HUGE_I</code> .                            |
| HUGE_N    | Holds <code>__no_init __huge</code> static and global variables.                                                             |
| HUGE_Z    | Holds zero-initialized <code>__huge</code> static and global variables.                                                      |
| INITTAB   | Contains compiler-generated table entries that describe the segment initialization that will be performed at system startup. |
| INTVEC    | Contains the reset and interrupt vectors.                                                                                    |
| NEAR_C    | Holds <code>__tiny</code> and <code>__near</code> constant data.                                                             |
| NEAR_F    | Holds static and global <code>__flash</code> variables.                                                                      |
| NEAR_HEAP | Holds the heap used for dynamically allocated data in near memory using the DLIB library.                                    |
| NEAR_I    | Holds <code>__near</code> static and global initialized variables.                                                           |
| NEAR_ID   | Holds initial values for <code>__near</code> static and global variables in <code>NEAR_I</code> .                            |
| NEAR_N    | Holds <code>__no_init __near</code> static and global variables.                                                             |
| NEAR_Z    | Holds zero-initialized <code>__near</code> static and global variables.                                                      |
| RSTACK    | Holds the internal return stack.                                                                                             |
| SWITCH    | Holds switch tables for all functions.                                                                                       |
| TINY_F    | Holds static and global <code>__tinyflash</code> variables.                                                                  |
| TINY_HEAP | Holds the heap used for dynamically allocated data in tiny memory using the DLIB library.                                    |
| TINY_I    | Holds <code>__tiny</code> static and global initialized variables.                                                           |
| TINY_ID   | Holds initial values for <code>__tiny</code> static and global variables in <code>TINY_I</code> .                            |
| TINY_N    | Holds <code>__no_init __tiny</code> static and global variables.                                                             |
| TINY_Z    | Holds zero-initialized <code>__tiny</code> static and global variables.                                                      |

Table 52: Segment summary (Continued)

## Descriptions of segments

This section gives reference information about each segment.

The segments are placed in memory by the segment placement linker directives `-Z` and `-P`, for sequential and packed placement, respectively. Some segments cannot use packed placement, as their contents must be continuous.

In each description, the segment memory type—CODE, DATA, or XDATA—indicates whether the segment should be placed in ROM, RAM, or EEPROM memory; see Table 7, *XLINK segment memory types*, page 54.

For information about the `-Z` and the `-P` directives, see the *IAR Linker and Library Tools Reference Guide*.

For information about how to define segments in the linker configuration file, see *Customizing the linker configuration file*, page 54.

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

## CHECKSUM

|                     |                                                                                                                                                                                                     |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds the checksum bytes generated by the linker. This segment also holds the <code>__checksum</code> symbol. Note that the size of this segment is affected by the linker option <code>-J</code> . |
| Segment memory type | CODE                                                                                                                                                                                                |
| Memory placement    | This segment can be placed anywhere in ROM memory.                                                                                                                                                  |
| Access type         | Read-only                                                                                                                                                                                           |

## CODE

|                     |                                                                                         |
|---------------------|-----------------------------------------------------------------------------------------|
| Description         | Holds <code>__nearfunc</code> program code, except the code for system initialization.  |
| Segment memory type | CODE                                                                                    |
| Memory placement    | Flash. This segment must be placed within the address range <code>0x0–0x01FFFE</code> . |
| Access type         | Read-only                                                                               |

## CSTACK

|                     |                                |
|---------------------|--------------------------------|
| Description         | Holds the internal data stack. |
| Segment memory type | DATA                           |

|                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Memory placement | <p>Data. The address range depends on the memory model:</p> <p>In the Tiny memory model, this segment must be placed within the address range 0x0-0xFF.</p> <p>In the Small memory model, this segment must be placed within the address range 0x0-0xFFFF.</p> <p>In the Large and Huge memory models, this segment must be placed within the address range 0x0-0xFFFFFFFF. In the Large and Huge memory models, the stack can be a maximum of 64 Kbytes, and CSTACK must not cross a 64-Kbyte boundary.</p> |
| Access type      | Read/write                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| See also         | <i>The stack</i> , page 61.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |

DIFUNCT

|                     |                                                                                                                                                             |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds the dynamic initialization vector used by C++.                                                                                                        |
| Segment memory type | CODE                                                                                                                                                        |
| Memory placement    | In the small data model, this segment must be placed in the first 64 Kbytes of memory. In other data models, this segment can be placed anywhere in memory. |
| Access type         | Read-only                                                                                                                                                   |

EEPROM\_I

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds <code>__eeprom</code> static and global initialized variables initialized by copying from the segment <code>EEPROM_ID</code> at application startup.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used.</p> <p>This segment is not copied to EEPROM during system startup. Instead it is used for programming the EEPROM during the download of the code.</p> |
| Segment memory type | XDATA                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| Memory placement    | This segment must be placed in EEPROM. Use the command line option <code>--eeprom_size</code> to set the address range for this segment.                                                                                                                                                                                                                                                                                                                                                                                                                                                     |



|             |                                       |
|-------------|---------------------------------------|
| Access type | Read-only                             |
| See also    | <code>--eeprom_size</code> , page 204 |

## EEPROM\_N

|                     |                                                                                                                                          |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds static and global <code>__no_init __eeprom</code> variables.                                                                       |
| Segment memory type | XDATA                                                                                                                                    |
| Memory placement    | This segment must be placed in EEPROM. Use the command line option <code>--eeprom_size</code> to set the address range for this segment. |
| Access type         | Read-only                                                                                                                                |
| See also            | <code>--eeprom_size</code> , page 204                                                                                                    |

## FARCODE

|                     |                                                                                                                                                                                                                                                  |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds <code>__farfunc</code> program code. The <code>__farfunc</code> memory attribute is available when using the <code>-v5</code> and <code>-v6</code> options, in which case the <code>__farfunc</code> is implicitly used for all functions. |
| Segment memory type | CODE                                                                                                                                                                                                                                             |
| Memory placement    | This segment must be placed within the address range <code>0x0-0x7FFFFE</code> in flash memory.                                                                                                                                                  |
| Access type         | Read-only                                                                                                                                                                                                                                        |

## FAR\_C

|                     |                                                                                                                                                                                                                                            |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds <code>__far</code> constant data. This can include constant variables, string and aggregate literals, etc.</p> <p><b>Note:</b> This segment is located in external ROM. Systems without external ROM cannot use this segment.</p> |
| Segment memory type | DATA                                                                                                                                                                                                                                       |
| Memory placement    | External ROM. This segment must be placed within the address range <code>0x0-0xFFFFF</code> .                                                                                                                                              |

Access type                      Read-only

**FAR\_F**

Description                      Holds the static and global `__farflash` variables and aggregate initializers.

Segment memory type              CODE

Memory placement              Flash. This segment must be placed within the address range `0x0-0x7FFFFFFF`.

Access type                      Read-only

**FAR\_HEAP**

Description                      Holds the heap used for dynamically allocated data in far memory, in other words data allocated by `far_malloc` and `far_free`, and in C++, `new` and `delete`.

**Note:** This segment is only used when you use the DLIB library.

Segment memory type              DATA

Memory placement              Data. This segment must be placed within the address range `0x0-0xFFFFFFFF`.

Access type                      Read/write

See also                          *The heap*, page 64 and *New and Delete operators*, page 149.

**FAR\_I**

Description                      Holds `__far` static and global initialized variables initialized by copying from the segment `FAR_ID` at application startup.

                                      This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the `-Z` directive must be used.

Segment memory type              DATA

Memory placement              Data. This segment must be placed within the address range `0x0-0xFFFFFFFF`.

Access type                      Read/write

## FAR\_ID

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds initial values for <code>__far</code> static and global variables in the <code>FAR_I</code> segment. These values are copied from <code>FAR_ID</code> to <code>FAR_I</code> at application startup.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used.</p> |
| Segment memory type | CODE                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| Memory placement    | Flash. This segment must be placed within the address range <code>0x0-0x7FFFFFFF</code> .                                                                                                                                                                                                                                                                                                                                                                                                 |
| Access type         | Read-only                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |

## FAR\_N

|                     |                                                                                          |
|---------------------|------------------------------------------------------------------------------------------|
| Description         | Holds static and global <code>__no_init __far</code> variables.                          |
| Segment memory type | DATA                                                                                     |
| Memory placement    | Data. This segment must be placed within the address range <code>0x0-0xFFFFFFFF</code> . |
| Access type         | Read/write                                                                               |

## FAR\_Z

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                             |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds zero-initialized <code>__far</code> static and global variables. The contents of this segment is declared by the system startup code.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used.</p> |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                                                                                        |
| Memory placement    | Data. This segment must be placed within the address range <code>0x0-0xFFFFFFFF</code> .                                                                                                                                                                                                                                                                                                                                    |
| Access type         | Read/write                                                                                                                                                                                                                                                                                                                                                                                                                  |

HEAP

|                     |                                                                                                                                                                                                                                                                                                                                                                                 |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds the heap used for dynamically allocated data, in other words data allocated by <code>malloc</code> and <code>free</code>, and in C++, <code>new</code> and <code>delete</code>.</p> <p><b>Note:</b> This segment is only used when you use the CLIB library.</p>                                                                                                       |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                                            |
| Memory placement    | <p>Data. The address range depends on the memory model.</p> <p>In the Tiny memory model, this segment must be placed within the address range 0x0-0xFF.</p> <p>In the Small memory model, this segment must be placed within the address range 0x0-0xFFFF.</p> <p>In the Large and Huge memory models, this segment must be placed within the address range 0x0-0xFFFFFFFF.</p> |
| Access type         | Read/write                                                                                                                                                                                                                                                                                                                                                                      |
| See also            | <i>The heap</i> , page 64.                                                                                                                                                                                                                                                                                                                                                      |

HUGE\_C

|                     |                                                                                                                                                                                                                                             |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds <code>__huge</code> constant data. This can include constant variables, string and aggregate literals, etc.</p> <p><b>Note:</b> This segment is located in external ROM. Systems without external ROM cannot use this segment.</p> |
| Segment memory type | DATA                                                                                                                                                                                                                                        |
| Memory placement    | External ROM. This segment must be placed within the address range 0x0-0xFFFFFFFF.                                                                                                                                                          |
| Access type         | Read-only                                                                                                                                                                                                                                   |

HUGE\_F

|                     |                                                                                            |
|---------------------|--------------------------------------------------------------------------------------------|
| Description         | Holds the static and global <code>__hugeflash</code> variables and aggregate initializers. |
| Segment memory type | CODE                                                                                       |

|                  |                                                                             |
|------------------|-----------------------------------------------------------------------------|
| Memory placement | Flash. This segment must be placed within the address range 0x0-0xFFFFFFFF. |
| Access type      | Read-only                                                                   |

## HUGE\_HEAP

|                     |                                                                                                                                                                                                                                                                                                    |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds the heap used for dynamically allocated data in huge memory, in other words data allocated by <code>huge_malloc</code> and <code>huge_free</code>, and in C++, <code>new</code> and <code>delete</code>.</p> <p><b>Note:</b> This segment is only used when you use the DLIB library.</p> |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                               |
| Memory placement    | Data. This segment must be placed within the address range 0x0-0xFFFFFFFF.                                                                                                                                                                                                                         |
| Access type         | Read/write                                                                                                                                                                                                                                                                                         |
| See also            | <i>The heap</i> , page 64 and <i>New and Delete operators</i> , page 149.                                                                                                                                                                                                                          |

## HUGE\_I

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds <code>__huge</code> static and global initialized variables initialized by copying from the segment <code>HUGE_ID</code> at application startup.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used.</p> <p>When the <code>-y</code> compiler option is used, <code>__huge</code> constant data is located in this segment.</p> |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| Memory placement    | Data. This segment must be placed within the address range 0x0-0xFFFFFFFF.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| Access type         | Read/write                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |

## HUGE\_ID

|             |                                                                                                                                                                                                                      |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | <p>Holds initial values for <code>__huge</code> static and global variables in the <code>HUGE_I</code> segment. These values are copied from <code>HUGE_ID</code> to <code>HUGE_I</code> at application startup.</p> |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the `-Z` directive must be used.

|                     |                                                                                        |
|---------------------|----------------------------------------------------------------------------------------|
| Segment memory type | CODE                                                                                   |
| Memory placement    | Flash. This segment must be placed within the address range <code>0x0-0xFFFFF</code> . |
| Access type         | Read-only                                                                              |

**HUGE\_N**

|                     |                                                                                       |
|---------------------|---------------------------------------------------------------------------------------|
| Description         | Holds static and global <code>__no_init __huge</code> variables.                      |
| Segment memory type | DATA                                                                                  |
| Memory placement    | Data. This segment must be placed within the address range <code>0x0-0xFFFFF</code> . |
| Access type         | Read/write                                                                            |

**HUGE\_Z**

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                              |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds zero-initialized <code>__huge</code> static and global variables. The contents of this segment is declared by the system startup code.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used.</p> |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                                                                                         |
| Memory placement    | Data. This segment must be placed within the address range <code>0x0-0xFFFFF</code> .                                                                                                                                                                                                                                                                                                                                        |
| Access type         | Read/write                                                                                                                                                                                                                                                                                                                                                                                                                   |

**INITTAB**

|                     |                                                                                                                            |
|---------------------|----------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds compiler-generated table entries that describe the segment initialization which will be performed at system startup. |
| Segment memory type | CODE                                                                                                                       |

|                  |                                                                                                          |
|------------------|----------------------------------------------------------------------------------------------------------|
| Memory placement | Flash. This segment must be placed within the address range 0x0-0xFFFF (0x7FFFF if farflash is enabled). |
| Access type      | Read-only                                                                                                |

## INTVEC

|                     |                                                                                                                                                                       |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds the interrupt vector table generated by the use of the <code>__interrupt</code> extended keyword in combination with the <code>#pragma vector</code> directive. |
| Segment memory type | CODE                                                                                                                                                                  |
| Memory placement    | This segment can be placed anywhere in memory.                                                                                                                        |
| Access type         | Read-only                                                                                                                                                             |

## NEAR\_C

|                     |                                                                                                                                                                                                                                                                                                                                                                             |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds <code>__tiny</code> and <code>__near</code> constant data. This can include constant variables, string and aggregate literals, etc. <code>TINY_I</code> is copied from this segment, because <code>TINY_C</code> is not a possible segment.</p> <p><b>Note:</b> This segment is located in external ROM. Systems without external ROM cannot use this segment.</p> |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                                        |
| Memory placement    | External ROM. This segment must be placed within the address range 0x0-0xFFFF.                                                                                                                                                                                                                                                                                              |
| Access type         | Read-only                                                                                                                                                                                                                                                                                                                                                                   |

## NEAR\_F

|                     |                                                                                        |
|---------------------|----------------------------------------------------------------------------------------|
| Description         | Holds the static and global <code>__flash</code> variables and aggregate initializers. |
| Segment memory type | CODE                                                                                   |
| Memory placement    | Flash. This segment must be placed within the address range 0x0-0xFFFF.                |
| Access type         | Read-only                                                                              |

## NEAR\_HEAP

|                     |                                                                                                                                                                                                                                                                                               |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds the heap used for dynamically allocated data in near memory, in other words data allocated by <code>near_malloc</code> and <code>near_free</code> , and in C++, <code>new</code> and <code>delete</code> .<br><br><b>Note:</b> This segment is only used when you use the DLIB library. |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                          |
| Memory placement    | Data. This segment must be placed within the address range <code>0x0-0xFFFF</code> .                                                                                                                                                                                                          |
| Access type         | Read/write                                                                                                                                                                                                                                                                                    |
| See also            | <i>The heap</i> , page 64 and <i>New and Delete operators</i> , page 149.                                                                                                                                                                                                                     |

## NEAR\_I

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds <code>__near</code> static and global initialized variables initialized by copying from the segment <code>NEAR_ID</code> at application startup.<br><br>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used.<br><br>When the <code>-Y</code> compiler option is used, <code>NEAR_C</code> data ( <code>__near</code> ) is located in this segment. |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| Memory placement    | Data. This segment must be placed within the address range <code>0x0-0xFFFF</code> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| Access type         | Read/write                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |

## NEAR\_ID

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds initial values for <code>__near</code> static and global variables in the <code>NEAR_I</code> segment. These values are copied from <code>NEAR_ID</code> to <code>NEAR_I</code> at application startup.<br><br>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used. |
| Segment memory type | CODE                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| Memory placement    | Flash. This segment must be placed within the address range <code>0x0-0x7FFFFFFF</code> .                                                                                                                                                                                                                                                                                                                                                                                              |



|             |           |
|-------------|-----------|
| Access type | Read-only |
|-------------|-----------|

## NEAR\_N

|                     |                                                                                      |
|---------------------|--------------------------------------------------------------------------------------|
| Description         | Holds static and global <code>__no_init __near</code> variables.                     |
| Segment memory type | DATA                                                                                 |
| Memory placement    | Data. This segment must be placed within the address range <code>0x0-0xFFFF</code> . |
| Access type         | Read/write                                                                           |

## NEAR\_Z

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                              |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds zero-initialized <code>__near</code> static and global variables. The contents of this segment is declared by the system startup code.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used.</p> |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                                                                                         |
| Memory placement    | Data. This segment must be placed within the address range <code>0x0-0xFFFF</code> .                                                                                                                                                                                                                                                                                                                                         |
| Access type         | Read/write                                                                                                                                                                                                                                                                                                                                                                                                                   |

## RSTACK

|                     |                                                                                      |
|---------------------|--------------------------------------------------------------------------------------|
| Description         | Holds the internal return stack.                                                     |
| Segment memory type | DATA                                                                                 |
| Memory placement    | Data. This segment must be placed within the address range <code>0x0-0xFFFF</code> . |
| Access type         | Read/write                                                                           |

## SWITCH

|             |                                        |
|-------------|----------------------------------------|
| Description | Holds switch tables for all functions. |
|-------------|----------------------------------------|

The `SWITCH` segment is for compiler internal use only and should always be defined. The segment allocates, if necessary, jump tables for C/C++ switch statements.

|                     |                                                                                                                                                                                                                                                                                            |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Segment memory type | CODE                                                                                                                                                                                                                                                                                       |
| Memory placement    | Flash. This segment must be placed within the address range 0x0-0xFFFF. If the <code>__farflash</code> extended keyword and the <code>--enhanced_core</code> option are used, the segment must be placed within the range 0x0-0x7FFFFFFF. This segment must not cross a 64-Kbyte boundary. |
| Access type         | Read-only                                                                                                                                                                                                                                                                                  |

TINY\_F

|                     |                                                                                            |
|---------------------|--------------------------------------------------------------------------------------------|
| Description         | Holds the static and global <code>__tinyflash</code> variables and aggregate initializers. |
| Segment memory type | CODE                                                                                       |
| Memory placement    | Flash. This segment must be placed within the address range 0x0-0xFF.                      |
| Access type         | Read-only                                                                                  |

TINY\_HEAP

|                     |                                                                                                                                                                                                                                                                                           |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | Holds the heap used for dynamically allocated data in tiny memory, in other words data allocated by <code>tiny_malloc</code> and <code>tiny_free</code> , and in C++, <code>new</code> and <code>delete</code> .<br><b>Note:</b> This segment is only used when you use the DLIB library. |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                      |
| Memory placement    | Data. This segment must be placed within the address range 0x0-0xFF.                                                                                                                                                                                                                      |
| Access type         | Read/write                                                                                                                                                                                                                                                                                |
| See also            | <i>The heap</i> , page 64 and <i>New and Delete operators</i> , page 149.                                                                                                                                                                                                                 |

TINY\_I

|             |                                                                                                                                                        |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description | Holds <code>__tiny</code> static and global initialized variables initialized by copying from the segment <code>TINY_ID</code> at application startup. |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|

|                     |                                                                                                                                                                                                                                                                                                                                                                                            |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                     | <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used.</p> <p>When the <code>-y</code> compiler option is not used, <code>NEAR_C</code> data is located in this segment.</p> |
| Segment memory type | DATA                                                                                                                                                                                                                                                                                                                                                                                       |
| Memory placement    | Data. This segment must be placed within the address range <code>0x0-0xFF</code> .                                                                                                                                                                                                                                                                                                         |
| Access type         | Read/write                                                                                                                                                                                                                                                                                                                                                                                 |

## TINY\_ID

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description         | <p>Holds initial values for <code>__tiny</code> static and global variables in the <code>TINY_I</code> segment. These values are copied from <code>TINY_ID</code> to <code>TINY_I</code> at application startup.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the <code>-Z</code> directive must be used.</p> |
| Segment memory type | CODE                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| Memory placement    | Flash. This segment must be placed within the address range <code>0x0-0x7FFFFFFF</code> .                                                                                                                                                                                                                                                                                                                                                                                                     |
| Access type         | Read-only                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |

## TINY\_N

|                     |                                                                                    |
|---------------------|------------------------------------------------------------------------------------|
| Description         | Holds static and global <code>__no_init __tiny</code> variables.                   |
| Segment memory type | DATA                                                                               |
| Memory placement    | Data. This segment must be placed within the address range <code>0x0-0xFF</code> . |
| Access type         | Read/write                                                                         |

## TINY\_Z

|             |                                                                                                                                              |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------|
| Description | Holds zero-initialized <code>__tiny</code> static and global variables. The contents of this segment is declared by the system startup code. |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------|

This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the `-Z` directive must be used.

|                     |                                                                                    |
|---------------------|------------------------------------------------------------------------------------|
| Segment memory type | DATA                                                                               |
| Memory placement    | Data. This segment must be placed within the address range <code>0x0-0xFF</code> . |
| Access type         | Read/write                                                                         |

# Implementation-defined behavior

This chapter describes how IAR Systems handles the implementation-defined areas of the C language.

Note: The IAR Systems implementation adheres to a freestanding implementation of Standard C. This means that parts of a standard library can be excluded in the implementation.

The text in this chapter applies to the DLIB library. Because the CLIB library does not follow Standard C, its implementation-defined behavior is not documented. For information about the CLIB library, see *The CLIB runtime environment*, page 109.

---

## Descriptions of implementation-defined behavior

This section follows the same order as the C standard. Each item includes references to the ISO chapter and section (in parenthesis) that explains the implementation-defined behavior.

### J.3.1 Translation

#### Diagnostics (3.10, 5.1.1.3)

Diagnostics are produced in the form:

*filename*, *linenumber* *level*[*tag*]: *message*

where *filename* is the name of the source file in which the error was encountered, *linenumber* is the line number at which the compiler detected the error, *level* is the level of seriousness of the message (remark, warning, error, or fatal error), *tag* is a unique tag that identifies the message, and *message* is an explanatory message, possibly several lines.

#### White-space characters (5.1.1.2)

At translation phase three, each non-empty sequence of white-space characters is retained.

## J.3.2 Environment

### The character set (5.1.1.2)

The source character set is the same as the physical source file multibyte character set. By default, the standard ASCII character set is used. However, if you use the `--enable_multibytes` compiler option, the host character set is used instead.

### Main (5.1.2.1)

The function called at program startup is called `main`. No prototype is declared for `main`, and the only definition supported for `main` is:

```
int main(void)
```

To change this behavior for the IAR DLIB runtime environment, see *Customizing system initialization*, page 87.

### The effect of program termination (5.1.2.1)

Terminating the application returns the execution to the startup code (just after the call to `main`).

### Alternative ways to define main (5.1.2.2.1)

There is no alternative ways to define the `main` function.

### The argv argument to main (5.1.2.2.1)

The `argv` argument is not supported.

### Streams as interactive devices (5.1.2.3)

The streams `stdin`, `stdout`, and `stderr` are treated as interactive devices.

### Signals, their semantics, and the default handling (7.14)

In the DLIB library, the set of supported signals is the same as in Standard C. A raised signal will do nothing, unless the `signal` function is customized to fit the application.

### Signal values for computational exceptions (7.14.1.1)

In the DLIB library, there are no implementation-defined values that correspond to a computational exception.

**Signals at system startup (7.14.1.1)**

In the DLIB library, there are no implementation-defined signals that are executed at system startup.

**Environment names (7.20.4.5)**

In the DLIB library, there are no implementation-defined environment names that are used by the `getenv` function.

**The system function (7.20.4.6)**

The `system` function is not supported.

**J.3.3 Identifiers****Multibyte characters in identifiers (6.4.2)**

Additional multibyte characters may not appear in identifiers.

**Significant characters in identifiers (5.2.4.1, 6.1.2)**

The number of significant initial characters in an identifier with or without external linkage is guaranteed to be no less than 200.

**J.3.4 Characters****Number of bits in a byte (3.6)**

A byte contains 8 bits.

**Execution character set member values (5.2.1)**

The values of the members of the execution character set are the values of the ASCII character set, which can be augmented by the values of the extra characters in the host character set.

**Alphabetic escape sequences (5.2.2)**

The standard alphabetic escape sequences have the values `\a-7`, `\b-8`, `\f-12`, `\n-10`, `\r-13`, `\t-9`, and `\v-11`.

**Characters outside of the basic executive character set (6.2.5)**

A character outside of the basic executive character set that is stored in a `char` is not transformed.

**Plain char (6.2.5, 6.3.1.1)**

A plain `char` is treated as an unsigned `char`.

**Source and execution character sets (6.4.4.4, 5.1.1.2)**

The source character set is the set of legal characters that can appear in source files. By default, the source character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the source character set will be the host computer's default character set.

The execution character set is the set of legal characters that can appear in the execution environment. By default, the execution character set is the standard ASCII character set.

However, if you use the command line option `--enable_multibytes`, the execution character set will be the host computer's default character set. The IAR DLIB Library needs a multibyte character scanner to support a multibyte execution character set. See *Locale*, page 93.

**Integer character constants with more than one character (6.4.4.4)**

An integer character constant that contains more than one character will be treated as an integer constant. The value will be calculated by treating the leftmost character as the most significant character, and the rightmost character as the least significant character, in an integer constant. A diagnostic message will be issued if the value cannot be represented in an integer constant.

**Wide character constants with more than one character (6.4.4.4)**

A wide character constant that contains more than one multibyte character generates a diagnostic message.

**Locale used for wide character constants (6.4.4.4)**

By default, the C locale is used. If the `--enable_multibytes` compiler option is used, the default host locale is used instead.

**Locale used for wide string literals (6.4.5)**

By default, the C locale is used. If the `--enable_multibytes` compiler option is used, the default host locale is used instead.

**Source characters as executive characters (6.4.5)**

All source characters can be represented as executive characters.



## J.3.5 Integers

### Extended integer types (6.2.5)

There are no extended integer types.

### Range of integer values (6.2.6.2)

The representation of integer values are in the two's complement form. The most significant bit holds the sign; 1 for negative, 0 for positive and zero.

For information about the ranges for the different integer types, see *Basic data types*, page 232.

### The rank of extended integer types (6.3.1.1)

There are no extended integer types.

### Signals when converting to a signed integer type (6.3.1.3)

No signal is raised when an integer is converted to a signed integer type.

### Signed bitwise operations (6.5)

Bitwise operations on signed integers work the same way as bitwise operations on unsigned integers; in other words, the sign-bit will be treated as any other bit.

## J.3.6 Floating point

### Accuracy of floating-point operations (5.2.4.2.2)

The accuracy of floating-point operations is unknown.

### Rounding behaviors (5.2.4.2.2)

There are no non-standard values of `FLT_ROUNDS`.

### Evaluation methods (5.2.4.2.2)

There are no non-standard values of `FLT_EVAL_METHOD`.

### Converting integer values to floating-point values (6.3.1.4)

When an integral value is converted to a floating-point value that cannot exactly represent the source value, the round-to-nearest rounding mode is used (`FLT_ROUNDS` is defined to 1).

### **Converting floating-point values to floating-point values (6.3.1.5)**

When a floating-point value is converted to a floating-point value that cannot exactly represent the source value, the round-to-nearest rounding mode is used (`FLT_ROUNDS` is defined to 1).

### **Denoting the value of floating-point constants (6.4.4.2)**

The round-to-nearest rounding mode is used (`FLT_ROUNDS` is defined to 1).

### **Contraction of floating-point values (6.5)**

Floating-point values are contracted. However, there is no loss in precision and because signaling is not supported, this does not matter.

### **Default state of `FENV_ACCESS` (7.6.1)**

The default state of the pragma directive `FENV_ACCESS` is `OFF`.

### **Additional floating-point mechanisms (7.6, 7.12)**

There are no additional floating-point exceptions, rounding-modes, environments, and classifications.

### **Default state of `FP_CONTRACT` (7.12.2)**

The default state of the pragma directive `FP_CONTRACT` is `OFF`.

## **J.3.7 Arrays and pointers**

### **Conversion from/to pointers (6.3.2.3)**

For information about casting of data pointers and function pointers, see *Casting*, page 238.

### **`ptrdiff_t` (6.5.6)**

For information about `ptrdiff_t`, see *ptrdiff\_t*, page 239.

## **J.3.8 Hints**

### **Honoring the register keyword (6.7.1)**

User requests for register variables are not honored.

**Inlining functions (6.7.4)**

User requests for inlining functions increases the chance, but does not make it certain, that the function will actually be inlined into another function. See the pragma directive *inline*, page 272.

**J.3.9 Structures, unions, enumerations, and bitfields****Sign of 'plain' bitfields (6.7.2, 6.7.2.1)**

For information about how a 'plain' `int` bitfield is treated, see *Bitfields*, page 233.

**Possible types for bitfields (6.7.2.1)**

All integer types can be used as bitfields in the compiler's extended mode, see *-e*, page 203.

**Bitfields straddling a storage-unit boundary (6.7.2.1)**

A bitfield is always placed in one—and one only—storage unit, which means that the bitfield cannot straddle a storage-unit boundary.

**Allocation order of bitfields within a unit (6.7.2.1)**

For information about how bitfields are allocated within a storage unit, see *Bitfields*, page 233.

**Alignment of non-bitfield structure members (6.7.2.1)**

The alignment of non-bitfield members of structures is the same as for the member types, see *Alignment*, page 231.

**Integer type used for representing enumeration types (6.7.2.2)**

The chosen integer type for a specific enumeration type depends on the enumeration constants defined for the enumeration type. The chosen integer type is the smallest possible.

**J.3.10 Qualifiers****Access to volatile objects (6.7.3)**

Any reference to an object with `volatile` qualified type is an access, see *Declaring objects volatile*, page 240.

## J.3.11 Preprocessing directives

### Mapping of header names (6.4.7)

Sequences in header names are mapped to source file names verbatim. A backslash '\' is not treated as an escape sequence. See *Overview of the preprocessor*, page 293.

### Character constants in constant expressions (6.10.1)

A character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set.

### The value of a single-character constant (6.10.1)

A single-character constant may only have a negative value if a plain character (`char`) is treated as a signed character, see *--char\_is\_signed*, page 195.

### Including bracketed filenames (6.10.2)

For information about the search algorithm used for file specifications in angle brackets `<>`, see *Include file search procedure*, page 182.

### Including quoted filenames (6.10.2)

For information about the search algorithm used for file specifications enclosed in quotes, see *Include file search procedure*, page 182.

### Preprocessing tokens in #include directives (6.10.2)

Preprocessing tokens in an `#include` directive are combined in the same way as outside an `#include` directive.

### Nesting limits for #include directives (6.10.2)

There is no explicit nesting limit for `#include` processing.

### Universal character names (6.10.3.2)

Universal character names (UCN) are not supported.

### Recognized pragma directives (6.10.6)

In addition to the pragma directives described in the chapter *Pragma directives*, the following directives are recognized and will have an indeterminate effect. If a pragma directive is listed both in the *Pragma directives* chapter and here, the information provided in the *Pragma directives* chapter overrides the information here.

`alignment`

```

baseaddr
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
public_equ
system_include
warnings

```

### Default `__DATE__` and `__TIME__` (6.10.8)

The definitions for `__TIME__` and `__DATE__` are always available.

## J.3.12 Library functions

### Additional library facilities (5.1.2.1)

Most of the standard library facilities are supported. Some of them—the ones that need an operating system—require a low-level implementation in the application. For more information, see *The DLIB runtime environment*, page 71.

### Diagnostic printed by the `assert` function (7.2.1.1)

The `assert()` function prints:

```
filename:linenr expression -- assertion failed
```

when the parameter evaluates to zero.

### Representation of the floating-point status flags (7.6.2.2)

For information about the floating-point status flags, see *fenv.h*, page 307.

### Feraiseexcept raising floating-point exception (7.6.2.3)

For information about the `feraiseexcept` function raising floating-point exceptions, see *Floating-point environment*, page 235.

### Strings passed to the setlocale function (7.11.1.1)

For information about strings passed to the `setlocale` function, see *Locale*, page 93.

### Types defined for float\_t and double\_t (7.12)

The `FLT_EVAL_METHOD` macro can only have the value 0.

### Domain errors (7.12.1)

No function generates other domain errors than what the standard requires.

### Return values on domain errors (7.12.1)

Mathematic functions return a floating-point NaN (not a number) for domain errors.

### Underflow errors (7.12.1)

Mathematic functions set `errno` to the macro `ERANGE` (a macro in `errno.h`) and return zero for underflow errors.

### fmod return value (7.12.10.1)

The `fmod` function returns a floating-point NaN when the second argument is zero.

### The magnitude of remquo (7.12.10.3)

The magnitude is congruent modulo `INT_MAX`.

### signal() (7.14.1.1)

The signal part of the library is not supported.

**Note:** Low-level interface functions exist in the library, but will not perform anything. Use the template source code to implement application-specific signal handling. See *Signal and raise*, page 96.

**NULL macro (7.17)**

The `NULL` macro is defined to 0.

**Terminating newline character (7.19.2)**

`stdout` stream functions recognize either `newline` or end of file (EOF) as the terminating character for a line.

**Space characters before a newline character (7.19.2)**

Space characters written to a stream immediately before a newline character are preserved.

**Null characters appended to data written to binary streams (7.19.2)**

No null characters are appended to data written to binary streams.

**File position in append mode (7.19.3)**

The file position is initially placed at the beginning of the file when it is opened in append-mode.

**Truncation of files (7.19.3)**

Whether a write operation on a text stream causes the associated file to be truncated beyond that point, depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 92.

**File buffering (7.19.3)**

An open file can be either block-buffered, line-buffered, or unbuffered.

**A zero-length file (7.19.3)**

Whether a zero-length file exists depends on the application-specific implementation of the low-level file routines.

**Legal file names (7.19.3)**

The legality of a filename depends on the application-specific implementation of the low-level file routines.

**Number of times a file can be opened (7.19.3)**

Whether a file can be opened more than once depends on the application-specific implementation of the low-level file routines.

### **Multibyte characters in a file (7.19.3)**

The encoding of multibyte characters in a file depends on the application-specific implementation of the low-level file routines.

#### **remove() (7.19.4.1)**

The effect of a remove operation on an open file depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 92.

#### **rename() (7.19.4.2)**

The effect of renaming a file to an already existing filename depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 92.

### **Removal of open temporary files (7.19.4.3)**

Whether an open temporary file is removed depends on the application-specific implementation of the low-level file routines.

### **Mode changing (7.19.5.4)**

`freopen` closes the named stream, then reopens it in the new mode. The streams `stdin`, `stdout`, and `stderr` can be reopened in any new mode.

### **Style for printing infinity or NaN (7.19.6.1, 7.24.2.1)**

The style used for printing infinity or NaN for a floating-point constant is `inf` and `nan` (`INF` and `NAN` for the `F` conversion specifier), respectively. The `n`-char-sequence is not used for `nan`.

### **%p in printf() (7.19.6.1, 7.24.2.1)**

The argument to a `%p` conversion specifier, print pointer, to `printf()` is treated as having the type `void *`. The value will be printed as a hexadecimal number, similar to using the `%x` conversion specifier.

### **Reading ranges in scanf (7.19.6.2, 7.24.2.1)**

A - (dash) character is always treated as a range symbol.

### **%p in scanf (7.19.6.2, 7.24.2.2)**

The `%p` conversion specifier, scan pointer, to `scanf()` reads a hexadecimal number and converts it into a value with the type `void *`.



**File position errors (7.19.9.1, 7.19.9.3, 7.19.9.4)**

On file position errors, the functions `fgetpos`, `ftell`, and `fsetpos` store `EFPOS` in `errno`.

**An n-char-sequence after nan (7.20.1.3, 7.24.4.1.1)**

An n-char-sequence after a NaN is read and ignored.

**errno value at underflow (7.20.1.3, 7.24.4.1.1)**

`errno` is set to `ERANGE` if an underflow is encountered.

**Zero-sized heap objects (7.20.3)**

A request for a zero-sized heap object will return a valid pointer and not a null pointer.

**Behavior of abort and exit (7.20.4.1, 7.20.4.4)**

A call to `abort()` or `_Exit()` will not flush stream buffers, not close open streams, and not remove temporary files.

**Termination status (7.20.4.1, 7.20.4.3, 7.20.4.4)**

The termination status will be propagated to `__exit()` as a parameter. `exit()` and `_Exit()` use the input parameter, whereas `abort` uses `EXIT_FAILURE`.

**The system function return value (7.20.4.6)**

The `system` function is not supported.

**The time zone (7.23.1)**

The local time zone and daylight savings time must be defined by the application. For more information, see *Time*, page 97.

**Range and precision of time (7.23)**

The implementation uses `signed long` for representing `clock_t` and `time_t`, based at the start of the year 1970. This gives a range of approximately plus or minus 69 years in seconds. However, the application must supply the actual implementation for the functions `time` and `clock`. See *Time*, page 97.

**clock() (7.23.2.1)**

The application must supply an implementation of the `clock` function. See *Time*, page 97.

### **%Z replacement string (7.23.3.5, 7.24.5.1)**

By default, ":" is used as a replacement for %Z. Your application should implement the time zone handling. See *Time*, page 97.

### **Math functions rounding mode (F.9)**

The functions in `math.h` honor the rounding direction mode in `FLT-ROUNDS`.

## **J.3.13 Architecture**

### **Values and expressions assigned to some macros (5.2.4.2, 7.18.2, 7.18.3)**

There are always 8 bits in a byte.

`MB_LEN_MAX` is at the most 6 bytes depending on the library configuration that is used.

For information about sizes, ranges, etc for all basic types, see *Data representation*, page 231.

The limit macros for the exact-width, minimum-width, and fastest minimum-width integer types defined in `stdint.h` have the same ranges as `char`, `short`, `int`, `long`, and `long long`.

The floating-point constant `FLT_ROUNDS` has the value 1 (to nearest) and the floating-point constant `FLT_EVAL_METHOD` has the value 0 (treat as is).

### **The number, order, and encoding of bytes (6.2.6.1)**

See *Data representation*, page 231.

### **The value of the result of the sizeof operator (6.5.3.4)**

See *Data representation*, page 231.

## **J.4 Locale**

### **Members of the source and execution character set (5.2.1)**

By default, the compiler accepts all one-byte characters in the host's default character set. If the compiler option `--enable_multibytes` is used, the host multibyte characters are accepted in comments and string literals as well.

**The meaning of the additional character set (5.2.1.2)**

Any multibyte characters in the extended source character set is translated verbatim into the extended execution character set. It is up to your application with the support of the library configuration to handle the characters correctly.

**Shift states for encoding multibyte characters (5.2.1.2)**

Using the compiler option `--enable_multibytes` enables the use of the host's default multibyte characters as extended source characters.

**Direction of successive printing characters (5.2.2)**

The application defines the characteristics of a display device.

**The decimal point character (7.1.1)**

The default decimal-point character is a '.'. You can redefine it by defining the library configuration symbol `_LOCALE_DECIMAL_POINT`.

**Printing characters (7.4, 7.25.2)**

The set of printing characters is determined by the chosen locale.

**Control characters (7.4, 7.25.2)**

The set of control characters is determined by the chosen locale.

**Characters tested for (7.4.1.2, 7.4.1.3, 7.4.1.7, 7.4.1.9, 7.4.1.10, 7.4.1.11, 7.25.2.1.2, 7.25.5.1.3, 7.25.2.1.7, 7.25.2.1.9, 7.25.2.1.10, 7.25.2.1.11)**

The sets of characters tested are determined by the chosen locale.

**The native environment (7.1.1.1)**

The native environment is the same as the "C" locale.

**Subject sequences for numeric conversion functions (7.20.1, 7.24.4.1)**

There are no additional subject sequences that can be accepted by the numeric conversion functions.

**The collation of the execution character set (7.21.4.3, 7.24.4.4.2)**

The collation of the execution character set is determined by the chosen locale.

**Message returned by strerror (7.21.6.2)**

The messages returned by the `strerror` function depending on the argument is:

| Argument   | Message                   |
|------------|---------------------------|
| EZERO      | no error                  |
| EDOM       | domain error              |
| ERANGE     | range error               |
| EFPOS      | file positioning error    |
| EILSEQ     | multi-byte encoding error |
| <0    >99  | unknown error             |
| all others | error <i>nnn</i>          |

*Table 53: Message returned by strerror()—IAR DLIB library*

# Implementation-defined behavior for C89

This chapter describes how the compiler handles the implementation-defined areas of the C language based on the C89 standard.

If you are using Standard C instead of C89, see *Implementation-defined behavior*, page 333. For a short overview of the differences between Standard C and C89, see *C language overview*, page 135.

---

## 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 (5.1.2.2.1)

The function called at program startup is called `main`. No prototype was declared for `main`, and the only definition supported for `main` is:

```
int main(void)
```

To change this behavior for the IAR DLIB runtime environment, see *Customizing system initialization*, page 87. To change this behavior for the IAR CLIB runtime environment, see *Customizing system initialization*, page 114.

### **Interactive devices (5.1.2.3)**

The streams `stdin` and `stdout` are treated as interactive devices.

## **Identifiers**

### **Significant characters without external linkage (6.1.2)**

The number of significant initial characters in an identifier without external linkage is 200.

### **Significant characters with external linkage (6.1.2)**

The number of significant initial characters in an identifier with external linkage is 200.

### **Case distinctions are significant (6.1.2)**

Identifiers with external linkage are treated as case-sensitive.

## **Characters**

### **Source and execution character sets (5.2.1)**

The source character set is the set of legal characters that can appear in source files. The default source character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the source character set will be the host computer's default character set.

The execution character set is the set of legal characters that can appear in the execution environment. The default execution character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the execution character set will be the host computer's default character set. The IAR DLIB Library needs a multibyte character scanner to support a multibyte execution character set. The IAR CLIB Library does not support multibyte characters.

See *Locale*, page 93.

### **Bits per character in execution character set (5.2.4.2.1)**

The number of bits in a character is represented by the manifest constant `CHAR_BIT`. The standard include file `limits.h` defines `CHAR_BIT` as 8.

**Mapping of characters (6.1.3.4)**

The mapping of members of the source character set (in character and string literals) to members of the execution character set is made in a one-to-one way. In other words, the same representation value is used for each member in the character sets except for the escape sequences listed in the ISO standard.

**Unrepresented character constants (6.1.3.4)**

The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or in the extended character set for a wide character constant generates a diagnostic message, and will be truncated to fit the execution character set.

**Character constant with more than one character (6.1.3.4)**

An integer character constant that contains more than one character will be treated as an integer constant. The value will be calculated by treating the leftmost character as the most significant character, and the rightmost character as the least significant character, in an integer constant. A diagnostic message will be issued if the value cannot be represented in an integer constant.

A wide character constant that contains more than one multibyte character generates a diagnostic message.

**Converting multibyte characters (6.1.3.4)**

The only locale supported—that is, the only locale supplied with the IAR C/C++ Compiler—is the ‘C’ locale. If you use the command line option `--enable_multibytes`, the IAR DLIB Library will support multibyte characters if you add a locale with multibyte support or a multibyte character scanner to the library. The IAR CLIB Library does not support multibyte characters.

See *Locale*, page 93.

**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 232, 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 235, 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 239, for information about *size\_t*.

### **Conversion from/to pointers (6.3.4)**

See *Casting*, page 238, for information about casting of data pointers and function pointers.

### **ptrdiff\_t (6.3.6, 7.1.1)**

See *ptrdiff\_t*, page 239, 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 232, for information about the alignment requirement for data objects.

### **Sign of 'plain' bitfields (6.5.2.1)**

A 'plain' *int* bitfield is treated as a signed *int* bitfield. All integer types are allowed as bitfields.

### **Allocation order of bitfields within a unit (6.5.2.1)**

Bitfields are allocated within an integer from least-significant to most-significant bit.

### **Can bitfields straddle a storage-unit boundary (6.5.2.1)**

Bitfields cannot straddle a storage-unit boundary for the chosen bitfield integer type.

### **Integer type chosen to represent enumeration types (6.5.2.2)**

The chosen integer type for a specific enumeration type depends on the enumeration constants defined for the enumeration type. The chosen integer type is the smallest possible.

## **Qualifiers**

### **Access to volatile objects (6.5.3)**

Any reference to an object with volatile qualified type is an access.

## **Declarators**

### **Maximum numbers of declarators (6.5.4)**

The number of declarators is not limited. The number is limited only by the available memory.

## **Statements**

### **Maximum number of case statements (6.6.4.2)**

The number of case statements (case values) in a switch statement is not limited. The number is limited only by the available memory.

## **Preprocessing directives**

### **Character constants and conditional inclusion (6.8.1)**

The character set used in the preprocessor directives is the same as the execution character set. The preprocessor recognizes negative character values if a 'plain' character is treated as a signed character.

### **Including bracketed filenames (6.8.2)**

For file specifications enclosed in angle brackets, the preprocessor does not search directories of the parent files. A parent file is the file that contains the `#include` directive. Instead, it begins by searching for the file in the directories specified on the compiler command line.

### Including quoted filenames (6.8.2)

For file specifications enclosed in quotes, the preprocessor directory search begins with the directories of the parent file, then proceeds through the directories of any grandparent files. Thus, searching begins relative to the directory containing the source file currently being processed. If there is no grandparent file and the file is not found, the search continues as if the filename was enclosed in angle brackets.

### Character sequences (6.8.2)

Preprocessor directives use the source character set, except for escape sequences. Thus, to specify a path for an include file, use only one backslash:

```
#include "mydirectory\myfile"
```

Within source code, two backslashes are necessary:

```
file = fopen("mydirectory\\myfile", "rt");
```

### Recognized pragma directives (6.8.6)

In addition to the pragma directives described in the chapter *Pragma directives*, the following directives are recognized and will have an indeterminate effect. If a pragma directive is listed both in the *Pragma directives* chapter and here, the information provided in the *Pragma directives* chapter overrides the information here.

```
alignment
baseaddr
building_runtime
can_instantiate
codeseg
cspy_support
define_type_info
do_not_instantiate
early_dynamic_initialization
function
hdrstop
important_typedef
instantiate
keep_definition
library_requirement_override
library_provides
library_requirement_override
```

```
memory
module_name
no_pch
once
public_equ
system_include
warnings
```

### **Default `__DATE__` and `__TIME__` (6.8.8)**

The definitions for `__TIME__` and `__DATE__` are always available.

## **IAR DLIB Library functions**

The information in this section is valid only if the runtime library configuration you have chosen supports file descriptors. See the chapter *The DLIB runtime environment* for more information about runtime library configurations.

### **NULL macro (7.1.6)**

The `NULL` macro is defined to 0.

### **Diagnostic printed by the assert function (7.2)**

The `assert()` function prints:

```
filename:linenr expression -- assertion failed
```

when the parameter evaluates to zero.

### **Domain errors (7.5.1)**

NaN (Not a Number) will be returned by the mathematic functions on domain errors.

### **Underflow of floating-point values sets `errno` to `ERANGE` (7.5.1)**

The mathematics functions set the integer expression `errno` to `ERANGE` (a macro in `errno.h`) on underflow range errors.

### **`fmod()` functionality (7.5.6.4)**

If the second argument to `fmod()` is zero, the function returns NaN; `errno` is set to `EDOM`.

**signal() (7.7.1.1)**

The signal part of the library is not supported.

**Note:** Low-level interface functions exist in the library, but will not perform anything. Use the template source code to implement application-specific signal handling. See *Signal and raise*, page 96.

**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 92.

**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 92.

**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 92.

**%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 95.

### **system() (7.10.4.5)**

How the command processor works depends on how you have implemented the `system` function. See *Environment interaction*, page 95.

Message returned by `strerror()` (7.11.6.2)

The messages returned by `strerror()` depending on the argument is:

| Argument   | Message                   |
|------------|---------------------------|
| EZERO      | no error                  |
| EDOM       | domain error              |
| ERANGE     | range error               |
| EFPOS      | file positioning error    |
| EILSEQ     | multi-byte encoding error |
| <0    >99  | unknown error             |
| all others | error <i>nnn</i>          |

Table 54: 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 97.

`clock()` (7.12.2.1)

From where the system clock starts counting depends on how you have implemented the `clock` function. See *Time*, page 97.

IAR CLIB Library functions

NULL macro (7.1.6)

The `NULL` macro is defined to `(void *) 0`.

Diagnostic printed by the `assert` function (7.2)

The `assert()` function prints:

Assertion failed: *expression*, file *Filename*, line *linenumber*  
when the parameter evaluates to zero.

Domain errors (7.5.1)

`HUGE_VAL`, the largest representable value in a double floating-point type, 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 zero (it does not change the integer expression `errno`).

### **`signal()` (7.7.1.1)**

The signal part of the library is not supported.

### **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)**

There are no binary streams implemented.

### **Files (7.9.3)**

There are no other streams than `stdin` and `stdout`. This means that a file system is not implemented.

### **`remove()` (7.9.4.1)**

There are no other streams than `stdin` and `stdout`. This means that a file system is not implemented.

### **`rename()` (7.9.4.2)**

There are no other streams than `stdin` and `stdout`. This means that a file system is not implemented.



**%p in printf() (7.9.6.1)**

The argument to a `%p` conversion specifier, print pointer, to `printf()` is treated as having the type `'char *'`. 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 explicitly as a `-` character.

**File position errors (7.9.9.1, 7.9.9.4)**

There are no other streams than `stdin` and `stdout`. This means that a file system is not implemented.

**Message generated by perror() (7.9.10.4)**

`perror()` is not supported.

**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 `exit()` function does not return.

**Environment (7.10.4.4)**

Environments are not supported.

**system() (7.10.4.5)**

The `system()` function is not supported.

**Message returned by strerror() (7.11.6.2)**

The messages returned by `strerror()` depending on the argument are:

| Argument   | Message       |
|------------|---------------|
| EZERO      | no error      |
| EDOM       | domain error  |
| ERANGE     | range error   |
| <0    >99  | unknown error |
| all others | error No.xx   |

*Table 55: Message returned by strerror()—IAR CLIB library*

**The time zone (7.12.1)**

The time zone function is not supported.

**clock() (7.12.2.1)**

The `clock()` function is not supported.

# A

- abort
  - implementation-defined behavior. . . . . 345
  - implementation-defined behavior (CLIB) . . . . . 361
  - implementation-defined behavior (DLIB) . . . . . 358
  - system termination (DLIB) . . . . . 86
- absolute location
  - data, placing at (@) . . . . . 165
  - language support for . . . . . 138
  - #pragma location . . . . . 274
- address spaces, managing multiple . . . . . 68
- addressing. *See* memory attributes,  
and memory models
- algorithm (STL header file) . . . . . 305
- alignment . . . . . 231
  - forcing stricter (#pragma data\_alignment) . . . . . 268
  - in structures (#pragma pack) . . . . . 277
  - of an object (\_\_ALIGNOF\_\_) . . . . . 138
  - of data types. . . . . 231
- alignment (pragma directive) . . . . . 340, 355
- \_\_ALIGNOF\_\_ (operator) . . . . . 138
- anonymous structures . . . . . 162
- anonymous symbols, creating . . . . . 135
- application
  - building, overview of . . . . . 28
  - startup and termination (CLIB) . . . . . 113
  - startup and termination (DLIB) . . . . . 84
- architecture
  - more information about . . . . . 17
  - of AVR. . . . . 37
- ARGFRAME (assembler directive) . . . . . 132
- argv (argument), implementation-defined behavior . . . . 334
- arrays
  - designated initializers in . . . . . 135
  - hints about index type . . . . . 161
  - implementation-defined behavior. . . . . 338, 353
  - incomplete at end of structs . . . . . 135
  - non-lvalue . . . . . 141
  - of incomplete types . . . . . 140
  - single-value initialization. . . . . 142
- asm, \_\_asm (language extension) . . . . . 136
- assembler code
  - calling from C . . . . . 120
  - calling from C++ . . . . . 122
  - inserting inline . . . . . 119
- assembler directives
  - for call frame information . . . . . 133
  - for static overlay . . . . . 132
  - using in inline assembler code . . . . . 119
- assembler instructions, inserting inline . . . . . 119
- assembler labels, making public (--public\_equ) . . . . . 220
- assembler language interface . . . . . 117
  - calling convention. *See* assembler code
- assembler list file, generating . . . . . 209
- assembler output file . . . . . 122
- assembler, inline . . . . . 136
- asserts . . . . . 98
  - implementation-defined behavior of . . . . . 341
  - implementation-defined behavior of, (CLIB). . . . . 359
  - implementation-defined behavior of, (DLIB). . . . . 356
  - including in application . . . . . 299
- assert.h (CLIB header file) . . . . . 310
- assert.h (DLIB header file) . . . . . 304
- atomic operations . . . . . 49
  - \_\_monitor . . . . . 257
- attributes
  - object . . . . . 248
  - type . . . . . 245
- auto variables . . . . . 44
  - at function entrance . . . . . 127
  - programming hints for efficient code . . . . . 173
  - using in inline assembler code . . . . . 119
- AVR
  - memory layout. . . . . 37
  - supported devices. . . . . 28

# B

|                                                         |          |
|---------------------------------------------------------|----------|
| backtrace information <i>See</i> call frame information |          |
| Barr, Michael                                           | 20       |
| baseaddr (pragma directive)                             | 341, 355 |
| __BASE_FILE__ (predefined symbol)                       | 294      |
| basic_template_matching (pragma directive)              | 267      |
| using                                                   | 152      |
| batch files                                             |          |
| error return codes                                      | 184      |
| for building library from command line                  | 83       |
| binary streams                                          | 343      |
| binary streams (CLIB)                                   | 360      |
| binary streams (DLIB)                                   | 357      |
| bit negation                                            | 174      |
| bitfields                                               |          |
| data representation of                                  | 233      |
| hints                                                   | 161      |
| implementation-defined behavior                         | 339      |
| implementation-defined behavior of                      | 353      |
| non-standard types in                                   | 139      |
| bitfields (pragma directive)                            | 267      |
| bits in a byte, implementation-defined behavior         | 335      |
| bold style, in this guide                               | 22       |
| bool (data type)                                        | 232      |
| adding support for in CLIB                              | 310      |
| adding support for in DLIB                              | 304, 306 |
| building_runtime (pragma directive)                     | 341, 355 |
| __BUILD_NUMBER__ (predefined symbol)                    | 294      |
| byte order (of value), reversing                        | 290      |
| byte order, identifying                                 | 296      |

# C

|                                                         |     |
|---------------------------------------------------------|-----|
| C and C++ linkage                                       | 125 |
| C/C++ calling convention. <i>See</i> calling convention |     |
| C header files                                          | 303 |
| C language, overview                                    | 135 |

|                                                    |          |
|----------------------------------------------------|----------|
| call frame information                             | 133      |
| in assembler list file                             | 121      |
| in assembler list file (-IA)                       | 209      |
| call stack                                         | 133      |
| callee-save registers, stored on stack             | 44       |
| calling convention                                 |          |
| C++, requiring C linkage                           | 122      |
| in compiler                                        | 123      |
| overriding default (__version_1)                   | 263      |
| calloc (library function)                          | 45       |
| <i>See also</i> heap                               |          |
| implementation-defined behavior of (CLIB)          | 361      |
| implementation-defined behavior of (DLIB)          | 358      |
| can_instantiate (pragma directive)                 | 341, 355 |
| cassert (library header file)                      | 306      |
| cast operators                                     |          |
| in Extended EC++                                   | 144, 154 |
| missing from Embedded C++                          | 144      |
| casting                                            |          |
| between pointer types                              | 42       |
| of pointers and integers                           | 238      |
| pointers to integers, language extension           | 140      |
| ccomplex (library header file)                     | 306      |
| cctype (DLIB header file)                          | 306      |
| cerrno (DLIB header file)                          | 306      |
| cfenv (library header file)                        | 306      |
| CFI (assembler directive)                          | 133      |
| cfloat (DLIB header file)                          | 306      |
| char (data type)                                   | 232      |
| changing default representation (--char_is_signed) | 195      |
| changing representation (--char_is_unsigned)       | 195      |
| implementation-defined behavior                    | 336      |
| signed and unsigned                                | 233      |
| character set, implementation-defined behavior     | 334      |
| characters, implementation-defined behavior of     | 335, 350 |
| character-based I/O                                |          |
| in CLIB                                            | 111      |
| in DLIB                                            | 89       |
| overriding in runtime library                      | 72, 81   |

- char\_is\_signed (compiler option) . . . . . 195
- char\_is\_unsigned (compiler option) . . . . . 195
- CHECKSUM (segment) . . . . . 319
- cinttypes (DLIB header file) . . . . . 306
- ciso646 (library header file) . . . . . 306
- class memory (extended EC++) . . . . . 146
- class template partial specialization  
  matching (extended EC++) . . . . . 151
- CLI (assembler instruction) . . . . . 285
- CLIB . . . . . 32, 309
  - documentation . . . . . 20
  - runtime environment . . . . . 109
  - summary of definitions . . . . . 310
- clib (compiler option) . . . . . 195
- climits (DLIB header file) . . . . . 306
- locale (DLIB header file) . . . . . 306
- clock (CLIB library function),  
  implementation-defined behavior of . . . . . 362
- clock (DLIB library function),  
  implementation-defined behavior of . . . . . 359
- clock (library function)  
  implementation-defined behavior . . . . . 345
- clock.c . . . . . 97
- \_\_close (DLIB library function) . . . . . 93
- clustering (compiler transformation). . . . . 172
  - disabling (--no\_clustering) . . . . . 212
- cmath (DLIB header file) . . . . . 306
- code
  - interruption of execution . . . . . 48
  - verifying linked result . . . . . 67
- code motion (compiler transformation). . . . . 171
  - disabling (--no\_code\_motion) . . . . . 213
- code segments, used for placement . . . . . 65
- CODE (segment) . . . . . 319
- codeseg (pragma directive) . . . . . 341, 355
- command line options
  - part of compiler invocation syntax . . . . . 181
  - passing . . . . . 181
  - See also* compiler options
  - typographic convention . . . . . 22
- command prompt icon, in this guide . . . . . 22
- comments
  - after preprocessor directives . . . . . 141
  - C++ style, using in C code . . . . . 135
- common block (call frame information) . . . . . 133
- common subexpr elimination (compiler transformation) . 170
  - disabling (--no\_cse) . . . . . 213
- compilation date
  - exact time of (\_\_TIME\_\_) . . . . . 297
  - identifying (\_\_DATE\_\_) . . . . . 294
- compiler
  - environment variables . . . . . 182
  - invocation syntax . . . . . 181
  - output from . . . . . 183
- compiler listing, generating (-l) . . . . . 209
- compiler object file . . . . . 28
  - including debug information in (--debug, -r) . . . . . 197
  - output from compiler . . . . . 183
- compiler optimization levels . . . . . 169
- compiler options . . . . . 187
  - passing to compiler . . . . . 181
  - reading from file (-f) . . . . . 206–207
  - specifying parameters . . . . . 189
  - summary . . . . . 190
  - syntax . . . . . 187
  - for creating skeleton code . . . . . 121
  - initializers\_in\_flash . . . . . 226
  - warnings\_affect\_exit\_code . . . . . 184
- compiler platform, identifying . . . . . 296
- compiler subversion number . . . . . 297
- compiler transformations . . . . . 168
- compiler version number . . . . . 298
- compiling
  - from the command line . . . . . 28
  - syntax . . . . . 181
- complex numbers, supported in Embedded C++ . . . . . 144
- complex (library header file) . . . . . 305
- complex.h (library header file) . . . . . 304
- compound literals . . . . . 135

|                                        |          |
|----------------------------------------|----------|
| computer style, typographic convention | 22       |
| configuration                          |          |
| basic project settings                 | 29       |
| __low_level_init                       | 87       |
| configuration symbols                  |          |
| for file input and output              | 92       |
| for locale                             | 94       |
| for printf and scanf                   | 91       |
| for strtod                             | 97       |
| in library configuration files         | 83, 88   |
| consistency, module                    | 103      |
| const                                  |          |
| declaring objects                      | 242      |
| non-top level                          | 141      |
| constants, placing in named segment    | 268      |
| constseg (pragma directive)            | 268      |
| const_cast (cast operator)             | 144      |
| contents, of this guide                | 18       |
| control characters,                    |          |
| implementation-defined behavior        | 347      |
| conventions, used in this guide        | 21       |
| copyright notice                       | 2        |
| __CORE__ (predefined symbol)           | 294      |
| core, identifying                      | 294      |
| cos (library function)                 | 302      |
| __cplusplus (predefined symbol)        | 294      |
| __CPU__ (predefined symbol)            | 294      |
| __cpu (runtime model attribute)        | 105      |
| --cpu (compiler option)                | 196      |
| __cpu_name (runtime model attribute)   | 105      |
| cpu, specifying on command line        | 196      |
| cross call (compiler transformation)   | 172      |
| --cross_call_passes (compiler option)  | 196      |
| csetjmp (DLIB header file)             | 306      |
| csignal (DLIB header file)             | 306      |
| cspy_support (pragma directive)        | 341, 355 |
| CSTACK (segment)                       | 319      |
| example                                | 61       |
| <i>See also</i> stack                  |          |

|                                                        |          |
|--------------------------------------------------------|----------|
| cstartup (system startup code)                         |          |
| customizing system initialization                      | 87       |
| process for overriding in runtime library              | 81       |
| cstartup.s90 (system startup code)                     |          |
| source files for (CLIB)                                | 113      |
| source files for (DLIB)                                | 84       |
| cstdarg (DLIB header file)                             | 306      |
| cstdbool (DLIB header file)                            | 306      |
| cstddef (DLIB header file)                             | 306      |
| cstdio (DLIB header file)                              | 306      |
| stdlib (DLIB header file)                              | 307      |
| cstring (DLIB header file)                             | 307      |
| ctgmth (library header file)                           | 307      |
| ctime (DLIB header file)                               | 307      |
| ctype.h (library header file)                          | 304, 310 |
| cwctype.h (library header file)                        | 307      |
| ?C_EXIT (assembler label)                              | 115      |
| ?C_GETCHAR (assembler label)                           | 115      |
| C_INCLUDE (environment variable)                       | 182      |
| ?C_PUTCHAR (assembler label)                           | 115      |
| C-SPY                                                  |          |
| debug support for C++                                  | 151      |
| interface to system termination                        | 87       |
| low-level interface (CLIB)                             | 115      |
| Terminal I/O window, including debug support for       | 79–80    |
| C++                                                    |          |
| <i>See also</i> Embedded C++ and Extended Embedded C++ |          |
| absolute location                                      | 166–167  |
| calling convention                                     | 122      |
| dynamic initialization in                              | 66       |
| header files                                           | 304      |
| language extensions                                    | 156      |
| special function types                                 | 51       |
| static member variables                                | 166–167  |
| support for                                            | 27       |
| C++ names, in assembler code                           | 123      |
| C++ objects, placing in memory type                    | 43       |
| C++ terminology                                        | 21       |
| C++-style comments                                     | 135      |

--c89 (compiler option) . . . . . 194

## D

-D (compiler option) . . . . . 197

### data

- alignment of . . . . . 231
- different ways of storing . . . . . 37
- located, declaring extern . . . . . 166
- placing . . . . . 164, 222, 269, 317
  - at absolute location . . . . . 165
- representation of . . . . . 231
- storage . . . . . 37
- verifying linked result . . . . . 67
- data block (call frame information) . . . . . 133
- data bus, enabling external . . . . . 204
- data memory attributes, using . . . . . 39
- data pointers . . . . . 237
- data segments . . . . . 57
- data types . . . . . 232
  - avoiding signed . . . . . 160
  - floating point . . . . . 235
  - in C++ . . . . . 243
  - integer types . . . . . 232
- dataseg (pragma directive) . . . . . 269
- data\_alignment (pragma directive) . . . . . 268
- \_\_DATE\_\_ (predefined symbol) . . . . . 294
- date (library function), configuring support for . . . . . 97
- debug (compiler option) . . . . . 197
- debug information, including in object file . . . . . 197
- decimal point, implementation-defined behavior . . . . . 347
- declarations
  - empty . . . . . 141
  - in for loops . . . . . 135
  - Kernighan & Ritchie . . . . . 174
  - of functions . . . . . 125
- declarations and statements, mixing . . . . . 135
- declarators, implementation-defined behavior . . . . . 354
- define\_type\_info (pragma directive) . . . . . 341, 355

- \_\_delay\_cycles (intrinsic function) . . . . . 284
- delete operator (extended EC++) . . . . . 149
- delete (keyword) . . . . . 45
- denormalized numbers. *See* subnormal numbers
- dependencies (compiler option) . . . . . 198
- deque (STL header file) . . . . . 305
- destructors and interrupts, using . . . . . 150
- \_\_DES\_decryption (intrinsic function) . . . . . 284
- \_\_DES\_encryption (intrinsic function) . . . . . 285
- \_\_device\_\_ (predefined symbol) . . . . . 294
- diagnostic messages . . . . . 185
  - classifying as compilation errors . . . . . 199
  - classifying as compilation remarks . . . . . 199
  - classifying as compiler warnings . . . . . 200
  - disabling compiler warnings . . . . . 217
  - disabling wrapping of in compiler . . . . . 217
  - enabling compiler remarks . . . . . 221
  - listing all used by compiler . . . . . 200
  - suppressing in compiler . . . . . 199
- diagnostics\_tables (compiler option) . . . . . 200
- diagnostics, implementation-defined behavior . . . . . 333
- diag\_default (pragma directive) . . . . . 269
- diag\_error (compiler option) . . . . . 199
- diag\_error (pragma directive) . . . . . 270
- diag\_remark (compiler option) . . . . . 199
- diag\_remark (pragma directive) . . . . . 270
- diag\_suppress (compiler option) . . . . . 199
- diag\_suppress (pragma directive) . . . . . 270
- diag\_warning (compiler option) . . . . . 200
- diag\_warning (pragma directive) . . . . . 271
- DIFUNCT (segment) . . . . . 67, 320
- directives
  - function for static overlay . . . . . 132
  - pragma . . . . . 34, 265
- directory, specifying as parameter . . . . . 188
- disable\_direct\_mode (compiler option) . . . . . 201
- \_\_disable\_interrupt (intrinsic function) . . . . . 285
- discard\_unused\_publics (compiler option) . . . . . 201
- disclaimer . . . . . 2

|                                                                  |               |
|------------------------------------------------------------------|---------------|
| <code>dlavrlibname.h</code> .....                                | 83            |
| <code>DLIB</code> .....                                          | 32, 303       |
| configurations .....                                             | 88            |
| configuring .....                                                | 72, 202       |
| documentation .....                                              | 20            |
| including debug support .....                                    | 78            |
| reference information. <i>See the online help system</i> .....   | 301           |
| runtime environment .....                                        | 71            |
| <code>--dlib</code> (compiler option) .....                      | 201           |
| <code>--dlib_config</code> (compiler option) .....               | 202           |
| <code>DLib_Defaults.h</code> (library configuration file) .....  | 83, 88        |
| <code>__DLIB_FILE_DESCRIPTOR</code> (configuration symbol) ..... | 92            |
| document conventions .....                                       | 21            |
| documentation .....                                              |               |
| overview of guides .....                                         | 19            |
| documentation, library .....                                     | 301           |
| domain errors, implementation-defined behavior .....             | 342, 356, 359 |
| <code>__DOUBLE__</code> (predefined symbol) .....                | 294           |
| double (data type) .....                                         | 235           |
| avoiding .....                                                   | 161           |
| configuring size of floating-point type .....                    | 32            |
| identifying size of ( <code>__DOUBLE__</code> ) .....            | 294           |
| <code>__double_size</code> (runtime model attribute) .....       | 105           |
| <code>--do_cross_call</code> (compiler option) .....             | 202           |
| <code>do_not_instantiate</code> (pragma directive) .....         | 341, 355      |
| <code>DW</code> (directive) .....                                | 287           |
| dynamic initialization .....                                     | 84, 113       |
| in C++ .....                                                     | 66            |
| dynamic memory .....                                             | 45            |

## E

|                                                            |          |
|------------------------------------------------------------|----------|
| <code>-e</code> (compiler option) .....                    | 203      |
| <code>early_initialization</code> (pragma directive) ..... | 341, 355 |
| <code>--ec++</code> (compiler option) .....                | 203      |
| <code>EC++</code> header files .....                       | 305      |
| edition, of this guide .....                               | 2        |
| <code>--eecr_address</code> (compiler option) .....        | 204      |

|                                                                              |          |
|------------------------------------------------------------------------------|----------|
| <code>--eec++</code> (compiler option) .....                                 | 203      |
| <code>__eeprom</code> (extended keyword) .....                               | 238, 250 |
| <code>EEPROM_I</code> (segment) .....                                        | 320      |
| <code>EEPROM_N</code> (segment) .....                                        | 321      |
| <code>--eeprom_size</code> (compiler option) .....                           | 204      |
| <code>ELPM</code> (instruction) .....                                        | 194      |
| Embedded C++ .....                                                           | 143      |
| differences from C++ .....                                                   | 143      |
| enabling .....                                                               | 203      |
| function linkage .....                                                       | 125      |
| language extensions .....                                                    | 143      |
| overview .....                                                               | 143      |
| Embedded C++ Technical Committee .....                                       | 21       |
| embedded systems, IAR special support for .....                              | 34       |
| <code>__embedded_cplusplus</code> (predefined symbol) .....                  | 295      |
| <code>--enable_external_bus</code> (compiler option) .....                   | 204      |
| <code>__enable_interrupt</code> (intrinsic function) .....                   | 285      |
| <code>--enable_multibytes</code> (compiler option) .....                     | 205      |
| <code>__enhanced_core</code> (runtime model attribute) .....                 | 106      |
| <code>--enhanced_core</code> (compiler option) .....                         | 205      |
| entry label, program .....                                                   | 85       |
| enumerations, implementation-defined behavior .....                          | 339, 353 |
| enums .....                                                                  |          |
| data representation .....                                                    | 233      |
| forward declarations of .....                                                | 140      |
| environment .....                                                            |          |
| implementation-defined behavior .....                                        | 334, 349 |
| runtime (CLIB) .....                                                         | 109      |
| runtime (DLIB) .....                                                         | 71       |
| environment names, implementation-defined behavior .....                     | 335      |
| environment variables .....                                                  |          |
| <code>C_INCLUDE</code> .....                                                 | 182      |
| <code>QCCAVR</code> .....                                                    | 182      |
| environment (native) .....                                                   |          |
| implementation-defined behavior .....                                        | 347      |
| <code>EQU</code> (assembler directive) .....                                 | 220      |
| <code>ERANGE</code> .....                                                    | 342, 356 |
| <code>errno</code> value at underflow, implementation-defined behavior ..... | 345      |
| <code>errno.h</code> (library header file) .....                             | 304, 310 |



- error messages . . . . . 186
  - classifying for compiler . . . . . 199
- error return codes . . . . . 184
- error (pragma directive) . . . . . 271
- error\_limit (compiler option) . . . . . 206
- escape sequences, implementation-defined behavior . . . . 335
- exception handling, missing from Embedded C++ . . . . 143
- exception vectors . . . . . 66
- \_Exit (library function) . . . . . 86
- exit (library function) . . . . . 86
  - implementation-defined behavior. . . . . 345, 358, 361
- \_exit (library function) . . . . . 86
- \_\_exit (library function) . . . . . 86
- export keyword, missing from Extended EC++ . . . . . 151
- extended command line file
  - for compiler . . . . . 206–207
  - passing options. . . . . 181
- Extended Embedded C++. . . . . 144
  - enabling . . . . . 203
  - standard template library (STL). . . . . 305
- extended keywords . . . . . 245
  - enabling (-e). . . . . 203
  - overview . . . . . 34
  - summary . . . . . 249
  - syntax. . . . . 41
    - object attributes. . . . . 249
    - type attributes on data objects . . . . . 246
    - type attributes on data pointers . . . . . 247
    - type attributes on function pointers . . . . . 248
    - type attributes on functions . . . . . 247
- \_\_eeprom . . . . . 238
- \_\_farflash . . . . . 238
- \_\_flash . . . . . 238
- \_\_generic . . . . . 238
- \_\_hugeflash . . . . . 238
- \_\_root. . . . . 221
- \_\_tinyflash . . . . . 238
- \_\_extended\_load\_program\_memory (intrinsic function) . 286
- extern "C" linkage. . . . . 148

- external data bus, enabling . . . . . 204
- external memory . . . . . 226–227
- \_\_ext\_io (extended keyword) . . . . . 251

## F

- f (compiler option). . . . . 206–207
- \_\_far (extended keyword) . . . . . 252
- FARCODE (segment) . . . . . 321
- \_\_farflash (extended keyword) . . . . . 252
- \_\_farfunc (extended keyword) . . . . . 253
- \_\_farfunc (function pointer) . . . . . 237
- FAR\_C (segment) . . . . . 321
- FAR\_F (segment) . . . . . 322
- FAR\_HEAP (segment) . . . . . 322
- FAR\_I (segment). . . . . 322
- FAR\_ID (segment) . . . . . 323
- FAR\_N (segment) . . . . . 323
- FAR\_Z (segment) . . . . . 323
- fatal error messages . . . . . 186
- fdopen, in stdio.h . . . . . 307
- fegettrapdisable. . . . . 307
- fegettrapenable . . . . . 307
- FENV\_ACCESS, implementation-defined behavior. . . . 338
- fenv.h (library header file). . . . . 304, 306
  - additional C functionality. . . . . 307
- fgetpos (library function), implementation-defined
  - behavior . . . . . 345, 358
- field width, library support for . . . . . 112
- \_\_FILE\_\_ (predefined symbol). . . . . 295
- file buffering, implementation-defined behavior . . . . 343
- file dependencies, tracking . . . . . 198
- file paths, specifying for #include files . . . . . 208
- file position, implementation-defined behavior. . . . . 343
- file systems . . . . . 360
- file (zero-length), implementation-defined behavior. . . . 343
- filename
  - extension for linker command file . . . . . 54
  - of object file. . . . . 218

|                                                                    |               |                                                                          |                    |
|--------------------------------------------------------------------|---------------|--------------------------------------------------------------------------|--------------------|
| search procedure for . . . . .                                     | 182           | FP_CONTRACT, implementation-defined behavior. . . . .                    | 338                |
| specifying as parameter . . . . .                                  | 188           | __fractional_multiply_signed (intrinsic function). . . . .               | 286                |
| filenames (legal), implementation-defined behavior . . . . .       | 343           | __fractional_multiply_signed_with_unsigned (intrinsic function). . . . . | 286                |
| fileno, in stdio.h . . . . .                                       | 307           | __fractional_multiply_unsigned (intrinsic function). . . . .             | 286                |
| files, implementation-defined behavior                             |               | fragmentation, of heap memory . . . . .                                  | 45                 |
| handling of temporary . . . . .                                    | 344           | free (library function). <i>See also</i> heap . . . . .                  | 45                 |
| multibyte characters in. . . . .                                   | 344           | fsetpos (library function), implementation-defined behavior . . . . .    | 345                |
| opening . . . . .                                                  | 343           | fstream (library header file) . . . . .                                  | 305                |
| __flash (extended keyword) . . . . .                               | 253           | ftell (library function), implementation-defined behavior . . . . .      | 345, 358           |
| flash memory . . . . .                                             | 226           | Full DLIB (library configuration) . . . . .                              | 88                 |
| library routines for accessing . . . . .                           | 310           | __func__ (predefined symbol) . . . . .                                   | 142, 295           |
| placing aggregate initializers . . . . .                           | 208           | FUNCALL (assembler directive) . . . . .                                  | 132                |
| float (data type). . . . .                                         | 235           | __FUNCTION__ (predefined symbol) . . . . .                               | 142, 295           |
| floating-point constants                                           |               | function calls                                                           |                    |
| hexadecimal notation . . . . .                                     | 135           | calling convention . . . . .                                             | 123                |
| hints . . . . .                                                    | 161           | stack image after . . . . .                                              | 129                |
| floating-point environment, accessing or not . . . . .             | 280           | function declarations, Kernighan & Ritchie . . . . .                     | 174                |
| floating-point expressions, contracting or not . . . . .           | 281           | function directives for static overlay . . . . .                         | 132                |
| floating-point format. . . . .                                     | 235           | function inlining (compiler transformation) . . . . .                    | 170                |
| hints . . . . .                                                    | 161           | disabling (--no_inline) . . . . .                                        | 214                |
| implementation-defined behavior. . . . .                           | 337, 352      | function pointers . . . . .                                              | 237                |
| special cases. . . . .                                             | 236           | function prototypes . . . . .                                            | 173                |
| 32-bits . . . . .                                                  | 236           | enforcing . . . . .                                                      | 221                |
| 64-bits . . . . .                                                  | 236           | function return addresses . . . . .                                      | 130                |
| floating-point numbers, support for in printf formatters . . . . . | 112           | function storage . . . . .                                               | 47                 |
| floating-point status flags . . . . .                              | 307           | function template parameter deduction (extended EC++) . . . . .          | 152                |
| floating-point type, configuring size of double . . . . .          | 32            | function type information, omitting in object output. . . . .            | 218                |
| float.h (library header file) . . . . .                            | 304, 310      | FUNCTION (assembler directive) . . . . .                                 | 132                |
| FLT_EVAL_METHOD, implementation-defined behavior . . . . .         | 337, 342, 346 | function (pragma directive). . . . .                                     | 341, 355           |
| FLT_ROUNDS, implementation-defined behavior . . . . .              | 337, 346      | functional (STL header file) . . . . .                                   | 305                |
| fmod (library function),                                           |               | functions . . . . .                                                      | 47                 |
| implementation-defined behavior . . . . .                          | 356, 360      | C++ and special function types . . . . .                                 | 51                 |
| for loops, declarations in. . . . .                                | 135           | declaring . . . . .                                                      | 125, 173           |
| --force_switch_type (compiler option) . . . . .                    | 206           | inlining. . . . .                                                        | 135, 170, 173, 272 |
| formats                                                            |               | interrupt . . . . .                                                      | 48–49              |
| floating-point values . . . . .                                    | 235           | intrinsic . . . . .                                                      | 117, 173           |
| standard IEEE (floating point) . . . . .                           | 235           | monitor . . . . .                                                        | 49                 |
| __formatted_write (library function) . . . . .                     | 112           |                                                                          |                    |

|                               |               |
|-------------------------------|---------------|
| omitting type info .....      | 218           |
| parameters .....              | 127           |
| placing in memory .....       | 164, 166, 222 |
| recursive                     |               |
| avoiding .....                | 173           |
| storing data on stack .....   | 44            |
| reentrancy (DLIB) .....       | 302           |
| related extensions .....      | 47            |
| return values from .....      | 130           |
| special function types .....  | 48            |
| verifying linked result ..... | 67            |

## G

|                                                           |     |
|-----------------------------------------------------------|-----|
| __generic (extended keyword) .....                        | 254 |
| generic pointers, avoiding .....                          | 162 |
| getchar (library function) .....                          | 111 |
| getenv (library function), configuring support for .....  | 95  |
| getw, in stdio.h .....                                    | 308 |
| getzone (library function), configuring support for ..... | 97  |
| getzone.c .....                                           | 97  |
| __get_interrupt_state (intrinsic function) .....          | 287 |
| global variables, initialization .....                    | 59  |
| --guard_calls (compiler option) .....                     | 207 |
| guidelines, reading .....                                 | 17  |

## H

|                                              |     |
|----------------------------------------------|-----|
| Harbison, Samuel P. ....                     | 20  |
| hardware support in compiler .....           | 71  |
| hash_map (STL header file) .....             | 305 |
| hash_set (STL header file) .....             | 305 |
| __HAS_EEPROM__ (predefined symbol) .....     | 295 |
| __HAS_EIND__ (predefined symbol) .....       | 295 |
| __HAS_ELPM__ (predefined symbol) .....       | 295 |
| __HAS_ENHANCED_CORE__ (predefined symbol) .. | 295 |
| __HAS_FISCR__ (predefined symbol) .....      | 296 |
| __HAS_MUL__ (predefined symbol) .....        | 296 |
| __HAS_RAMPD__ (predefined symbol) .....      | 296 |

|                                                     |          |
|-----------------------------------------------------|----------|
| __HAS_RAMPX__ (predefined symbol) .....             | 296      |
| __HAS_RAMPY__ (predefined symbol) .....             | 296      |
| __HAS_RAMPZ__ (predefined symbol) .....             | 296      |
| hdrstop (pragma directive) .....                    | 341, 355 |
| header files                                        |          |
| C .....                                             | 303      |
| C++ .....                                           | 304      |
| EC++ .....                                          | 305      |
| library .....                                       | 301      |
| special function registers .....                    | 176      |
| STL .....                                           | 305      |
| dlavrlibname.h .....                                | 83       |
| DLib_Defaults.h .....                               | 83, 88   |
| including stdbool.h for bool .....                  | 232      |
| including stddef.h for wchar_t .....                | 233      |
| header names, implementation-defined behavior ..... | 340      |
| --header_context (compiler option) .....            | 207      |
| heap                                                |          |
| DLIB support for multiple .....                     | 98       |
| dynamic memory .....                                | 45       |
| segments for .....                                  | 64       |
| storing data .....                                  | 38       |
| heap segments                                       |          |
| CLIB .....                                          | 64       |
| DLIB .....                                          | 64       |
| FAR_F (segment) .....                               | 322      |
| FAR_HEAP (segment) .....                            | 322      |
| HEAP (segment) .....                                | 324      |
| HUGE_F (segment) .....                              | 324      |
| HUGE_HEAP (segment) .....                           | 325      |
| NEAR_F (segment) .....                              | 327      |
| NEAR_HEAP (segment) .....                           | 328      |
| placing .....                                       | 65       |
| TINY_F (segment) .....                              | 330      |
| TINY_HEAP (segment) .....                           | 330      |
| heap size                                           |          |
| and standard I/O .....                              | 65       |
| changing default .....                              | 64       |
| HEAP (segment) .....                                | 64, 324  |

|                                                           |     |
|-----------------------------------------------------------|-----|
| heap (zero-sized), implementation-defined behavior. . . . | 345 |
| hints                                                     |     |
| for good code generation . . . . .                        | 172 |
| implementation-defined behavior. . . . .                  | 338 |
| using efficient data types . . . . .                      | 160 |
| __huge (extended keyword) . . . . .                       | 255 |
| __hugeflash (extended keyword) . . . . .                  | 256 |
| HUGE_C (segment) . . . . .                                | 324 |
| HUGE_F (segment) . . . . .                                | 324 |
| HUGE_HEAP (segment) . . . . .                             | 325 |
| HUGE_I (segment) . . . . .                                | 325 |
| HUGE_ID (segment) . . . . .                               | 325 |
| HUGE_N (segment) . . . . .                                | 326 |
| HUGE_Z (segment) . . . . .                                | 326 |

## I

|                                                        |          |
|--------------------------------------------------------|----------|
| -I (compiler option). . . . .                          | 208      |
| IAR Command Line Build Utility. . . . .                | 83       |
| IAR Postlink (utility) . . . . .                       | 68       |
| IAR Systems Technical Support . . . . .                | 186      |
| iarbuild.exe (utility) . . . . .                       | 83       |
| __iar_cos_accuratef (library function) . . . . .       | 302      |
| __iar_cos_accuratel (library function) . . . . .       | 302      |
| __iar_pow_accuratef (library function). . . . .        | 302      |
| __iar_pow_accuratel (library function). . . . .        | 302      |
| __iar_sin_accuratef (library function). . . . .        | 302      |
| __iar_sin_accuratel (library function). . . . .        | 302      |
| __IAR_SYSTEMS_ICC__ (predefined symbol) . . . . .      | 296      |
| __iar_tan_accuratef (library function). . . . .        | 302      |
| __iar_tan_accuratel (library function). . . . .        | 302      |
| __ICCAVR__ (predefined symbol). . . . .                | 296      |
| iccbutl.h (library header file). . . . .               | 310      |
| icons, in this guide . . . . .                         | 22       |
| IDE                                                    |          |
| building a library from . . . . .                      | 83       |
| building applications from, an overview . . . . .      | 28       |
| identifiers, implementation-defined behavior . . . . . | 335, 350 |
| IEEE format, floating-point values . . . . .           | 235      |

|                                                             |          |
|-------------------------------------------------------------|----------|
| implementation-defined behavior . . . . .                   | 333      |
| C89 . . . . .                                               | 349      |
| important_typedef (pragma directive). . . . .               | 341, 355 |
| include files                                               |          |
| including before source files . . . . .                     | 219      |
| specifying . . . . .                                        | 182      |
| include_alias (pragma directive) . . . . .                  | 271      |
| __indirect_jump_to (intrinsic function) . . . . .           | 287      |
| infinity . . . . .                                          | 236      |
| infinity (style for printing), implementation-defined       |          |
| behavior . . . . .                                          | 344      |
| inheritance, in Embedded C++ . . . . .                      | 143      |
| initialization                                              |          |
| dynamic . . . . .                                           | 84, 113  |
| single-value . . . . .                                      | 142      |
| initialized data segments . . . . .                         | 59       |
| --initializers_in_flash (compiler option) . . . . .         | 208      |
| initializers, static . . . . .                              | 140      |
| INITTAB (segment) . . . . .                                 | 66, 326  |
| inline assembler . . . . .                                  | 119, 136 |
| avoiding . . . . .                                          | 173      |
| <i>See also</i> assembler language interface                |          |
| inline functions . . . . .                                  | 135      |
| in compiler. . . . .                                        | 170      |
| inline (pragma directive). . . . .                          | 272      |
| inlining functions, implementation-defined behavior . . . . | 339      |
| __insert_opcode (intrinsic function) . . . . .              | 287      |
| installation directory . . . . .                            | 21       |
| instantiate (pragma directive) . . . . .                    | 341, 355 |
| int (data type) signed and unsigned. . . . .                | 232      |
| integer types . . . . .                                     | 232      |
| casting . . . . .                                           | 238      |
| implementation-defined behavior. . . . .                    | 337      |
| intptr_t . . . . .                                          | 239      |
| ptrdiff_t . . . . .                                         | 239      |
| size_t . . . . .                                            | 239      |
| uintptr_t . . . . .                                         | 239      |
| integers, implementation-defined behavior . . . . .         | 351      |
| integral promotion . . . . .                                | 174      |
| internal error . . . . .                                    | 186      |

- `__interrupt` (extended keyword) . . . . . 49, 256
  - using in pragma directives . . . . . 282
- interrupt functions . . . . . 48
  - placement in memory . . . . . 66
- interrupt state, restoring . . . . . 290
- interrupt vector . . . . . 49
  - specifying with pragma directive . . . . . 282
- interrupt vector table . . . . . 49
  - in linker command file . . . . . 66
  - INITTAB segment . . . . . 326
  - INTVEC segment . . . . . 327
  - start address for . . . . . 49
- interrupts
  - disabling . . . . . 257
  - during function execution . . . . . 49
  - processor state . . . . . 44
  - using with EC++ destructors . . . . . 150
- `intptr_t` (integer type) . . . . . 239
- `__intrinsic` (extended keyword) . . . . . 257
- intrinsic functions . . . . . 173
  - overview . . . . . 117
  - summary . . . . . 283
- `intrinsics.h` (header file) . . . . . 283
- `inttypes.h` (library header file) . . . . . 304
- INTVEC (segment) . . . . . 66, 327
- `intwri.c` (library source code) . . . . . 113
- invocation syntax . . . . . 181
- `__io` (extended keyword) . . . . . 257
- `iomanip` (library header file) . . . . . 305
- `ios` (library header file) . . . . . 305
- `iosfwd` (library header file) . . . . . 305
- `iostream` (library header file) . . . . . 305
- `iso646.h` (library header file) . . . . . 304
- `istream` (library header file) . . . . . 305
- italic style, in this guide . . . . . 22
- iterator (STL header file) . . . . . 305
- I/O module, overriding in runtime library . . . . . 72, 81
- I/O, character-based . . . . . 111

## J

Josuttis, Nicolai M. . . . . 20

## K

- `keep_definition` (pragma directive) . . . . . 341, 355
- Kernighan & Ritchie function declarations . . . . . 174
  - disallowing . . . . . 221
- Kernighan, Brian W. . . . . 20
- keywords . . . . . 245
  - extended, overview of . . . . . 34

## L

- `-l` (compiler option) . . . . . 209
  - for creating skeleton code . . . . . 121
- labels . . . . . 141
  - assembler, making public . . . . . 220
  - `__program_start` . . . . . 85
- Labrosse, Jean J. . . . . 20
- `__lac` (intrinsic function) . . . . . 287
- Lajoie, Josée . . . . . 21
- language extensions
  - Embedded C++ . . . . . 143
  - enabling . . . . . 273
  - enabling (-e) . . . . . 203
- language overview . . . . . 27
- language (pragma directive) . . . . . 273
- `__las` (intrinsic function) . . . . . 288
- `__lat` (intrinsic function) . . . . . 288
- libraries
  - definition of . . . . . 28
  - standard template library . . . . . 305
  - using a prebuilt . . . . . 73
  - using a prebuilt (CLIB) . . . . . 110
- library configuration files
  - `dlavrlibname.h` . . . . . 83
  - DLIB . . . . . 88

|                                                           |          |
|-----------------------------------------------------------|----------|
| DLib_Defaults.h . . . . .                                 | 83, 88   |
| modifying . . . . .                                       | 83       |
| specifying . . . . .                                      | 202      |
| library documentation . . . . .                           | 301      |
| library features, missing from Embedded C++ . . . . .     | 144      |
| library functions . . . . .                               | 301      |
| for accessing flash . . . . .                             | 310      |
| summary, CLIB . . . . .                                   | 310      |
| summary, DLIB . . . . .                                   | 303      |
| memcmp_G . . . . .                                        | 311      |
| memcpy_G . . . . .                                        | 311      |
| memcpy_P . . . . .                                        | 311      |
| printf_P . . . . .                                        | 312      |
| puts_G . . . . .                                          | 312      |
| puts_P . . . . .                                          | 312      |
| scanf_P . . . . .                                         | 312      |
| sprintf_P . . . . .                                       | 312      |
| sscanf_P . . . . .                                        | 313      |
| strcat_G . . . . .                                        | 313      |
| strcmp_G . . . . .                                        | 313      |
| strcmp_P . . . . .                                        | 313      |
| strcpy_G . . . . .                                        | 313      |
| strcpy_P . . . . .                                        | 313      |
| strerror_P . . . . .                                      | 314      |
| strlen_G . . . . .                                        | 314      |
| strlen_P . . . . .                                        | 314      |
| strncat_G . . . . .                                       | 314      |
| strncmp_G . . . . .                                       | 314      |
| strncmp_P . . . . .                                       | 314      |
| strncpy_G . . . . .                                       | 315      |
| strncpy_P . . . . .                                       | 315      |
| library header files . . . . .                            | 301      |
| library modules . . . . .                                 |          |
| creating . . . . .                                        | 210      |
| overriding . . . . .                                      | 81       |
| library object files . . . . .                            | 302      |
| library options, setting . . . . .                        | 34       |
| library project template . . . . .                        | 33       |
| using . . . . .                                           | 83       |
| library_default_requirements (pragma directive) . . . . . | 355      |
| --library_module (compiler option) . . . . .              | 210      |
| library_provides (pragma directive) . . . . .             | 355      |
| library_requirement_override (pragma directive) . . . . . | 355      |
| lightbulb icon, in this guide . . . . .                   | 22       |
| limits.h (library header file) . . . . .                  | 304, 310 |
| __LINE__ (predefined symbol) . . . . .                    | 296      |
| linkage, C and C++ . . . . .                              | 125      |
| linker command file . . . . .                             | 54       |
| customizing . . . . .                                     | 54       |
| using the -P command . . . . .                            | 56       |
| using the -Z command . . . . .                            | 56       |
| linker map file . . . . .                                 | 67       |
| linker output files . . . . .                             | 29       |
| linker segment. <i>See</i> segment                        |          |
| linking . . . . .                                         |          |
| from the command line . . . . .                           | 29       |
| required input. . . . .                                   | 29       |
| Lippman, Stanley B. . . . .                               | 21       |
| list (STL header file). . . . .                           | 305      |
| listing, generating . . . . .                             | 209      |
| literals, compound. . . . .                               | 135      |
| literature, recommended . . . . .                         | 20       |
| __LITTLE_ENDIAN__ (predefined symbol). . . . .            | 296      |
| __load_program_memory (intrinsic function) . . . . .      | 288      |
| local variables, <i>See</i> auto variables                |          |
| locale . . . . .                                          |          |
| adding support for in library . . . . .                   | 95       |
| changing at runtime . . . . .                             | 95       |
| implementation-defined behavior. . . . .                  | 336, 346 |
| removing support for . . . . .                            | 94       |
| support for . . . . .                                     | 93       |
| locale.h (library header file) . . . . .                  | 304      |
| located data segments . . . . .                           | 65       |
| located data, declaring extern . . . . .                  | 166      |
| location (pragma directive) . . . . .                     | 165, 274 |
| LOCFRAME (assembler directive). . . . .                   | 132      |
| --lock_regs (compiler option) . . . . .                   | 210      |
| long double (data type) . . . . .                         | 235      |

- long float (data type), synonym for double . . . . . 141
- long long (data type)
  - avoiding . . . . . 161
  - restrictions . . . . . 233
- longjmp, restrictions for using . . . . . 303
- loop-invariant expressions. . . . . 171
- \_\_low\_level\_init . . . . . 85
  - customizing . . . . . 87
- low\_level\_init.c. . . . . 84, 113
- low-level processor operations . . . . . 137, 283
  - accessing . . . . . 117
- \_\_lseek (library function) . . . . . 93

## M

- m (compiler option). . . . . 211
- macros
  - embedded in #pragma optimize . . . . . 276
  - ERANGE (in errno.h) . . . . . 342, 356
  - inclusion of assert . . . . . 299
  - NULL, implementation-defined behavior . . . . . 343, 356
  - substituted in #pragma directives . . . . . 137
  - variadic . . . . . 135
- macro\_positions\_in\_diagnostics (compiler option) . . . . . 211
- main (function), definition . . . . . 349
- main (function), implementation-defined behavior . . . . . 334
- malloc (library function)
  - See also* heap . . . . . 45
  - implementation-defined behavior. . . . . 358, 361
- Mann, Bernhard . . . . . 21
- map (STL header file). . . . . 305
- map, linker . . . . . 67
- math functions rounding mode,
  - implementation-defined behavior . . . . . 346
- math.h (library header file) . . . . . 304, 310
- MB\_LEN\_MAX, implementation-defined behavior . . . . . 346
- \_\_medium\_write (library function). . . . . 112
- member functions, pointers to. . . . . 155
- memcpy\_G (library function) . . . . . 311

- memcpy\_G (library function) . . . . . 311
- memcpy\_P (library function) . . . . . 311
- memory
  - accessing . . . . . 39
  - allocating in C++ . . . . . 45
  - dynamic . . . . . 45
  - external . . . . . 226–227
  - flash . . . . . 226
  - heap . . . . . 45
  - non-initialized . . . . . 176
  - RAM, saving . . . . . 173
  - releasing in C++ . . . . . 45
  - stack. . . . . 44
    - saving . . . . . 173
  - used by global or static variables . . . . . 37
- memory attributes, summary. . . . . 40
- memory consumption, reducing . . . . . 112
- memory layout, AVR . . . . . 37
- memory management, type-safe . . . . . 143
- memory map
  - customizing the linker configuration file for . . . . . 54
- memory models . . . . . 38
  - configuration . . . . . 31
  - identifying (\_\_MEMORY\_MODEL\_\_) . . . . . 297
  - specifying on command line (--memory\_model) . . . . . 211
- memory placement
  - using pragma directive . . . . . 41
  - using type definitions. . . . . 41, 247
- memory segment. *See* segment
- memory types . . . . . 39
  - C++ . . . . . 43
  - hints . . . . . 162
  - placing variables in . . . . . 43
  - pointers . . . . . 41
  - specifying . . . . . 39
  - structures . . . . . 42
- memory (pragma directive). . . . . 341, 356
- memory (STL header file). . . . . 305
- \_\_MEMORY\_MODEL\_\_ (predefined symbol) . . . . . 297

|                                                                 |          |
|-----------------------------------------------------------------|----------|
| __memory_model (runtime model attribute) . . . . .              | 106      |
| --memory_model (compiler option) . . . . .                      | 211      |
| __memory_of, operator . . . . .                                 | 147      |
| message (pragma directive) . . . . .                            | 274      |
| messages                                                        |          |
| disabling . . . . .                                             | 223      |
| forcing . . . . .                                               | 274      |
| Meyers, Scott . . . . .                                         | 21       |
| --mfc (compiler option) . . . . .                               | 212      |
| migration, from earlier IAR compilers . . . . .                 | 20       |
| MISRA C                                                         |          |
| documentation . . . . .                                         | 20       |
| --misrac (compiler option) . . . . .                            | 191      |
| --misrac_verbose (compiler option) . . . . .                    | 191      |
| --misrac1998 (compiler option) . . . . .                        | 191      |
| --misrac2004 (compiler option) . . . . .                        | 191      |
| mode changing, implementation-defined behavior . . . . .        | 344      |
| module consistency . . . . .                                    | 103      |
| rtmodel . . . . .                                               | 278      |
| module map, in linker map file . . . . .                        | 67       |
| module name, specifying (--module_name) . . . . .               | 212      |
| module summary, in linker map file . . . . .                    | 67       |
| --module_name (compiler option) . . . . .                       | 212      |
| module_name (pragma directive) . . . . .                        | 341, 356 |
| __monitor (extended keyword) . . . . .                          | 257      |
| monitor functions . . . . .                                     | 49, 257  |
| multibyte character support . . . . .                           | 205      |
| multibyte characters, implementation-defined behavior . . . . . | 335, 347 |
| multiple address spaces, output for . . . . .                   | 68       |
| multiple inheritance                                            |          |
| in Extended EC++ . . . . .                                      | 144      |
| missing from Embedded C++ . . . . .                             | 143      |
| missing from STL . . . . .                                      | 144      |
| multiple output files, from XLINK . . . . .                     | 68       |
| __multiply_signed (intrinsic function) . . . . .                | 288      |
| __multiply_signed_with_unsigned (intrinsic function) . . . . .  | 288      |
| __multiply_unsigned (intrinsic function) . . . . .              | 288      |
| multi-file compilation . . . . .                                | 168      |
| mutable attribute, in Extended EC++ . . . . .                   | 144, 155 |

## N

|                                                |          |
|------------------------------------------------|----------|
| names block (call frame information) . . . . . | 133      |
| namespace support                              |          |
| in Extended EC++ . . . . .                     | 144, 155 |
| missing from Embedded C++ . . . . .            | 144      |
| naming conventions . . . . .                   | 22       |
| NaN                                            |          |
| implementation of . . . . .                    | 237      |
| implementation-defined behavior . . . . .      | 344      |
| native environment                             |          |
| implementation-defined behavior . . . . .      | 347      |
| NDEBUG (preprocessor symbol) . . . . .         | 299      |
| __near (extended keyword) . . . . .            | 258      |
| __nearfunc (extended keyword) . . . . .        | 258      |
| __nearfunc (function pointer) . . . . .        | 237      |
| NEAR_C (segment) . . . . .                     | 327      |
| NEAR_F (segment) . . . . .                     | 327      |
| NEAR_HEAP (segment) . . . . .                  | 328      |
| NEAR_I (segment) . . . . .                     | 328      |
| NEAR_ID (segment) . . . . .                    | 328      |
| NEAR_N (segment) . . . . .                     | 329      |
| NEAR_Z (segment) . . . . .                     | 329      |
| __nested (extended keyword) . . . . .          | 259      |
| new calling convention . . . . .               | 124      |
| new operator (extended EC++) . . . . .         | 149      |
| new (keyword) . . . . .                        | 45       |
| new (library header file) . . . . .            | 305      |
| non-initialized variables, hints for . . . . . | 176      |
| non-scalar parameters, avoiding . . . . .      | 173      |
| NOP (assembler instruction) . . . . .          | 289      |
| __noreturn (extended keyword) . . . . .        | 260      |
| Normal DLIB (library configuration) . . . . .  | 88       |
| Not a number (NaN) . . . . .                   | 237      |
| --no_clustering (compiler option) . . . . .    | 212      |
| --no_code_motion (compiler option) . . . . .   | 213      |
| --no_cross_call (compiler option) . . . . .    | 213      |
| --no_cse (compiler option) . . . . .           | 213      |
| __no_init (extended keyword) . . . . .         | 176, 259 |



--no\_inline (compiler option) . . . . . 214  
 \_\_no\_operation (intrinsic function) . . . . . 289  
 --no\_path\_in\_file\_macros (compiler option) . . . . . 214  
 no\_pch (pragma directive) . . . . . 341, 356  
 \_\_no\_rampd (runtime model attribute) . . . . . 106  
 --no\_rampd (compiler option) . . . . . 214  
 \_\_no\_runtime\_init (extended keyword) . . . . . 259  
 --no\_static\_destruction (compiler option) . . . . . 215  
 --no\_system\_include (compiler option) . . . . . 215  
 --no\_tbaa (compiler option) . . . . . 215  
 --no\_typedefs\_in\_diagnostics (compiler option) . . . . . 216  
 --no\_ubrof\_messages (compiler option) . . . . . 216  
 --no\_warnings (compiler option) . . . . . 217  
 --no\_wrap\_diagnostics (compiler option) . . . . . 217  
 NULL  
     implementation-defined behavior . . . . . 343  
     in library header file (CLIB) . . . . . 310  
     pointer constant, relaxation to Standard C . . . . . 140  
 NULL (macro), implementation-defined behavior . . . 356, 359  
 numeric conversion functions,  
     implementation-defined behavior . . . . . 347  
 numeric (STL header file) . . . . . 305

## O

-O (compiler option) . . . . . 217  
 -o (compiler option) . . . . . 218  
 object attributes . . . . . 248  
 object filename, specifying (-o) . . . . . 218  
 object module name, specifying (--module\_name) . . . . 212  
 object\_attribute (pragma directive) . . . . . 176, 275  
 offsetof . . . . . 310  
 old calling convention . . . . . 124  
 --omit\_types (compiler option) . . . . . 218  
 once (pragma directive) . . . . . 341, 356  
 --only\_stdout (compiler option) . . . . . 218  
 \_\_open (library function) . . . . . 93  
 operators

*See also @ (operator)*

for cast, in Extended EC++ . . . . . 144  
 for cast, missing from Embedded C++ . . . . . 144  
 for segment control . . . . . 139  
 in inline assembler . . . . . 136  
 new and delete . . . . . 149  
 precision for 32-bit float . . . . . 236  
 precision for 64-bit float . . . . . 236  
 sizeof, implementation-defined behavior . . . . . 346  
 variants for cast . . . . . 154  
 \_Pragma (preprocessor) . . . . . 135  
 \_\_ALIGNOF\_\_, for alignment control . . . . . 138  
 \_\_memory\_of\_\_ . . . . . 147  
 ?, language extensions for . . . . . 157  
 @ . . . . . 48  
 optimization  
     clustering, disabling . . . . . 212  
     code motion, disabling . . . . . 213  
     common sub-expression elimination, disabling . . . . 213  
     configuration . . . . . 32  
     disabling . . . . . 170  
     function inlining, disabling (--no\_inline) . . . . . 214  
     hints . . . . . 172  
     size, specifying . . . . . 229  
     specifying (-O) . . . . . 217  
     speed, specifying . . . . . 222  
     techniques . . . . . 170  
     type-based alias analysis, disabling (--tbaa) . . . . . 215  
     using inline assembler code . . . . . 119  
     using pragma directive . . . . . 275  
 optimization levels . . . . . 169  
 optimize (pragma directive) . . . . . 275  
 option parameters . . . . . 187  
 options, compiler. *See* compiler options  
 Oram, Andy . . . . . 20  
 ostream (library header file) . . . . . 305  
 output  
     from preprocessor . . . . . 219  
     multiple files from linker . . . . . 68  
     specifying for linker . . . . . 29

|                                     |     |
|-------------------------------------|-----|
| supporting non-standard. . . . .    | 113 |
| --output (compiler option). . . . . | 218 |
| overhead, reducing . . . . .        | 170 |

## P

|                                                                         |          |
|-------------------------------------------------------------------------|----------|
| pack (pragma directive) . . . . .                                       | 276      |
| parameters . . . . .                                                    |          |
| function . . . . .                                                      | 127      |
| hidden . . . . .                                                        | 127      |
| non-scalar, avoiding . . . . .                                          | 173      |
| register. . . . .                                                       | 127      |
| rules for specifying a file or directory . . . . .                      | 188      |
| specifying . . . . .                                                    | 189      |
| stack. . . . .                                                          | 127, 129 |
| typographic convention . . . . .                                        | 22       |
| part number, of this guide . . . . .                                    | 2        |
| permanent registers . . . . .                                           | 126      |
| perorr (library function),<br>implementation-defined behavior . . . . . | 358, 361 |
| placement . . . . .                                                     |          |
| code and data . . . . .                                                 | 317      |
| in named segments. . . . .                                              | 166      |
| plain char, implementation-defined behavior . . . . .                   | 336      |
| pointer types . . . . .                                                 | 237      |
| differences between . . . . .                                           | 42       |
| mixing . . . . .                                                        | 141      |
| using the best. . . . .                                                 | 162      |
| pointers . . . . .                                                      |          |
| casting . . . . .                                                       | 42, 238  |
| data . . . . .                                                          | 237      |
| function . . . . .                                                      | 237      |
| implementation-defined behavior. . . . .                                | 338, 353 |
| polymorphism, in Embedded C++ . . . . .                                 | 143      |
| porting, code containing pragma directives. . . . .                     | 266      |
| Postlink (utility) . . . . .                                            | 68       |
| postlink.htm . . . . .                                                  | 69       |
| pow (library function). . . . .                                         | 97       |
| alternative implementation of. . . . .                                  | 302      |

|                                                    |               |
|----------------------------------------------------|---------------|
| powXp (library function) . . . . .                 | 97            |
| pragma directives . . . . .                        | 34            |
| summary . . . . .                                  | 265           |
| basic_template_matching, using . . . . .           | 152           |
| for absolute located data . . . . .                | 165           |
| list of all recognized. . . . .                    | 340, 355      |
| pack . . . . .                                     | 276           |
| type_attribute, using. . . . .                     | 41            |
| _Pragma (preprocessor operator) . . . . .          | 135           |
| precision arguments, library support for . . . . . | 112           |
| predefined symbols . . . . .                       |               |
| overview . . . . .                                 | 35            |
| summary . . . . .                                  | 294           |
| --predef_macro (compiler option). . . . .          | 219           |
| --preinclude (compiler option) . . . . .           | 219           |
| --preprocess (compiler option) . . . . .           | 219           |
| preprocessor . . . . .                             |               |
| operator (_Pragma) . . . . .                       | 135           |
| output. . . . .                                    | 219           |
| overview of . . . . .                              | 293           |
| preprocessor directives . . . . .                  |               |
| comments at the end of . . . . .                   | 141           |
| implementation-defined behavior. . . . .           | 340, 354      |
| #pragma . . . . .                                  | 265           |
| preprocessor extensions . . . . .                  |               |
| __VA_ARGS__ . . . . .                              | 135           |
| #warning message . . . . .                         | 299           |
| preprocessor symbols . . . . .                     | 294           |
| defining . . . . .                                 | 197           |
| preserved registers . . . . .                      | 126           |
| __PRETTY_FUNCTION__ (predefined symbol). . . . .   | 297           |
| primitives, for special functions . . . . .        | 48            |
| print formatter, selecting. . . . .                | 76            |
| printf (library function). . . . .                 | 76, 112       |
| choosing formatter. . . . .                        | 76            |
| configuration symbols . . . . .                    | 91            |
| customizing . . . . .                              | 113           |
| implementation-defined behavior. . . . .           | 344, 357, 361 |
| selecting. . . . .                                 | 112           |

\_\_printf\_args (pragma directive) . . . . . 277  
 printf\_P (library function) . . . . . 312  
 printing characters, implementation-defined behavior . . . 347  
 processor configuration . . . . . 30  
 processor operations  
     accessing . . . . . 117  
     low-level . . . . . 137, 283  
 program entry label . . . . . 85  
 program termination, implementation-defined behavior . . 334  
 programming hints . . . . . 172  
 \_\_program\_start (label) . . . . . 85  
 projects  
     basic settings for . . . . . 29  
     setting up for a library . . . . . 83  
 prototypes, enforcing . . . . . 221  
 ptrdiff\_t (integer type) . . . . . 239, 310  
 PUBLIC (assembler directive) . . . . . 220  
 publication date, of this guide . . . . . 2  
 --public\_equ (compiler option) . . . . . 220  
 public\_equ (pragma directive) . . . . . 341, 356  
 putchar (library function) . . . . . 111  
 putenv (library function), absent from DLIB . . . . . 95  
 puts\_G (library function) . . . . . 312  
 puts\_P (library function) . . . . . 312  
 putw, in stdio.h . . . . . 308

## Q

QCCAVR (environment variable) . . . . . 182  
 qualifiers  
     const and volatile . . . . . 240  
     implementation-defined behavior . . . . . 339, 354  
 queue (STL header file) . . . . . 306

## R

-r (compiler option) . . . . . 197  
 raise (library function), configuring support for . . . . . 96  
 raise.c . . . . . 96

RAM  
     non-zero initialized variables . . . . . 59  
     saving memory . . . . . 173  
 RAMPZ (register) . . . . . 194  
 range errors, in linker . . . . . 67  
 \_\_raw (extended keyword) . . . . . 260  
 \_\_read (library function) . . . . . 93  
     customizing . . . . . 89  
 read formatter, selecting . . . . . 77, 113  
 reading guidelines . . . . . 17  
 reading, recommended . . . . . 20  
 realloc (library function) . . . . . 45  
     implementation-defined behavior . . . . . 358, 361  
     *See also* heap  
 recursive functions  
     avoiding . . . . . 173  
     storing data on stack . . . . . 44  
 reentrancy (DLIB) . . . . . 302  
 reference information, typographic convention . . . . . 22  
 register keyword, implementation-defined behavior . . . . 338  
 register parameters . . . . . 127  
 registered trademarks . . . . . 2  
 registers  
     callee-save, stored on stack . . . . . 44  
     for function returns . . . . . 130  
     implementation-defined behavior . . . . . 353  
     in assembler-level routines . . . . . 123  
     preserved . . . . . 126  
     scratch . . . . . 126  
     \_\_regvar (extended keyword) . . . . . 260  
 reinterpret\_cast (cast operator) . . . . . 144  
 --relaxed\_fp (compiler option) . . . . . 220  
 remark (diagnostic message) . . . . . 185  
     classifying for compiler . . . . . 199  
     enabling in compiler . . . . . 221  
     --remarks (compiler option) . . . . . 221  
 remove (library function) . . . . . 93  
     implementation-defined behavior . . . . . 344, 357, 360  
 remquo, magnitude of . . . . . 342

|                                                          |               |
|----------------------------------------------------------|---------------|
| rename (library function) . . . . .                      | 93            |
| implementation-defined behavior. . . . .                 | 344, 357, 360 |
| __ReportAssert (library function) . . . . .              | 98            |
| __require (intrinsic function) . . . . .                 | 289           |
| required (pragma directive) . . . . .                    | 277           |
| --require_prototypes (compiler option) . . . . .         | 221           |
| __restore_interrupt (intrinsic function) . . . . .       | 289–290       |
| return address stack, changing default size of . . . . . | 63            |
| return addresses . . . . .                               | 130           |
| return data stack                                        |               |
| reducing usage of . . . . .                              | 196           |
| using cross-call optimizations . . . . .                 | 213           |
| return values, from functions . . . . .                  | 130           |
| Ritchie, Dennis M. . . . .                               | 20            |
| __root (extended keyword) . . . . .                      | 261           |
| --root_variables (compiler option) . . . . .             | 221           |
| routines, time-critical . . . . .                        | 117, 137, 283 |
| RSTACK (segment) . . . . .                               | 329           |
| example . . . . .                                        | 63            |
| <i>See also</i> stack                                    |               |
| rtmodel (assembler directive) . . . . .                  | 105           |
| rtmodel (pragma directive) . . . . .                     | 278           |
| rtti support, missing from STL . . . . .                 | 144           |
| __rt_version (runtime model attribute) . . . . .         | 105           |
| runtime environment                                      |               |
| CLIB . . . . .                                           | 109           |
| DLIB . . . . .                                           | 71            |
| setting options for . . . . .                            | 34            |
| setting up (DLIB) . . . . .                              | 72            |
| runtime libraries (CLIB)                                 |               |
| introduction . . . . .                                   | 301           |
| filename syntax . . . . .                                | 110           |
| setting up from command line . . . . .                   | 33            |
| setting up from IDE . . . . .                            | 33            |
| using a prebuilt . . . . .                               | 110           |
| runtime libraries (DLIB)                                 |               |
| introduction . . . . .                                   | 301           |
| customizing system startup code . . . . .                | 87            |
| customizing without rebuilding . . . . .                 | 75            |

|                                                               |     |
|---------------------------------------------------------------|-----|
| filename syntax . . . . .                                     | 74  |
| overriding modules in . . . . .                               | 81  |
| setting up from command line . . . . .                        | 33  |
| setting up from IDE . . . . .                                 | 33  |
| using a prebuilt . . . . .                                    | 73  |
| runtime model attributes . . . . .                            | 103 |
| runtime model definitions . . . . .                           | 278 |
| runtime type information, missing from Embedded C++ . . . . . | 143 |

## S

|                                                 |               |
|-------------------------------------------------|---------------|
| -s (compiler option) . . . . .                  | 222           |
| __save_interrupt (intrinsic function) . . . . . | 290           |
| scanf (library function) . . . . .              | 113           |
| choosing formatter (DLIB) . . . . .             | 77            |
| configuration symbols . . . . .                 | 91            |
| implementation-defined behavior. . . . .        | 344, 358, 361 |
| __scanf_args (pragma directive) . . . . .       | 279           |
| scanf_P (library function) . . . . .            | 312           |
| scratch registers . . . . .                     | 126           |
| section (pragma directive) . . . . .            | 279           |
| segment group name . . . . .                    | 57            |
| segment map, in linker map file . . . . .       | 67            |
| segment memory types, in XLINK . . . . .        | 54            |
| segment (pragma directive) . . . . .            | 279           |
| segments . . . . .                              | 317           |
| code . . . . .                                  | 65            |
| data . . . . .                                  | 57            |
| definition of . . . . .                         | 53            |
| initialized data . . . . .                      | 59            |
| introduction . . . . .                          | 53            |
| located data . . . . .                          | 65            |
| naming . . . . .                                | 58            |
| packing in memory . . . . .                     | 56            |
| placing in sequence . . . . .                   | 56            |
| static memory . . . . .                         | 57            |
| summary . . . . .                               | 317           |
| too long for address range . . . . .            | 67            |
| too long, in linker. . . . .                    | 67            |

- declaring (#pragma segment) . . . . . 279
- HEAP . . . . . 64
- INITTAB . . . . . 66
- RSTACK
  - reducing usage of . . . . . 196
  - using cross-call optimizations . . . . . 213
- specifying (--segment) . . . . . 222
- SWITCH . . . . . 66
- \_\_segment\_begin (extended operator) . . . . . 139
- \_\_segment\_end (extended operator) . . . . . 139
- \_\_segment\_size (extended operator) . . . . . 139
- SEI (assembler instruction) . . . . . 285
- semaphores
  - C example . . . . . 50
  - operations on . . . . . 257
- separate\_cluster\_for\_initialized\_variables (compiler option) . . . . . 223
- set (STL header file) . . . . . 306
- setjmp.h (library header file) . . . . . 304, 310
- setlocale (library function) . . . . . 95
- settings, basic for project configuration . . . . . 29
- \_\_set\_interrupt\_state (intrinsic function) . . . . . 290
- severity level, of diagnostic messages . . . . . 185
  - specifying . . . . . 186
- SFR
  - accessing special function registers . . . . . 176
  - declaring extern special function registers . . . . . 166
- shared object . . . . . 184
- short (data type) . . . . . 232
- signal (library function)
  - configuring support for . . . . . 96
  - implementation-defined behavior . . . . . 342, 357
- signals, implementation-defined behavior . . . . . 334
  - at system startup . . . . . 335
- signal.c . . . . . 96
- signal.h (library header file) . . . . . 304
- signed char (data type) . . . . . 232–233
  - specifying . . . . . 195
- signed int (data type) . . . . . 232
- signed long long (data type) . . . . . 232
- signed long (data type) . . . . . 232
- signed short (data type) . . . . . 232
- signed values, avoiding . . . . . 160
- silent (compiler option) . . . . . 223
- silent operation, specifying in compiler . . . . . 223
- sin (library function) . . . . . 302
- 64-bits (floating-point format) . . . . . 236
- size optimization, specifying . . . . . 229
- size\_t (integer type) . . . . . 239, 310
- skeleton code, creating for assembler language interface . 120
- skeleton.s90 (assembler source output) . . . . . 121
- SLEEP (assembler instruction) . . . . . 291
- slist (STL header file) . . . . . 306
- \_small\_write (library function) . . . . . 112
- \_\_software\_interrupt (intrinsic function) . . . . . 291
- source files, list all referred . . . . . 207
- space characters, implementation-defined behavior . . . . 343
- special function registers (SFR) . . . . . 176
- special function types . . . . . 48
  - overview . . . . . 35
- speed optimization, specifying . . . . . 222
- spmc\_r\_address (compiler option) . . . . . 224
- sprintf (library function) . . . . . 76, 112
  - choosing formatter . . . . . 76
  - customizing . . . . . 113
- sprintf\_P (library function) . . . . . 312
- sscanf (library function) . . . . . 113
  - choosing formatter (DLIB) . . . . . 77
- sscanf\_P (library function) . . . . . 313
- sstream (library header file) . . . . . 305
- stack . . . . . 44, 61
  - advantages and problems using . . . . . 44
  - changing default size of . . . . . 62
  - cleaning after function return . . . . . 130
  - contents of . . . . . 44
  - layout . . . . . 129
  - saving space . . . . . 173
  - size . . . . . 62
- stack parameters . . . . . 127, 129

|                                             |               |                                              |               |
|---------------------------------------------|---------------|----------------------------------------------|---------------|
| stack pointer                               | 44            | stdin                                        | 93            |
| stack pointer register, considerations      | 126           | implementation-defined behavior              | 357, 360      |
| stack segment                               |               | stdint.h (library header file)               | 304, 306      |
| CSTACK                                      | 319           | stdio.h (library header file)                | 304, 310      |
| placing in memory                           | 62–63         | stdio.h, additional C functionality          | 307           |
| RSTACK                                      | 329           | stdlib.h (library header file)               | 304, 310      |
| stack (STL header file)                     | 306           | stdout                                       | 93, 218       |
| Standard C                                  |               | implementation-defined behavior              | 343, 357, 360 |
| library compliance with                     | 32, 301       | Steele, Guy L.                               | 20            |
| specifying strict usage                     | 224           | STL                                          | 153           |
| standard error, redirecting in compiler     | 218           | strcascmp, in string.h                       | 308           |
| standard input                              | 89            | strcat_G (library function)                  | 313           |
| standard output                             | 89            | strcmp_G (library function)                  | 313           |
| specifying in compiler                      | 218           | strcmp_P (library function)                  | 313           |
| standard template library (STL)             |               | strcpy_G (library function)                  | 313           |
| in Extended EC++                            | 144, 153, 305 | strcpy_P (library function)                  | 313           |
| missing from Embedded C++                   | 144           | strdup, in string.h                          | 308           |
| startup system. <i>See</i> system startup   |               | streambuf (library header file)              | 305           |
| statements, implementation-defined behavior | 354           | streams                                      |               |
| static clustering (compiler transformation) | 172           | implementation-defined behavior              | 334           |
| static data, in linker command file         | 61            | supported in Embedded C++                    | 144           |
| static memory segments                      | 57            | strerror (library function)                  |               |
| static overlay                              | 132           | implementation-defined behavior              | 359, 362      |
| static variables                            | 37            | behavior                                     | 348           |
| initialization                              | 59            | strerror_P (library function)                | 314           |
| taking the address of                       | 173           | --strict (compiler option)                   | 224           |
| static_assert()                             | 139           | string (library header file)                 | 305           |
| static_cast (cast operator)                 | 144           | strings, supported in Embedded C++           | 144           |
| status flags for floating-point             | 307           | --string_literals_in_flash (compiler option) | 224           |
| std namespace, missing from EC++            |               | string.h (library header file)               | 304, 310      |
| and Extended EC++                           | 155           | string.h, additional C functionality         | 308           |
| stdarg.h (library header file)              | 304, 310      | strlen_G (library function)                  | 314           |
| stdbool.h (library header file)             | 232, 304, 310 | strlen_P (library function)                  | 314           |
| __STDC__ (predefined symbol)                | 297           | strncasecmp, in string.h                     | 308           |
| STDC CX_LIMITED_RANGE (pragma directive)    | 280           | strncat_G (library function)                 | 314           |
| STDC FENV_ACCESS (pragma directive)         | 280           | strncmp_G (library function)                 | 314           |
| STDC FP_CONTRACT (pragma directive)         | 281           | strcmp_P (library function)                  | 314           |
| __STDC_VERSION__ (predefined symbol)        | 297           | strncpy_G (library function)                 | 315           |
| stddef.h (library header file)              | 233, 304, 310 | strncpy_P (library function)                 | 315           |
| stderr                                      | 93, 218       |                                              |               |

strlen, in string.h . . . . . 308  
 Stroustrup, Bjarne . . . . . 21  
 strtstream (library header file) . . . . . 305  
 strtod (library function), configuring support for . . . . . 97  
 structure types  
     alignment . . . . . 240  
     layout of . . . . . 240  
 structures  
     aligning . . . . . 277  
     anonymous . . . . . 139, 162  
     implementation-defined behavior . . . . . 339, 353  
     placing in memory type . . . . . 42  
 subnormal numbers . . . . . 237  
 \_\_SUBVERSION\_\_ (predefined symbol) . . . . . 297  
 support, technical . . . . . 186  
 Sutter, Herb . . . . . 21  
 \_\_swap\_nibbles (intrinsic function) . . . . . 291  
 SWITCH (segment) . . . . . 66, 329  
 symbols  
     anonymous, creating . . . . . 135  
     including in output . . . . . 277  
     listing in linker map file . . . . . 67  
     overview of predefined . . . . . 35  
     preprocessor, defining . . . . . 197  
 syntax  
     command line options . . . . . 187  
     extended keywords . . . . . 41, 246–249  
     invoking compiler . . . . . 181  
 system function, implementation-defined behavior . . 335, 345  
 system startup  
     CLIB . . . . . 114  
     customizing . . . . . 87  
     DLIB . . . . . 84  
 system termination  
     CLIB . . . . . 114  
     C-SPY interface to . . . . . 87  
     DLIB . . . . . 86  
 system (library function)  
     implementation-defined behavior . . . . . 358, 361

system (library function), configuring support for . . . . . 95  
 system\_include (pragma directive) . . . . . 341, 356  
 --system\_include\_dir (compiler option) . . . . . 225

## T

tan (library function) . . . . . 302  
 \_\_task (extended keyword) . . . . . 261  
 technical support, IAR Systems . . . . . 186  
 template support  
     in Extended EC++ . . . . . 144, 151  
     missing from Embedded C++ . . . . . 143  
 Terminal I/O window  
     making available (CLIB) . . . . . 115  
     making available (DLIB) . . . . . 79–80  
     not supported when . . . . . 81  
 terminal I/O, debugger runtime interface for . . . . . 79  
 terminal output, speeding up . . . . . 79, 81  
 termination of system. *See* system termination  
 termination status, implementation-defined behavior . . 345  
 terminology . . . . . 21  
 tgmth.h (library header file) . . . . . 304  
 32-bits (floating-point format) . . . . . 236  
 this (pointer) . . . . . 122  
     class memory . . . . . 146  
     referring to a class object . . . . . 146  
 \_\_TID\_\_ (predefined symbol) . . . . . 298  
 \_\_TIME\_\_ (predefined symbol) . . . . . 297  
 time zone (library function)  
     implementation-defined behavior . . . . . 359, 362  
 time zone (library function), implementation-defined  
     behavior . . . . . 345  
 time-critical routines . . . . . 117, 137, 283  
 time.c . . . . . 97  
 time.h (library header file) . . . . . 304  
     additional C functionality . . . . . 308  
 time32 (library function), configuring support for . . . . . 97  
 time64 (library function), configuring support for . . . . . 97  
 \_\_tiny (extended keyword) . . . . . 262

|                                                                |          |
|----------------------------------------------------------------|----------|
| __tinyflash (extended keyword) . . . . .                       | 262      |
| __TINY_AVR__ (predefined symbol) . . . . .                     | 298      |
| TINY_F (segment) . . . . .                                     | 330      |
| TINY_HEAP (segment) . . . . .                                  | 330      |
| TINY_I (segment) . . . . .                                     | 330      |
| TINY_ID (segment) . . . . .                                    | 331      |
| TINY_N (segment) . . . . .                                     | 331      |
| TINY_Z (segment) . . . . .                                     | 331      |
| tips, programming . . . . .                                    | 172      |
| tools icon, in this guide . . . . .                            | 22       |
| trademarks . . . . .                                           | 2        |
| transformations, compiler . . . . .                            | 168      |
| translation, implementation-defined behavior . . . . .         | 333, 349 |
| trap vectors, specifying with pragma directive . . . . .       | 282      |
| type attributes . . . . .                                      | 245      |
| specifying . . . . .                                           | 281      |
| type definitions, used for specifying memory storage . 41, 247 |          |
| type information, omitting . . . . .                           | 218      |
| type qualifiers . . . . .                                      |          |
| const and volatile . . . . .                                   | 240      |
| implementation-defined behavior. . . . .                       | 339, 354 |
| typedefs . . . . .                                             |          |
| excluding from diagnostics . . . . .                           | 216      |
| repeated . . . . .                                             | 141      |
| type_attribute (pragma directive) . . . . .                    | 41, 281  |
| type-based alias analysis (compiler transformation) . . . . .  | 171      |
| disabling . . . . .                                            | 215      |
| type-safe memory management . . . . .                          | 143      |
| typographic conventions . . . . .                              | 22       |

## U

### UBROF

|                                                             |     |
|-------------------------------------------------------------|-----|
| format of linkable object files . . . . .                   | 183 |
| specifying, example of . . . . .                            | 29  |
| uchar.h (library header file) . . . . .                     | 304 |
| uintptr_t (integer type) . . . . .                          | 239 |
| underflow errors, implementation-defined behavior . . . . . | 342 |

|                                                             |          |
|-------------------------------------------------------------|----------|
| underflow range errors, . . . . .                           |          |
| implementation-defined behavior . . . . .                   | 356, 360 |
| __ungetchar, in stdio.h . . . . .                           | 308      |
| unions . . . . .                                            |          |
| anonymous . . . . .                                         | 139, 162 |
| implementation-defined behavior. . . . .                    | 339, 353 |
| universal character names, implementation-defined . . . . . |          |
| behavior . . . . .                                          | 340      |
| unsigned char (data type) . . . . .                         | 232–233  |
| changing to signed char . . . . .                           | 195      |
| unsigned int (data type) . . . . .                          | 232      |
| unsigned long long (data type) . . . . .                    | 232      |
| unsigned long (data type) . . . . .                         | 232      |
| unsigned short (data type) . . . . .                        | 232      |
| --use_c++_inline (compiler option) . . . . .                | 225      |
| utility (STL header file) . . . . .                         | 306      |

## V

|                                                                |         |
|----------------------------------------------------------------|---------|
| -v (compiler option) . . . . .                                 | 226     |
| variable type information, omitting in object output . . . . . | 218     |
| variables . . . . .                                            |         |
| auto . . . . .                                                 | 44      |
| defined inside a function . . . . .                            | 44      |
| global . . . . .                                               |         |
| initialization of . . . . .                                    | 59      |
| placement in memory . . . . .                                  | 37      |
| hints for choosing . . . . .                                   | 173     |
| local. <i>See</i> auto variables . . . . .                     |         |
| non-initialized . . . . .                                      | 176     |
| omitting type info . . . . .                                   | 218     |
| placing at absolute addresses . . . . .                        | 166     |
| placing in named segments . . . . .                            | 166     |
| static . . . . .                                               |         |
| placement in memory . . . . .                                  | 37      |
| taking the address of . . . . .                                | 173     |
| static and global, initializing . . . . .                      | 59      |
| variadic macros . . . . .                                      | 139     |
| vector (pragma directive) . . . . .                            | 49, 282 |



vector (STL header file) . . . . . 306  
 \_\_VER\_\_ (predefined symbol) . . . . . 298  
 version  
   compiler subversion number . . . . . 297  
   IAR Embedded Workbench . . . . . 2  
   of compiler. . . . . 298  
 \_\_version\_1 (extended keyword) . . . . . 263  
 \_\_VERSION\_1\_CALLS\_\_ (predefined symbol) . . . . . 298  
 --version1\_calls (compiler option) . . . . . 227  
 --vla (compiler option) . . . . . 228  
 void, pointers to . . . . . 140  
 volatile  
   and const, declaring objects . . . . . 242  
   declaring objects . . . . . 240  
   protecting simultaneously accesses variables . . . . . 175  
   rules for access . . . . . 241

## W

#warning message (preprocessor extension) . . . . . 299  
 warnings . . . . . 185  
   classifying in compiler . . . . . 200  
   disabling in compiler . . . . . 217  
   exit code in compiler . . . . . 228  
 warnings icon, in this guide . . . . . 22  
 warnings (pragma directive) . . . . . 341, 356  
 --warnings\_affect\_exit\_code (compiler option) . . . . 184, 228  
 --warnings\_are\_errors (compiler option) . . . . . 228  
 watchdog reset instruction . . . . . 291  
 \_\_watchdog\_reset (intrinsic function) . . . . . 291  
 wchar\_t (data type), adding support for in C . . . . . 233  
 wchar.h (library header file) . . . . . 304, 307  
 wctype.h (library header file) . . . . . 304  
 web sites, recommended . . . . . 21  
 white-space characters, implementation-defined behavior 333  
 \_\_write (library function) . . . . . 93  
   customizing . . . . . 89  
 write formatter, selecting . . . . . 112–113  
 \_\_write\_array, in stdio.h . . . . . 308

\_\_write\_buffered (DLIB library function) . . . . . 79, 81

## X

\_\_xch (intrinsic function) . . . . . 291  
 XLINK errors  
   range error . . . . . 67  
   segment too long . . . . . 67  
 XLINK options  
   -O . . . . . 68  
   -y . . . . . 68  
 XLINK segment memory types . . . . . 54  
 \_\_XMEGA\_CORE\_\_ (predefined symbol) . . . . . 298  
 xreportassert.c . . . . . 98

## Y

-y (compiler option) . . . . . 229

## Z

-z (compiler option) . . . . . 229  
 --zero\_register (compiler option) . . . . . 230

# Symbols

\_\_Exit (library function) . . . . . 86  
 \_\_exit (library function) . . . . . 86  
 \_\_exit (system termination code)  
   in CLIB . . . . . 113  
   in DLIB . . . . . 84  
 \_\_formatted\_write (library function) . . . . . 112  
 \_\_medium\_write (library function) . . . . . 112  
 \_\_small\_write (library function) . . . . . 112  
 \_\_ALIGNOF\_\_ (operator) . . . . . 138  
 \_\_asm (language extension) . . . . . 136  
 \_\_assignment\_by\_bitwise\_copy\_allowed, symbol used in library . . . . . 309  
 \_\_BASE\_FILE\_\_ (predefined symbol) . . . . . 294

|                                                                                        |          |                                                                   |          |
|----------------------------------------------------------------------------------------|----------|-------------------------------------------------------------------|----------|
| <code>__BUILD_NUMBER__</code> (predefined symbol) . . . . .                            | 294      | <code>__get_interrupt_state</code> (intrinsic function) . . . . . | 287      |
| <code>__close</code> (library function) . . . . .                                      | 93       | <code>__has_constructor</code> , symbol used in library . . . . . | 309      |
| <code>__constrange()</code> , symbol used in library . . . . .                         | 309      | <code>__has_destructor</code> , symbol used in library . . . . .  | 309      |
| <code>__construction_by_bitwise_copy_allowed</code> , symbol used in library . . . . . | 309      | <code>__HAS_EEPROM__</code> (predefined symbol) . . . . .         | 295      |
| <code>__CORE__</code> (predefined symbol). . . . .                                     | 294      | <code>__HAS_EIND__</code> (predefined symbol) . . . . .           | 295      |
| <code>__cplusplus</code> (predefined symbol) . . . . .                                 | 294      | <code>__HAS_ELPM__</code> (predefined symbol). . . . .            | 295      |
| <code>__cpu</code> (runtime model attribute) . . . . .                                 | 105      | <code>__HAS_ENHANCED_CORE__</code> (predefined symbol) . . . . .  | 295      |
| <code>__cpu_name</code> (runtime model attribute). . . . .                             | 105      | <code>__HAS_FISCR__</code> (predefined symbol) . . . . .          | 296      |
| <code>__CPU__</code> (predefined symbol) . . . . .                                     | 294      | <code>__HAS_MUL__</code> (predefined symbol). . . . .             | 296      |
| <code>__DATE__</code> (predefined symbol) . . . . .                                    | 294      | <code>__HAS_RAMPD__</code> (predefined symbol) . . . . .          | 296      |
| <code>__delay_cycles</code> (intrinsic function) . . . . .                             | 284      | <code>__HAS_RAMPX__</code> (predefined symbol) . . . . .          | 296      |
| <code>__DES_decryption</code> (intrinsic function) . . . . .                           | 284      | <code>__HAS_RAMPY__</code> (predefined symbol) . . . . .          | 296      |
| <code>__DES_encryption</code> (intrinsic function) . . . . .                           | 285      | <code>__HAS_RAMPZ__</code> (predefined symbol) . . . . .          | 296      |
| <code>__device__</code> (predefined symbol). . . . .                                   | 294      | <code>__huge</code> (extended keyword) . . . . .                  | 238, 255 |
| <code>__disable_interrupt</code> (intrinsic function). . . . .                         | 285      | <code>__hugeflash</code> (extended keyword) . . . . .             | 238, 256 |
| <code>__DLIB_FILE_DESCRIPTOR</code> (configuration symbol) . . . . .                   | 92       | <code>__huge_size_t</code> . . . . .                              | 150      |
| <code>__double_size</code> (runtime model attribute). . . . .                          | 105      | <code>__IAR_SYSTEMS_ICC__</code> (predefined symbol) . . . . .    | 296      |
| <code>__DOUBLE__</code> (predefined symbol) . . . . .                                  | 294      | <code>__ICCAVR__</code> (predefined symbol). . . . .              | 296      |
| <code>__eeprom</code> (extended keyword) . . . . .                                     | 238, 250 | <code>__indirect_jump_to</code> (intrinsic function) . . . . .    | 287      |
| <code>__embedded_cplusplus</code> (predefined symbol) . . . . .                        | 295      | <code>__insert_opcode</code> (intrinsic function) . . . . .       | 287      |
| <code>__enable_interrupt</code> (intrinsic function) . . . . .                         | 285      | <code>__interrupt</code> (extended keyword) . . . . .             | 49, 256  |
| <code>__enhanced_core</code> (runtime model attribute) . . . . .                       | 106      | using in pragma directives . . . . .                              | 282      |
| <code>__exit</code> (library function) . . . . .                                       | 86       | <code>__intrinsic</code> (extended keyword). . . . .              | 257      |
| <code>__extended_load_program_memory</code> (intrinsic function) . . . . .             | 286      | <code>__io</code> (extended keyword). . . . .                     | 257      |
| <code>__ext_io</code> (extended keyword) . . . . .                                     | 251      | <code>__lac</code> (intrinsic function). . . . .                  | 287      |
| <code>__far</code> (extended keyword) . . . . .                                        | 238, 252 | <code>__las</code> (intrinsic function). . . . .                  | 288      |
| <code>__farflash</code> (extended keyword) . . . . .                                   | 238, 252 | <code>__lat</code> (intrinsic function) . . . . .                 | 288      |
| <code>__farfunc</code> (extended keyword) . . . . .                                    | 253      | <code>__LINE__</code> (predefined symbol) . . . . .               | 296      |
| <code>__farfunc</code> (function pointer) . . . . .                                    | 237      | <code>__LITTLE_ENDIAN__</code> (predefined symbol). . . . .       | 296      |
| <code>__FILE__</code> (predefined symbol). . . . .                                     | 295      | <code>__load_program_memory</code> (intrinsic function) . . . . . | 288      |
| <code>__flash</code> (extended keyword) . . . . .                                      | 238, 253 | <code>__low_level_init</code> . . . . .                           | 85       |
| <code>__fractional_multiply_signed</code> (intrinsic function). . . . .                | 286      | <code>__low_level_init</code> , customizing . . . . .             | 87       |
| <code>__fractional_multiply_signed_with_unsigned</code> (intrinsic function) . . . . . | 286      | <code>__lseek</code> (library function) . . . . .                 | 93       |
| <code>__fractional_multiply_unsigned</code> (intrinsic function). . . . .              | 286      | <code>__memory_model</code> (runtime model attribute). . . . .    | 106      |
| <code>__FUNCTION__</code> (predefined symbol). . . . .                                 | 142, 295 | <code>__MEMORY_MODEL__</code> (predefined symbol) . . . . .       | 297      |
| <code>__func__</code> (predefined symbol) . . . . .                                    | 142, 295 | <code>__memory_of</code> , operator. . . . .                      | 147      |
| <code>__generic</code> (extended keyword) . . . . .                                    | 238, 254 | <code>__memory_of</code> , symbol used in library . . . . .       | 309      |
| <code>__gets</code> , in <code>stdio.h</code> . . . . .                                | 307      | <code>__monitor</code> (extended keyword) . . . . .               | 257      |
|                                                                                        |          | <code>__multiply_signed</code> (intrinsic function) . . . . .     | 288      |

- `__multiply_signed_with_unsigned` (intrinsic function) . . . 288
- `__multiply_unsigned` (intrinsic function) . . . . . 288
- `__near` (extended keyword) . . . . . 238, 258
- `__nearfunc` (extended keyword) . . . . . 258
- `__nearfunc` (function pointer) . . . . . 237
- `__nested` (extended keyword) . . . . . 259
- `__noreturn` (extended keyword) . . . . . 260
- `__no_init` (extended keyword) . . . . . 176, 259
- `__no_operation` (intrinsic function) . . . . . 289
- `__no_rampd` (runtime model attribute) . . . . . 106
- `__no_runtime_init` (extended keyword) . . . . . 259
- `__open` (library function) . . . . . 93
- `__PRETTY_FUNCTION__` (predefined symbol) . . . . . 297
- `__printf_args` (pragma directive) . . . . . 277
- `__program_start` (label) . . . . . 85
- `__raw` (extended keyword) . . . . . 260
- `__read` (library function) . . . . . 93
  - customizing . . . . . 89
- `__regvar` (extended keyword) . . . . . 260
- `__ReportAssert` (library function) . . . . . 98
- `__require` (intrinsic function) . . . . . 289
- `__require`, adding . . . . . 204
- `__restore_interrupt` (intrinsic function) . . . . . 289–290
- `__root` (extended keyword) . . . . . 221, 261
- `__rt_version` (runtime model attribute) . . . . . 105
- `__save_interrupt` (intrinsic function) . . . . . 290
- `__scanf_args` (pragma directive) . . . . . 279
- `__segment_begin` (extended operator) . . . . . 139
- `__segment_end` (extended operators) . . . . . 139
- `__segment_size` (extended operators) . . . . . 139
- `__set_interrupt_state` (intrinsic function) . . . . . 290
- `__software_interrupt` (intrinsic function) . . . . . 291
- `__STDC_VERSION__` (predefined symbol) . . . . . 297
- `__STDC__` (predefined symbol) . . . . . 297
- `__SUBVERSION__` (predefined symbol) . . . . . 297
- `__swap_nibbles` (intrinsic function) . . . . . 291
- `__task` (extended keyword) . . . . . 261
- `__TID__` (predefined symbol) . . . . . 298
- `__TIME__` (predefined symbol) . . . . . 297
- `__tiny` (extended keyword) . . . . . 237, 262
- `__tinyflash` (extended keyword) . . . . . 238, 262
- `__TINY_AVR__` (predefined symbol) . . . . . 298
- `__ungetchar`, in `stdio.h` . . . . . 308
- `__VA_ARGS__` (preprocessor extension) . . . . . 135
- `__version_1` (extended keyword) . . . . . 263
- `__VERSION_1_CALLS__` (predefined symbol) . . . . . 298
- `__VER__` (predefined symbol) . . . . . 298
- `__watchdog_reset` (intrinsic function) . . . . . 291
- `__write` (library function) . . . . . 93
  - customizing . . . . . 89
- `__write_array`, in `stdio.h` . . . . . 308
- `__write_buffered` (DLIB library function) . . . . . 79, 81
- `__xch` (intrinsic function) . . . . . 291
- `__XMEGA_CORE__` (predefined symbol) . . . . . 298
- `__64bit_doubles` (runtime model attribute) . . . . . 105
- `-D` (compiler option) . . . . . 197
- `-e` (compiler option) . . . . . 203
- `-f` (compiler option) . . . . . 206–207
- `-I` (compiler option) . . . . . 208
- `-l` (compiler option) . . . . . 209
  - for creating skeleton code . . . . . 121
- `-m` (compiler option) . . . . . 211
- `-O` (compiler option) . . . . . 217
- `-o` (compiler option) . . . . . 218
- `-O` (XLINK option) . . . . . 68
- `-r` (compiler option) . . . . . 197
- `-s` (compiler option) . . . . . 222
- `-v` (compiler option) . . . . . 226
- `-y` (compiler option) . . . . . 229
- `-y` (XLINK option) . . . . . 68
- `-z` (compiler option) . . . . . 229
- `--char_is_signed` (compiler option) . . . . . 195
- `--char_is_unsigned` (compiler option) . . . . . 195
- `--clib` (compiler option) . . . . . 195
- `--cpu` (compiler option) . . . . . 196
- `--cross_call_passes` (compiler option) . . . . . 196
- `--c89` (compiler option) . . . . . 194
- `--debug` (compiler option) . . . . . 197

|                                                              |          |                                                                             |          |
|--------------------------------------------------------------|----------|-----------------------------------------------------------------------------|----------|
| --dependencies (compiler option) . . . . .                   | 198      | --no_rampd (compiler option). . . . .                                       | 214      |
| --diagnostics_tables (compiler option) . . . . .             | 200      | --no_static_destruction (compiler option) . . . . .                         | 215      |
| --diag_error (compiler option) . . . . .                     | 199      | --no_system_include (compiler option) . . . . .                             | 215      |
| --diag_remark (compiler option). . . . .                     | 199      | --no_typedefs_in_diagnostics (compiler option) . . . . .                    | 216      |
| --diag_suppress (compiler option) . . . . .                  | 199      | --no_ubrof_messages (compiler option) . . . . .                             | 216      |
| --diag_warning (compiler option). . . . .                    | 200      | --no_warnings (compiler option) . . . . .                                   | 217      |
| --disable_direct_mode (compiler option) . . . . .            | 201      | --no_wrap_diagnostics (compiler option) . . . . .                           | 217      |
| --discard_unused_publics (compiler option) . . . . .         | 201      | --omit_types (compiler option) . . . . .                                    | 218      |
| --dlib (compiler option). . . . .                            | 201      | --only_stdout (compiler option) . . . . .                                   | 218      |
| --dlib_config (compiler option). . . . .                     | 202      | --output (compiler option). . . . .                                         | 218      |
| --do_cross_call (compiler option). . . . .                   | 202      | --predef_macro (compiler option). . . . .                                   | 219      |
| --ec++ (compiler option). . . . .                            | 203      | --preinclude (compiler option) . . . . .                                    | 219      |
| --eecr_address (compiler option) . . . . .                   | 204      | --preprocess (compiler option) . . . . .                                    | 219      |
| --eec++ (compiler option). . . . .                           | 203      | --relaxed_fp (compiler option) . . . . .                                    | 220      |
| --eeprom_size (compiler option). . . . .                     | 204      | --remarks (compiler option) . . . . .                                       | 221      |
| --enable_external_bus (compiler option). . . . .             | 204      | --require_prototypes (compiler option). . . . .                             | 221      |
| --enable_multibytes (compiler option) . . . . .              | 205      | --root_variables (compiler option) . . . . .                                | 221      |
| --enhanced_core (compiler option). . . . .                   | 205      | --segment (compiler option) . . . . .                                       | 222      |
| --error_limit (compiler option) . . . . .                    | 206      | --separate_cluster_for_initialized_variables (compiler<br>option) . . . . . | 223      |
| --force_switch_type (compiler option) . . . . .              | 206      | --silent (compiler option) . . . . .                                        | 223      |
| --guard_calls (compiler option). . . . .                     | 207      | --spmc_r_address (compiler option) . . . . .                                | 224      |
| --header_context (compiler option). . . . .                  | 207      | --strict (compiler option). . . . .                                         | 224      |
| --initializers_in_flash (compiler option) . . . . .          | 208, 226 | --string_literals_in_flash (compiler option). . . . .                       | 224      |
| --library_module (compiler option) . . . . .                 | 210      | --system_include_dir (compiler option) . . . . .                            | 225      |
| --lock_regs (compiler option) . . . . .                      | 210      | --use_c++_inline (compiler option) . . . . .                                | 225      |
| --macro_positions_in_diagnostics (compiler option) . . . . . | 211      | --version1_calls (compiler option) . . . . .                                | 227      |
| --memory_model (compiler option) . . . . .                   | 211      | --vla (compiler option) . . . . .                                           | 228      |
| --mfc (compiler option). . . . .                             | 212      | --warnings_affect_exit_code (compiler option) . . . . .                     | 184, 228 |
| --misrac (compiler option) . . . . .                         | 191      | --warnings_are_errors (compiler option) . . . . .                           | 228      |
| --misrac_verbose (compiler option) . . . . .                 | 191      | --zero_register (compiler option) . . . . .                                 | 230      |
| --misrac1998 (compiler option) . . . . .                     | 191      | --64bit_doubles (compiler option) . . . . .                                 | 194      |
| --misrac2004 (compiler option) . . . . .                     | 191      | --64k_flash (compiler option). . . . .                                      | 194      |
| --module_name (compiler option) . . . . .                    | 212      | ?C_EXIT (assembler label). . . . .                                          | 115      |
| --no_clustering (compiler option) . . . . .                  | 212      | ?C_GETCHAR (assembler label). . . . .                                       | 115      |
| --no_code_motion (compiler option) . . . . .                 | 213      | ?C_PUTCHAR (assembler label). . . . .                                       | 115      |
| --no_cross_call (compiler option). . . . .                   | 213      | @ (operator) . . . . .                                                      | 48       |
| --no_cse (compiler option) . . . . .                         | 213      | placing at absolute address. . . . .                                        | 165      |
| --no_inline (compiler option) . . . . .                      | 214      | placing in segments . . . . .                                               | 166      |
| --no_path_in_file_macros (compiler option). . . . .          | 214      | #include files, specifying . . . . .                                        | 182, 208 |

#warning message (preprocessor extension) . . . . . 299  
 %Z replacement string,  
 implementation-defined behavior . . . . . 346

## Numerics

32-bits (floating-point format) . . . . . 236  
 \_\_64bit\_doubles (runtime model attribute) . . . . . 105  
 --64bit\_doubles (compiler option) . . . . . 194  
 --64k\_flash (compiler option) . . . . . 194  
 64-bit data types, avoiding . . . . . 161  
 64-bits (floating-point format) . . . . . 236