

AVR® IAR C/C++ Compiler

Reference Guide

for Atmel® Corporation's

AVR® Microcontroller

COPYRIGHT NOTICE

© Copyright 1996–2005 IAR Systems. All rights reserved.

No part of this document may be reproduced without the prior written consent of IAR Systems. 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, From Idea to Target, IAR Embedded Workbench, visualSTATE, IAR MakeApp and C-SPY are trademarks owned by IAR Systems AB.

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

Microsoft and Windows are registered trademarks of Microsoft Corporation.

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

EDITION NOTICE

Fourth edition: February 2005

Part number: CAVR-4

This guide applies to version 4.x of AVR IAR Embedded Workbench®.

Brief contents

Tables	xv
Preface	xix
Part 1. Using the compiler	1
Getting started	3
Data storage	15
Functions	27
Placing code and data	33
The DLIB runtime environment	53
The CLIB runtime environment	85
Assembler language interface	93
Using C++	109
Efficient coding for embedded applications	121
Part 2. Compiler reference	135
Data representation	137
Segment reference	149
Compiler options	167
Extended keywords	203
Pragma directives	215
The preprocessor	227
Intrinsic functions	237
Library functions	243

Implementation-defined behavior	255
IAR language extensions	269
Diagnostics	279
Index	281

Contents

Tables	xv
Preface	xix
Who should read this guide	xix
How to use this guide	xix
What this guide contains	xx
Other documentation	xxi
Further reading	xxi
Document conventions	xxii
Typographic conventions	xxii
Part I. Using the compiler	1
Getting started	3
IAR language overview	3
Building applications—an overview	4
Compiling	4
Linking	4
Basic settings for project configuration	5
Processor configuration	5
Memory model	9
Size of double floating-point type	10
Optimization for speed and size	10
Runtime environment	11
Special support for embedded systems	12
Extended keywords	12
Pragma directives	13
Predefined symbols	13
Header files for I/O	13
Accessing low-level features	13

Data storage	15
Introduction	15
Storing data	15
Extended keywords for data	16
Memory models	17
Specifying a memory model	17
Memory types and memory attributes	18
Using data memory attributes	19
Pointers and memory types	20
Structures and memory types	21
More examples	21
C++ and memory types	22
The stack and auto variables	23
Dynamic memory on the heap	25
Functions	27
Controlling functions	27
Extended keywords for functions	27
Function storage	28
Special function types	29
Interrupt functions	29
Monitor functions	30
C++ and special function types	32
Function directives	32
Placing code and data	33
Segments and memory	33
What is a segment?	33
Placing segments in memory	34
Customizing the linker command file	35
Data segments	37
Static memory segments	37
Segments for static data in the linker command file	41
The data stack	41

The return address stack	43
The heap	44
Located data	46
User-defined data segments	46
Code segments	46
Interrupt and reset vectors	46
Functions	46
User-defined segments	47
Compiler-generated segments	47
Efficient usage of segments and memory	47
Controlling data and function placement	47
Using user-defined segments	49
Verifying the linked result of code and data placement	50
Segment too long errors and range errors	50
Linker map file	50
Managing multiple memory spaces	51
The DLIB runtime environment	53
Introduction to the runtime environment	53
Runtime environment functionality	53
Library selection	54
Situations that require library building	55
Library configurations	55
Debug support in the runtime library	56
Using a prebuilt library	56
Customizing a prebuilt library without rebuilding	58
Choosing formatters for printf and scanf	59
Choosing printf formatter	59
Choosing scanf formatter	60
Overriding library modules	61
Building and using a customized library	62
Setting up a library project	63
Modifying the library functionality	63
Using a customized library	63

System startup and termination	64
System startup	65
System termination	65
Customizing system initialization	66
__low_level_init	66
Modifying the file cstartup.s90	67
Standard streams for input and output	67
Implementing low-level character input and output	67
Configuration symbols for printf and scanf	69
Customizing formatting capabilities	70
File input and output	70
Locale	71
Locale support in prebuilt libraries	71
Customizing the locale support	71
Changing locales at runtime	72
Environment interaction	73
Signal and raise	73
Time	74
Strtod	74
Assert	75
Heaps	75
C-SPY Debugger runtime interface	76
Low-level debugger runtime interface	76
The debugger terminal I/O window	77
Checking module consistency	77
Runtime model attributes	77
Using runtime model attributes	78
Predefined runtime attributes	79
User-defined runtime model attributes	80
Implementation of system startup code	81
Modules and segment parts	81
Added C functionality	82
stdint.h	83
stdbool.h	83

math.h	83
stdio.h	83
stdlib.h	83
printf, scanf and strtod	84
The CLIB runtime environment	85
Runtime environment	85
Input and output	87
Character-based I/O	87
Formatters used by printf and sprintf	87
Formatters used by scanf and sscanf	89
System startup and termination	89
System startup	89
System termination	90
Overriding default library modules	90
Customizing system initialization	90
Implementation of cstartup	90
C-SPY runtime interface	91
The debugger terminal I/O window	91
Termination	91
Checking module consistency	91
Assembler language interface	93
Mixing C and assembler	93
Intrinsic functions	93
Mixing C and assembler modules	94
Inline assembler	95
Calling assembler routines from C	96
Creating skeleton code	96
Compiling the code	97
Calling assembler routines from C++	98
Calling convention	99
Choosing a calling convention	99
Function declarations	100
C and C++ linkage	100

Preserved versus scratch registers	101
Function call	102
Function exit	105
Return address handling	106
Restrictions for special function types	106
Examples	107
Call frame information	108
Using C++	109
Overview	109
Standard Embedded C++	109
IAR Extended Embedded C++	110
Enabling C++ support	110
Feature descriptions	111
Classes	111
Functions	114
New and Delete operators	114
Templates	115
Variants of casts	118
Mutable	118
Namespace	119
The STD namespace	119
Pointer to member functions	119
Using interrupts and EC++ destructors	119
Efficient coding for embedded applications	121
Taking advantage of the compilation system	121
Controlling compiler optimizations	122
Fine-tuning enabled transformations	123
Selecting data types and placing data in memory	125
Locating strings in ROM, RAM and flash	125
Using efficient data types	126
Memory model and memory attributes for data	128
Using the best pointer type	128
Anonymous structs and unions	128

Writing efficient code	130
Saving stack space and RAM memory	131
Function prototypes	131
Integer types and bit negation	132
Protecting simultaneously accessed variables	132
Accessing special function registers	133
Non-initialized variables	134
Part 2. Compiler reference	135
Data representation	137
Alignment	137
Alignment in the AVR IAR C/C++ Compiler	137
Basic data types	138
Integer types	138
Floating-point types	139
Pointer types	141
Function pointers	141
Data pointers	141
Casting	142
Structure types	143
Alignment	143
General layout	143
Type and object attributes	144
Type attributes	144
Object attributes	145
Declaring objects in source files	146
Declaring objects volatile	146
Data types in C++	147
Segment reference	149
Summary of segments	149
Descriptions of segments	151

Compiler options	167
Setting command line options	167
Specifying parameters	168
Specifying environment variables	169
Error return codes	169
Options summary	169
Descriptions of options	173
Extended keywords	203
Summary of extended keywords	203
Descriptions of extended keywords	204
Pragma directives	215
Summary of pragma directives	215
Descriptions of pragma directives	216
The preprocessor	227
Overview of the preprocessor	227
Predefined symbols	227
Summary of predefined symbols	228
Descriptions of predefined symbols	229
Preprocessor extensions	235
Intrinsic functions	237
Intrinsic functions summary	237
Descriptions of intrinsic functions	238
Library functions	243
Introduction	243
Header files	243
Library object files	244
Reentrancy	244
IAR DLIB Library	244
C header files	245
C++ header files	246

Library functions as intrinsic functions	248
IAR CLIB Library	248
Library definitions summary	249
AVR-specific library functions	249
Specifying read and write formatters	250
Implementation-defined behavior	255
Descriptions of implementation-defined behavior	255
Translation	255
Environment	256
Identifiers	256
Characters	256
Integers	258
Floating point	258
Arrays and pointers	259
Registers	259
Structures, unions, enumerations, and bitfields	259
Qualifiers	260
Declarators	260
Statements	260
Preprocessing directives	260
IAR CLIB Library functions	262
IAR DLIB Library functions	265
IAR language extensions	269
Why should language extensions be used?	269
Descriptions of language extensions	269
Diagnostics	279
Message format	279
Severity levels	279
Setting the severity level	280
Internal error	280
Index	281

Tables

1: Typographic conventions used in this guide	xxii
2: Mapping of processor options	6
3: Summary of processor configuration	8
4: Summary of memory models	10
5: Command line options for specifying library and dependency files	12
6: Memory model characteristics	18
7: Memory attributes for data	19
8: Memory attributes for functions	28
9: XLINK segment memory types	34
10: Memory layout of a target system (example)	35
11: Segment name suffixes	38
12: Heaps, memory types, and segments	44
13: Library configurations	55
14: Levels of debugging support in runtime libraries	56
15: Prebuilt libraries	57
16: Customizable items	58
17: Formatters for printf	59
18: Formatters for scanf	60
19: Descriptions of printf configuration symbols	69
20: Descriptions of scanf configuration symbols	69
21: Low-level I/O files	70
22: Heaps and memory types	75
23: Functions with special meanings when linked with debug info	76
24: Example of runtime model attributes	78
25: Predefined runtime model attributes	79
26: Prebuilt libraries	86
27: Registers used for passing parameters	102
28: Passing parameters in registers	103
29: Registers used for returning values	106
30: Compiler optimization levels	122
31: Integer types	138

32: Floating-point types	139
33: Function pointers	141
34: Data pointers	141
35: size_t typedef	142
36: ptrdiff_t typedef	143
37: Volatile objects with special handling	147
38: Segment summary	149
39: Memory models	151
40: Heap memory address range	157
41: Environment variables	169
42: Error return codes	169
43: Compiler options summary	169
44: Generating a list of dependencies (--dependencies)	175
45: Specifying switch type	182
46: Accessing variables with aggregate initializers	184
47: Generating a compiler list file (-l)	185
48: Enabling MISRA C rules (--misrac)	187
49: Directing preprocessor output to file (--preprocess)	193
50: Specifying speed optimization (-s)	195
51: Processor variant command line options	197
52: Specifying size optimization (-z)	200
53: Summary of extended keywords for functions	203
54: Summary of extended keywords for data	204
55: EEPROM address ranges	205
56: Near address ranges	206
57: Far address ranges	206
58: Farflash address ranges	206
59: Farfunc pointer size	207
60: Flash address ranges	208
61: Generic pointer size	208
62: Huge address ranges	208
63: Hugeflash address ranges	209
64: I/O address ranges	210
65: Near address ranges	210

66: Nearfunc pointer size	211
67: Tiny address ranges	213
68: Tinyflash address ranges	213
69: Pragma directives summary	215
70: Predefined symbols summary	228
71: Predefined memory model symbol values	232
72: Intrinsic functions summary	237
73: Traditional standard C header files—DLIB	245
74: Embedded C++ header files	246
75: Additional Embedded C++ header files—DLIB	246
76: Standard template library header files	247
77: New standard C header files—DLIB	247
78: IAR CLIB Library header files	249
79: Miscellaneous IAR CLIB Library header files	249
80: Message returned by <code>strerror()</code> —IAR CLIB library	265
81: Message returned by <code>strerror()</code> —IAR DLIB library	268

Preface

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

Who should read this guide

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

- 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 machine.

How to use this guide

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

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

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

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 AVR IAR C/C++ Compiler for efficiently developing your application.
- *Data storage* describes how data can be stored in memory, with emphasis on the different memory models and memory type attributes.
- *Functions* describes the special function types, such as interrupt functions and monitor functions.
- *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 runtime environment in which an application executes. It covers how you can modify it by setting options, overriding default library modules, or building your own library. The chapter also describes system initialization and introduces the file `cstartup`, as well as how to use modules for locale, and file I/O.
- *The CLIB runtime environment* gives an overview of the runtime libraries and how they can be customized. 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 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. Compiler reference

- *Data representation* describes the available data types, pointers, and structure types. This chapter also gives information about type and object attributes.
- *Segment reference* gives reference information about the compiler's use of segments.
- *Compiler options* explains how to set the compiler options, gives a summary of the options, and contains detailed reference information for each compiler option.
- *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.
- *The preprocessor* gives a brief overview of the preprocessor, including reference information about the different preprocessor directives, symbols, and other related information.
- *Intrinsic functions* gives reference information about the functions that can be used for accessing AVR-specific low-level features.

- *Library functions* gives an introduction to the C or C++ library functions, and summarizes the header files.
- *Implementation-defined behavior* describes how the AVR IAR C/C++ Compiler handles the implementation-defined areas of the C language standard.
- *IAR language extensions* describes the IAR extensions to the ISO/ANSI standard for the C programming language.
- *Diagnostics* describes how the compiler's diagnostic system works.

Other documentation

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

- Using the IAR Embedded Workbench™ IDE with the IAR C-SPY™ Debugger, refer to the *AVR® IAR Embedded Workbench™ IDE User Guide*
- Programming for the AVR IAR Assembler, refer to the *AVR® IAR Assembler Reference Guide*
- Using the IAR XLINK Linker™, the IAR XAR Library Builder™, and the IAR XLIB Librarian™, refer to the *IAR Linker and Library Tools Reference Guide*
- Using the IAR DLIB Library functions, refer to the online help system
- Using the IAR CLIB Library functions, refer to the *IAR C Library Functions Reference Guide*, available from the online help system.
- Porting application code and projects created with a previous AVR IAR Embedded Workbench IDE, refer to *AVR® IAR Embedded Workbench Migration Guide*.

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

FURTHER READING

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

- Barr, Michael, and Andy Oram, ed. *Programming Embedded Systems in C and C++*. O'Reilly & Associates.
- Harbison, Samuel P. and Guy L. Steele (contributor). *C: A Reference Manual*. Prentice Hall.
- Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall. [The later editions describe the ANSI C standard.]
- Labrosse, Jean J. *Embedded Systems Building Blocks: Complete and Ready-To-Use Modules in C*. R&D Books.
- Lippman, Stanley B. and Josée Lajoie. *C++ Primer*. Addison-Wesley.
- Mann, Bernhard. *C für Mikrocontroller*. Franzis-Verlag. [Written in German.]
- Stroustrup, Bjarne. *The C++ Programming Language*. Addison-Wesley.

We recommend that you visit the following websites:

- The Atmel® Corporation website, **www.atmel.com**, contains information and news about the AVR microcontrollers.
- The IAR website, **www.iar.com**, holds application notes and other product information.
- Finally, the Embedded C++ Technical Committee website, **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.

TYPOGRAPHIC CONVENTIONS

This guide uses the following typographic conventions:




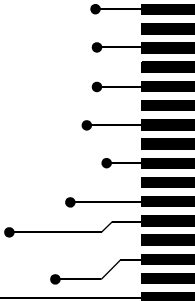
Style	Used for
<code>computer</code>	Text that you enter or that appears on the screen.
<i>parameter</i>	A label representing the actual value you should enter as part of a command.
[option]	An optional part of a command.
{a b c}	Alternatives in a command.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
<i>reference</i>	A cross-reference within this guide or to another guide.
...	An ellipsis indicates that the previous item can be repeated an arbitrary number of times.
	Identifies instructions specific to the IAR Embedded Workbench interface.
	Identifies instructions specific to the command line interface.
	Identifies helpful tips and programming hints.

Table 1: Typographic conventions used in this guide

Part I. Using the compiler

This part of the AVR® IAR C/C++ Compiler Reference Guide includes the following chapters:

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





Getting started

This chapter gives the information you need to get started using the AVR IAR C/C++ 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 will be studied in more detail.

IAR language overview

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

- C, the most widely used high-level programming language used in the embedded systems industry. Using the AVR IAR C/C++ Compiler, you can build freestanding applications that follow the standard ISO 9899:1990. This standard is commonly known as ANSI C.
- C++, a modern object-oriented programming language with a full-featured library well suited for modular programming. IAR Systems supports two levels of the C++ language:
 - Embedded C++ (EC++), a subset of the C++ programming standard, which is intended for embedded systems programming. It is defined by an industry consortium, the Embedded C++ Technical committee. See the chapter *Using C++*.
 - IAR Extended EC++, with additional features such as full template support, namespace support, the new cast operators, as well as the Standard Template Library (STL).

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

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

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

Building applications—an overview

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

A library is a collection of object files. 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 command file, which describes the available resources of the target system.



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

COMPILING

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

```
iccavr myfile.c
```

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

LINKING

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

- A number of object files and possibly certain libraries
- The standard library containing the runtime environment and the standard language functions
- A program start label
- A linker command file that describes the memory layout 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  
c13s-ec.r90 -o aout.a90 -FIntel-extended
```

In this example, `myfile.r90` and `myfile2.r90` are object files, `lnkm128s.xcl` is the linker command file, and `c13s-ec.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 `-F` can be used for specifying the output format. (The default output format is `Motorola`.)

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—in which case you need output without debug information, such as Intel-hex or Motorola S-records.

Basic settings for project configuration

This section gives an overview of the basic settings for the project setup that are needed to make the compiler generate the best code for the AVR device you are using. You can specify the options either from the command line interface or in the IAR Embedded Workbench IDE. For details about how to set options, see *Setting command line options*, page 167, and the *AVR® IAR Embedded Workbench™ IDE User Guide*, respectively.

The basic settings available for the AVR microcontroller are:

- Processor configuration
- Memory model
- Size of double floating-point type
- Optimization for speed and size
- Runtime environment.

In addition to these settings, there are many other options and settings available for fine-tuning the result even further. See the chapter *Compiler options* for a list of all available options.

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=derivative and -vn
```

Your program may use only one processor option at a time, and the same processor option must be used by all user and library modules in order 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 and allows the compiler to optimize accesses to code memory in a way that the `-v` option does not. See *Mapping of processor options --cpu and -v*, page 6.

The `--cpu=derivative` option implicitly sets up all internal compiler settings needed to generate code for the processor variant you are using. The following 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*, page 8. To read more about the generated code, see `-v`, page 197.



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



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

Mapping of processor options `--cpu` and `-v`

The following table shows the mapping of processor options and which AVR microcontrollers they support:

Processor variant	Generic processor option	Supported AVR derivative
<code>--cpu=1200</code>	<code>-v0</code>	AT90S1200
<code>--cpu=2313</code>	<code>-v0</code>	AT90S2313
<code>--cpu=2323</code>	<code>-v0</code>	AT90S2323
<code>--cpu=2333</code>	<code>-v0</code>	AT90S2333
<code>--cpu=2343</code>	<code>-v0</code>	AT90S2343
<code>--cpu=4414</code>	<code>-v1</code>	AT90S4414
<code>--cpu=4433</code>	<code>-v0</code>	AT90S4433
<code>--cpu=4434</code>	<code>-v1</code>	AT90S4434
<code>--cpu=8515</code>	<code>-v1</code>	AT90S8515
<code>--cpu=8534</code>	<code>-v1</code>	AT90S8534
<code>--cpu=8535</code>	<code>-v1</code>	AT90S8535

Table 2: Mapping of processor options

Processor variant	Generic processor option	Supported AVR derivative
--cpu=at43usb320a	-v3	AT43USB320A
--cpu=at43usb325	-v3	AT43USB325
--cpu=at43usb326	-v3	AT43USB326
--cpu=at43usb351m	-v3	AT43USB351m
--cpu=at43usb353m	-v3	AT43USB353m
--cpu=at43usb355	-v3	AT43USB355
--cpu=at94k	-v3	FpSLic
--cpu=at86rf401	-v0	AT86RF401
--cpu=can128	-v3	AT90CAN128
--cpu=m8	-v1	ATmega8
--cpu=m16	-v3	ATmega16
--cpu=m32	-v3	ATmega32
--cpu=m48	-v1	ATmega48
--cpu=m64	-v3	ATmega64
--cpu=m88	-v1	ATmega88
--cpu=m103	-v3	ATmega103
--cpu=m128	-v3	ATmega128
--cpu=m161	-v3	ATmega161
--cpu=m162	-v3	ATmega162
--cpu=m163	-v3	ATmega163
--cpu=m165	-v3	ATmega165
--cpu=m168	-v3	ATmega168
--cpu=m169	-v3	ATmega169
--cpu=m2560	-v5	ATmega2560
--cpu=m2561	-v5	ATmega2561
--cpu=m323	-v3	ATmega323
--cpu=m325	-v3	ATmega325
--cpu=m3250	-v3	ATmega3250

Table 2: Mapping of processor options (Continued)

Processor variant	Generic processor option	Supported AVR derivative
--cpu=m329	-v3	ATmega329
--cpu=m3290	-v3	ATmega3290
--cpu=m406	-v3	ATmega406
--cpu=m645	-v3	ATmega645
--cpu=m6450	-v3	ATmega6450
--cpu=m649	-v3	ATmega649
--cpu=m6490	-v3	ATmega6490
--cpu=m8515	-v1	ATmega8515
--cpu=m8535	-v1	ATmega8535
--cpu=tiny10	-v0	ATtiny10
--cpu=tiny11	-v0	ATtiny11
--cpu=tiny12	-v0	ATtiny12
--cpu=tiny13	-v0	ATtiny13
--cpu=tiny15	-v0	ATtiny15
--cpu=tiny25	-v0	ATtiny25
--cpu=tiny26	-v0	ATtiny26
--cpu=tiny28	-v0	ATtiny28
--cpu=tiny45	-v1	ATtiny45
--cpu=tiny85	-v1	ATtiny85
--cpu=tiny2313	-v0	ATtiny2313

Table 2: Mapping of processor options (Continued)

Summary of processor configuration

The following 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	Tiny	<code>__nearfunc</code>	≤ 256 bytes	≤ 8 Kbytes
-v1	Tiny, Small	<code>__nearfunc</code>	≤ 64 Kbytes	≤ 8 Kbytes

Table 3: Summary of processor configuration

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

Table 3: Summary of processor configuration (Continued)

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 142
- `-v2` and `-v4`: There are currently no derivatives that match these processor options, which have been added to support future derivatives
- 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 Mbyte 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 Mbyte data.

MEMORY MODEL

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

In the AVR IAR C/C++ 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. 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.

Your program may use only one memory model at a time, and the same model must be used by all user modules and all library modules.

Summary of memory models

The following table summarizes the characteristics for each memory model:

Memory model	Generic processor option	Default memory attribute	Default data pointer	Max. stack size
Tiny	-v0, -v1, -v2, -v3, -v5	__tiny	__tiny	≤ 256 bytes
Small	-v1, -v3, -v4, -v5, -v6	__near	__near	≤ 64 Kbytes
Large	-v4, -v6	__far	__far	≤ 16 Mbytes

Table 4: Summary of memory models

For more details about memory models, see *Memory models*, page 17.

SIZE OF DOUBLE FLOATING-POINT TYPE

Floating-point values are represented by 32- and 64-bit numbers in standard IEEE754 format. By enabling the compiler option `--64bit_doubles`, you can choose whether data declared as `double` should be represented with 32 bits or 64 bits. The data type `float` is always represented using 32 bits.

OPTIMIZATION FOR SPEED AND SIZE

The AVR IAR C/C++ 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 two optimization goals—*size* and *speed*. Most optimizations will make the application both smaller and faster. However, when this is not the case, the compiler uses the selected optimization goal to decide how to perform the optimization.

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

For details about compiler optimizations, see *Controlling compiler optimizations*, page 122. For more information about efficient coding techniques, see the chapter *Efficient coding for embedded applications*.

RUNTIME ENVIRONMENT

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

There are two different sets of runtime libraries provided:

- The IAR DLIB Library, which supports ISO/ANSI C and C++. This library also supports floating-point numbers in IEEE 754 format and it can be configured to include different levels of support for locale, file descriptors, multibyte characters, et cetera.
- The IAR CLIB Library is a light-weight library, which is not fully compliant with ISO/ANSI 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 you choose can be one of the prebuilt libraries, or a library that you have customized and built yourself. The IAR Embedded Workbench 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, there is no need to choose a runtime library.

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

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



Choosing a runtime library in IAR Embedded Workbench

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 two different configurations—Normal and Full—which include different levels of support for locale, file descriptors, multibyte characters, et cetera. See *Library configurations*, page 55, for more information.

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



Choosing a runtime library from the command line

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

Command line	Description
<code>-I\avr\inc</code>	Specifies the include paths
<code>-I\avr\inc\{clib dlib}</code>	Specifies the library-specific include path. Use <code>clib</code> or <code>dlib</code> depending on which library you are using.
<code>libraryfile.r90</code>	Specifies the library object file
<code>--dlib_config</code> <code>C:\...\configfile.h</code>	Specifies the library configuration file (for the IAR DLIB Library only)

Table 5: Command line options for specifying library and dependency files

For a list of all prebuilt library object files for the IAR DLIB Library, see Table 15, *Prebuilt libraries*, page 57. The table also shows how the object files correspond to the dependent project options, and the corresponding configuration files. Make sure to use the object file that matches your other project options.

For a list of all prebuilt object files for the IAR CLIB Library, see Table 26, *Prebuilt libraries*, page 86. The table also shows how the object files correspond to the dependent project options. 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 59 (DLIB) and *Input and output*, page 87 (CLIB).
- The size of the stack and the heap, see *The data stack*, page 41, and *The return address stack*, page 43, respectively.

Special support for embedded systems

This section briefly describes the extensions provided by the AVR IAR C/C++ Compiler to support specific features of the AVR microcontroller.

EXTENDED KEYWORDS

The AVR IAR C/C++ 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 IAR Embedded Workbench.



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 179 for additional information.

For detailed descriptions of the extended keywords, see the chapter *Extended keywords*. To read about special function types, see *Special function types*, page 29.

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 AVR IAR C/C++ Compiler. They are consistent with ISO/ANSI C, and are very useful when you want to make sure that the source code is portable.

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

PREDEFINED SYMBOLS

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

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

HEADER FILES FOR I/O

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

For an example, see *Accessing special function registers*, page 133.

ACCESSING LOW-LEVEL FEATURES

For hardware-related parts of your application, accessing low-level features is essential. The AVR IAR C/C++ 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 93.

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, AVR IAR C/C++ 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, 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 type 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.

STORING DATA

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

- On the stack. This is memory space that can be used by a function as long as it is executing. When the function returns to its caller, the memory space is no longer valid.
- Static memory. This kind of memory is allocated once and for all; it remains valid through the entire execution of the application. Variables that are either global or declared static are placed in this type of memory. The word *static* in this context means that the amount of memory allocated for this type of variable does not change while the application is running.

- On the heap. Once memory has been allocated on the heap, it remains valid until it is explicitly released back to the system by the application. This type of memory is useful when the number of objects is not known until the application executes. Note that there are potential risks connected with using the heap in systems with a limited amount of memory, or systems that are expected to run for a long time.

EXTENDED KEYWORDS FOR DATA

The extended keywords that can be used for data control the following:

- For data memory space, keywords that control the placement and type of objects and pointers: `__tiny`, `__near`, `__far`, `__huge`, and `__regvar`
- For the EEPROM memory space, keyword that controls the placement and type of objects and pointers: `__eeprom`
- For the code (flash) memory space, keywords that control the placement and type of objects and pointers: `__tinyflash`, `__flash`, `__farflash`, and `__hugeflash`
- For the I/O memory space, keyword that controls the placement and type of objects and pointers: `__ext_io`, `__io`
- Special pointer that can access data objects in both data and code memory space: `__generic`
- Other characteristics of objects: `__root` and `__no_init`.

See the chapter *Data storage* in *Part 1. Using the compiler* for more information about how to use data memory types.

Syntax

The keywords follow the same syntax as the type qualifiers `const` and `volatile`. The following declarations place the variable `i` and `j` in EEPROM memory. The variables `k` and `l` behave in the same way:

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

Note that the keyword affects both identifiers.

In addition to the rules presented here—to place the keyword directly in the code—the directives `#pragma type_attribute` and `#pragma object_attribute` can be used for specifying the keywords. Refer to the chapter *Pragma directives* for details about how to use the extended keywords together with pragma directives.

Pointers

A keyword that is followed by an asterisk (*), affects the type of the pointer being declared. A pointer to EEPROM memory is thus declared by:

```
char __eeprom * p;
```

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

```
char __eeprom * __far p2;
```

Type definitions

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

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

and

```
__far char b;
char __far *bp;
```

Memory models

The AVR IAR C/C++ Compiler supports a number of memory models that can be used for applications with different data requirements.

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

- The placement of static and global variables, as well as 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.

The memory model only specifies the default memory type. It is possible to override this for individual variables and pointers. For information about how to specify a memory type for individual objects, see *Using data memory attributes*, page 19.

SPECIFYING A MEMORY MODEL

Three memory models are implemented: Tiny, Small, and Large. These models are controlled by the `--memory_model` option. Each memory 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.

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. 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.

The following table summarizes the different memory models:

Memory model	Default memory attribute	Default data pointer	Max stack size	Supported by processor option
Tiny	<code>__tiny</code>	<code>__tiny</code>	≤ 256 bytes	<code>-v0</code> , <code>-v1</code> , <code>-v2</code> , <code>-v3</code> , <code>-v5</code>
Small	<code>__near</code>	<code>__near</code>	≤ 64 Kbytes	<code>-v1</code> , <code>-v3</code> , <code>-v4</code> , <code>-v5</code> , <code>-v6</code>
Large	<code>__far</code>	<code>__far</code>	≤ 16 Mbytes	<code>-v4</code> , <code>-v6</code>

Table 6: Memory model characteristics



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



Use the `--memory_model` option to specify the memory model for your project; see `-m`, *--memory_model*, page 186.

Note that the default memory type can be overridden by explicitly specifying a memory attribute, using either keywords or the `#pragma type_attribute` directive. For more information about the different memory types, see *Memory types and memory attributes*, page 18.

Memory types and memory attributes

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

The AVR IAR C/C++ 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 range, to cheap methods that can access limited memory areas. Each memory type corresponds to one memory access method. By mapping different memories—or part of memories—to memory types, the compiler can generate code that can access data efficiently.

For example, the memory 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 AVR IAR C/C++ 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.

Summary of characteristics of memory attributes

The following table summarizes the available memory attributes:

Memory attribute	Pointer size	Memory space	Address range	Max object size
<code>__tiny</code>	1 byte	Data	0-0xFF	128 bytes
<code>__near</code>	2 bytes	Data	0-0xFFFF	32 Kbytes
<code>__far</code>	3 bytes	Data	0-0xFFFFFFFF (16-bit pointer arithmetics)	32 Kbytes
<code>__huge</code>	3 bytes	Data	0-0xFFFFFFFF	8 Mbytes
<code>__tinyflash</code>	1 byte	Code	0-0xFF	128 bytes
<code>__flash</code>	2 bytes	Code	0-0xFFFF	32 Kbytes
<code>__farflash</code>	3 bytes	Code	0-0xFFFFFFFF (16-bit pointer arithmetics)	32 Kbytes
<code>__hugeflash</code>	3 bytes	Data	0-0xFFFFFFFF	8 Mbytes
<code>__eeprom</code>	1 bytes	EEPROM	0-0xFF	128 bytes
<code>__eeprom</code>	2 bytes	EEPROM	0-0xFFFFFFFF	32 Kbytes
<code>__io</code>	N/A	I/O space	0-0x3F	4 bytes
<code>__io</code>	N/A	Data	0x60-0xFF	4 bytes
<code>__ext_io</code>	N/A	Data	0x100-0xFFFF	4 bytes

Table 7: Memory attributes for data

Memory attribute	Pointer size	Memory space	Address range	Max object size
__generic	2 bytes 3 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 <code>-v0</code> and <code>-v1</code> .	32 Kbytes 8 Mbytes
__regvar	N/A	Data	0x4-0x0F	4 bytes

Table 7: Memory attributes for data (Continued)

The keywords are only available if language extensions are enabled in the AVR IAR C/C++ Compiler.



In IAR Embedded Workbench, language extensions are enabled by default.



Use the `-e` compiler option to enable language extensions. See `-e`, page 179 for additional information.

For syntax information, see *Extended keywords for data*, page 16. For reference information about each keyword, see *Descriptions of extended keywords*, page 204.

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 AVR IAR C/C++ 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 farflash memory is declared by:

```
int __farflash * p;
```

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

```
char __farflash * __tiny p2;
```

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. For information about the sizes of the different pointer types, see *Pointer types*, page 141.

In the AVR IAR C/C++ Compiler, it is illegal to convert pointers between different types without using explicit casts. For more details, see *Casting*, page 142.

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 alpha;
    int __flash * beta; /* Pointer to variables in flash memory */
};
__eeprom struct MyStruct gamma;
```

The following declaration is incorrect:

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

MORE EXAMPLES

The following is a series of examples with descriptions. First, some integer variables are defined and then pointer variables are introduced. Finally, a function accepting a pointer to an integer in flash memory is declared. The function returns a pointer to an integer in eeprom memory. It makes no difference whether the memory attribute is placed before or after the data type. In order to read the following examples, start from the left and add one qualifier at each step

<code>int a;</code>	A variable defined in default memory.
<code>int __flash b;</code>	A variable in flash memory.
<code>__eeprom int c;</code>	A variable in eeprom memory.
<code>int * d;</code>	A pointer stored in default memory. The pointer points to an integer in default memory.

<code>int __flash * e;</code>	A pointer stored in default memory. The pointer points to an integer in flash memory.
<code>int __flash * __eeprom f;</code>	A pointer stored in eeprom memory pointing to an integer stored in flash memory.
<code>int __eeprom * myFunction(int __flash *);</code>	A declaration of a function that takes a parameter which is a pointer to an integer stored in flash memory. The function returns a pointer to an integer stored in eeprom memory.

In short:

<code>int</code>	The basic type is an integer.
<code>int __flash</code>	The integer is stored in flash memory.
<code>int __flash *</code>	This is a pointer to the integer.
<code>int __flash * __eeprom</code>	The pointer is stored in eeprom memory.

C++ and memory types

A C++ class object is placed in one memory type, in the same way as for normal C structures. However, the class members that are considered to be part of the object are the non-static member variables. The static member variables can be placed individually in any kind of memory.

Remember, in C++ there is only one instance of each static member variable, regardless of the number of class objects.

Also note that for non-static member functions—unless class memory is used, see *Classes*, page 111—the `this` pointer will be of the default data pointer type. This means that it must be possible to convert a pointer to the object to the default pointer type. The restrictions that apply to the default pointer type also apply to the `this` pointer.

Example

In the example below, an object, named `delta`, of the type `MyClass` is defined in `data16` memory. The class contains a static member variable that is stored in `data20` memory.

```
// The class declaration (placed in a header file):
class MyClass
{
public:
    int alpha;
    int beta;

    __eeprom static int gamma;
};

// Definitions needed (should be placed in a source file):
__eeprom int MyClass::gamma;

// A variable definition:
MyClass delta;
```

The stack and auto variables

Variables that are defined inside a function—not declared static—are named *auto variables* by the C standard. A small number 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 live as long as the function executes; when the function returns, the memory allocated on the stack is released.

The stack can contain:

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

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

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

Advantages

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

It is possible for a function to call itself—a so-called 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 has returned. The following function demonstrates a common programming mistake. It returns a pointer to the variable `x`, a variable that ceases to exist when the function returns.

```
int * MyFunction()
{
    int x;
    ... do something ...
    return &x;
}
```

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

Dynamic memory on the heap

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

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

In C++, there is a special keyword, `new`, designed to allocate memory and run constructors. Memory allocated with `new` must be released using the keyword `delete`.

The AVR IAR C/C++ Compiler supports having heaps in tiny, near, far, and huge memory. For more information about this, see *The return address stack*, page 43.

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 because your application simply uses too much memory. It can also become full if memory that no longer is in use has not been released.

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

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

Unfortunately, fragmentation tends to increase as memory is allocated and released. Hence, 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. First, you get a brief overview of mechanisms for controlling functions, as well as information about memory type attributes for function storage. Then, the special function types interrupt and monitor are described, including how to declare C++ member functions by using special function types.

Controlling functions

Writing a function in C or C++, you may want to control for example the following:

- The function storage, see *Function storage*, page 28
- The function execution, see *Special function types*, page 29
- The used calling convention, see *Calling convention*, page 99.

The AVR IAR C/C++ Compiler provides a wide set of extended keywords which let you control a function.

EXTENDED KEYWORDS FOR FUNCTIONS

The extended keywords that can be used for functions can be divided into three groups:

- Keywords that control the placement and type of the functions. Keywords of this group must be specified both when the function is declared and when it is defined: `__nearfunc` and `__farfunc`.
- Keywords that control the type of the functions. Keywords of this group only have to be specified when the function is defined: `__interrupt`, `__task`, and `__version_1`.
- Keywords that only control the defined function: `__root`, `__monitor`, and `__noreturn`.

Keywords that control the placement and type of the functions are also referred to as *type attributes*. Typically these functions controls aspects of the function visible to the surrounding context. Keywords that only control the behavior of the function and do not affect the function interface are referred to as *object attributes*. To read more about type and object attributes, see *Type and object attributes*, page 144. For reference information about each keyword, see the chapter *Extended keywords*, page 203.

Syntax

The extended keywords are specified before the return type, for example:

```
__interrupt void alpha(void);
```

The keywords that are *type* attributes must be specified both when they are defined and in the declaration. *Object* attributes only have to be specified when they are defined since they do not affect the way an object or function is used.

In addition to the rules presented here—to place the keyword directly in the code—the directives `#pragma type_attribute` and `#pragma object_attribute` can be used for specifying the keywords. Refer to the chapter *Pragma directives* for details about how to use the extended keywords together with pragma directives.

Function storage

There are two memory attributes for controlling function storage: `__nearfunc` and `__farfunc`.

The following table summarizes the characteristics for each memory attribute:

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>	0-0x7FFFE (8 Mbytes)	24 bits	-v5, -v6

Table 8: Memory attributes for functions

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 attribute 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. 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; see *Casting*, page 142. For more information about function pointers, see *Function pointers*, page 141.

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*, page 47.

Special function types

This section describes the special function types interrupt and monitor. The AVR IAR C/C++ Compiler allows an application to take full advantage of these AVR features, without forcing you to implement anything in assembler language.

INTERRUPT FUNCTIONS

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

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

The 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, alternatively multiple vector numbers, which is specified in the AVR microcontroller documentation from the chip manufacturer. The header file `ioderivative.h`, where *derivative* corresponds to the selected derivative, contains predefined names for the existing exception vectors.

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

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

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

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

MONITOR FUNCTIONS

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

To define a monitor function, you can use the `__monitor` keyword. For reference information, see `__monitor`, page 210.

Example of implementing a semaphore in C

In the following example, a semaphore is implemented using one static variable and two monitor functions. A semaphore can be locked by one process, and is used for preventing processes from simultaneously using resources that can only be used by one process at a time, for example a printer.

```

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

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

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

/* release_lock -- Unlock the lock. */

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

```

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

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

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

The drawback using this method is that interrupts are disabled for the entire monitor function.

Example of implementing a semaphore in C++

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

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

```
#include <inavr.h>

volatile long tick_count = 0;

/* Class for controlling critical blocks */
class Mutex
{
public:
    Mutex ()
    {
        _state = __save_interrupt();
        __disable_interrupt();
    }

    ~Mutex ()
    {
        __restore_interrupt(_state);
    }

private:
    unsigned char _state;
};
```

```

void f()
{
    static long next_stop = 100;
    extern void do_stuff();
    long tick;

    /* A critical block */
    {
        Mutex m;
        /* Read volatile variable 'tick_count' in a safe way
           and put the value in a local variable */

        tick = tick_count;
    }

    if (tick >= next_stop)
    {
        next_stop += 100;
        do_stuff();
    }
}

```

C++ AND SPECIAL FUNCTION TYPES

C++ member functions can be declared using special function types. However, the following restriction apply:

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

FUNCTION DIRECTIVES

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

The function directives `FUNCTION`, `ARGFRAME`, `LOCFRAME`, and `FUNCALL` are generated by the AVR IAR C/C++ 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 reference information about the function directives, see the *AVR® IAR Assembler Reference Guide*.

Placing code and data

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

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

Segments and memory

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

WHAT IS A SEGMENT?

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

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

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

For detailed information about individual segments, see the chapter *Segment reference* in *Part 2. Compiler reference*.

Segment memory type

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

By default, the AVR IAR C/C++ Compiler uses only the following XLINK segment memory types:

Segment memory type	Description
CODE	For executable code in flash memory space
DATA	For data placed in data memory space (RAM), and for data placed in flash memory space
XDATA	For data placed in EEPROM memory space

Table 9: XLINK segment memory types

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

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

Placing segments in memory

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

In particular, the linker command file specifies:

- The placement of segments in memory
- The maximum stack size
- The maximum heap size (only for the IAR DLIB runtime environment).

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

CUSTOMIZING THE LINKER COMMAND FILE

The only change you will normally have to make to the supplied linker command file is to customize it so it reflects the target system memory map.

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

Range	Type
0x100-0xFFF	RAM
0x0-0x1FFFF	FLASH
0x0-0xFFF	EEPROM
0x1000-0xFFFFF	External RAM

Table 10: Memory layout of a target system (example)

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

The contents of the linker command file

The `config` directory contains ready-made linker command files. The file contains the information required by the linker, and is ready to be used. If, for example, your application uses additional external RAM, you need to add details about the external RAM memory area. Remember not to change the original file. We recommend that you make a copy in the working directory, and modify the copy instead.

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

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

- The CPU used:
`-ca90`
 This specifies your target microcontroller.
- Definitions of constants used later in the file. These are defined using the `XLINK` option `-D`.
- The placement directives (the largest part of the linker command file). Segments can be placed using the `-Z` and `-P` options. The former will place the segment parts in the order they are found, while the latter will try to rearrange them to make better use of the memory. The `-P` option is useful when the memory where the segment should be placed is not continuous.

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

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

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

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

The following illustrates how to use the **-z** command to place the segment `MYSEGMENTA` followed by the segment `MYSEGMENTB` in `DATA` memory (that is, external RAM) in the memory range `0x1000-0xCFFF`.

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

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

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

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

```
-Z (DATA) MYSMALLSEGMENT=1000-20FF
-Z (DATA) 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 specified ranges.

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. The command will place the data segment `MYDATA` in `DATA` memory (that is, in RAM) in the memory range:

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

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

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

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. In most cases these segments are located in the data memory space. However, some of the segments are located in the code memory space.

To get a clear understanding about how the data segments work, you must be familiar with the different memory types and the different memory models available in the AVR IAR C/C++ Compiler. If you need to refresh these details, see the chapter *Data storage*.

STATIC MEMORY SEGMENTS

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

- Variables that are initialized to a non-zero value
- Variables that are initialized to zero
- Variables that are located by use of the `@` operator or the `#pragma location` directive
- Variables that are declared as `const` and therefore can be stored in external 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

All static data segment names consist of two parts—the *segment base name* and a *suffix*—for instance, `NEAR_Z`. The segment base names are derived from the memory type attributes, for example the attribute:

```
__near
```

would yield the segment base name `NEAR`.

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

Suffix	Categories of declared data
<code>_N</code>	Non-initialized data
<code>_Z</code>	Zero-initialized data
<code>_I</code>	Non-zero initialized data
<code>_ID</code>	Initializers for the above
<code>_C</code>	Constants
<code>_F</code>	Data placed in flash memory
<code>_AN</code>	Non-initialized absolute addressed data
<code>_AC</code>	Constant absolute addressed data

Table 11: Segment name suffixes

For information about the static data segments, their possible memory ranges, and in what type of memory they can be placed, see the chapter *Segment reference* in this guide. For more details about segment memory types, see *Segment memory type*, page 34.

Examples

Assume the following examples:

<code>__near int j;</code>	The near variables that are to be initialized to zero when the system starts will be placed in the segment <code>NEAR_Z</code> .
<code>__near int i = 0;</code>	
<code>__no_init __near int j;</code>	The near non-initialized variables will be placed in the segment <code>NEAR_N</code> .
<code>__near int j = 4;</code>	When using the <code>-y</code> option, the near non-zero initialized variables will be placed in the segment <code>NEAR_I</code> , and initializer data in segment <code>NEAR_ID</code> .

Initialized data

In ISO/ANSI C all static variables—variables that are allocated at a fixed memory address—have to be initialized by the run-time system to a known value. This value is either an explicit value assigned to the variable, or if no value is given, it is cleared to zero.

In the AVR IAR C/C++ Compiler, there are two exceptions to this rule and they both use keywords for ISO/ANSI C language extensions. Variables declared `__no_init` are not initialized at all. Variables declared `__eeprom` are allocated in the EEPROM memory and because the EEPROM memory can typically be used for storing configuration data—data that need to live through a reset—these variables will not be initialized by the runtime system. Instead, these initializers are stored in a separate segment, which can be loaded to the EEPROM as part of the program download.

Initialization at system startup

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

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

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

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

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

```
NEAR_I           0x1000-0x10FF and 0x1200-0x12FF
```

```
NEAR_ID         0x4000-0x41FF
```

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

```
NEAR_I           0x1000-0x10FF and 0x1200-0x12FF
```

```
NEAR_ID         0x4000-0x40FF and 0x4200-0x42FF
```

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 ship 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, 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 run-time as the cast cannot copy the object from the flash memory to the data memory.

Note: To access strings located in flash, you must use alternative library routines that expect flash strings. A few such alternative functions are provided 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 249.

SEGMENTS FOR STATIC DATA IN THE LINKER COMMAND FILE

As described in the section *Static memory segments*, page 37, static data can be placed in many different segments depending on the application requirements and your target system. In the linker command 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_EEPROM_BASE+_EXT_EEPROM_SIZE

/* The _EXT_EEPROM_BASE and _EXT_EEPROM_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 DATA STACK

The data stack is used by functions to store auto variables, function parameters and temporary storage that is used locally by functions, as described in the chapter *Data storage*. It is a continuous block of memory pointed to by the processor stack pointer register `Y`.

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 when you use the command line interface compared to when you use the IAR Embedded Workbench IDE.



Data stack size allocation in IAR Embedded Workbench

Select **Project>Options**. In the **General Options** category, click the **System** page.

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



Data stack size allocation from the command line

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

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

```
-D_CSTACK_SIZE=size
```

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



Placement of data 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 will be wasted. If the given stack size is too small, there are two things that can happen, depending on where in memory you have located your stack. Both alternatives are likely to result in application failure. Either variable storage 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. Because the second alternative is easier to detect, you should consider placing your stack so that it grows towards the end of the memory.

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.

To determine the size of the return address stack, see *Stack size considerations*, page 42. 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 may 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 181.

Allocating a memory area for the stack is done differently when you use the command line interface compared to when you use the IAR Embedded Workbench IDE.



RSTACK size allocation in IAR Embedded Workbench

Select **Project>Options**. In the **General Options** category, click the **System** page.

Add the required stack size in the **Return address stack** text box.



RSTACK size allocation from the command line

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

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

```
-D_RSTACK_SIZE=size
```

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



Placement of the 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 use of the C function `malloc` (or one of its relatives) or the C++ operator `new`.

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

- Linker segment used for the heap, which differs between the DLIB and the CLIB runtime environment
- Allocating the heap size, which differs depending on which build interface you are using
- Placing the heap segments in memory.

Heap segments in DLIB

Heaps can be placed in the following memory types:

Memory type	Segment name	Memory attribute
Tiny	TINY_HEAP*	__tiny
Near	NEAR_HEAP	__near
Far	FAR_HEAP	__far
Huge	HUGE_HEAP	__huge

Table 12: Heaps, memory types, and segments

* The TINY_HEAP segment is only available in the Tiny memory model.

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 information about how to access a heap in a specific memory using C++, see *New and Delete operators*, page 114.

Heap segments in CLIB

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 IAR Embedded Workbench

Select **Project>Options**. In the **General Options** category, click the **Heap configuration** page.

Add the required heap size in the appropriate text box.



Heap size allocation from the command line

The size of the heap segments are defined in the linker command file.

The default linker file sets up a constant, representing the size of each heap, at the beginning of the linker command 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 */
```

Specify the appropriate size for your application.

If your application uses near or far memory, symbols for heaps for these memories should also be defined in the linker command file.



Placement of heap segments

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 the IAR C-SPY Debugger, you are not likely to notice the speed penalty, but it is quite noticeable when the application runs on an 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 has been explicitly placed at an address, for example by using the compiler @ syntax, will be placed in either the `SEGMENT_AC` or the `SEGMENT_AN` segment. The former is used for constant-initialized data, and the latter for items declared as `__no_init`. The individual segment part of the segment knows its location in the memory space, and it *should not* be specified in the linker command file.

USER-DEFINED DATA SEGMENTS

If you create your own data segments—see *Controlling data and function placement*, page 47—these must also be defined in the linker command file using the `-Z` or `-P` segment control directives.

Code segments

This section contains examples and descriptions of the segments used for storing code, in other words, functions, and the interrupt vector table. Typically, these segments are placed in the code memory space (flash).

The `-z` command is used for defining all segments in the following examples. The addresses used in the examples are based on the assumed target system described in the Table 10, *Memory layout of a target system (example)*, on page 35. Note that because the described target system is limited in size of code memory space, several segments supported by the compiler are not applicable for this target system.

For a complete list of all segments, see *Summary of segments*, page 149.

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 derivative 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 the Table 3, *Summary of processor configuration*, on page 8.

In the linker command file it can look like this:

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

USER-DEFINED SEGMENTS

If you create your own segments—see *Controlling data and function placement*, page 47—these must also be defined in the linker command file using the `-Z` or `-P` segment control directives. In the linker command file it can look like this:

```
-Z (CODE) MYSEGMENT=100-2FF
```

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 consist 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.
- The `DIFUNCT` segment is only used when a source file has been compiled in C++ mode and the file contains global objects (class instances). The segment will then contain a number of function pointers that point to constructor code that should be performed for each object.

In the linker command file it can look like this:

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

Efficient usage of segments and memory

This section lists several features and methods to help you manage memory and segments.

CONTROLLING DATA AND FUNCTION PLACEMENT

The `@` operator, alternatively the `#pragma location` directive, can be used for placing global and static variables at absolute addresses. The syntax can also be used for placing variables or functions in named segments. The variables must be declared either `__no_init` or `const`. If declared `const`, it is legal for them to have initializers. The named segment can either be a predefined segment, or a user-defined segment.

Note: Take care when explicitly placing a variable or function in a predefined segment other than the one used by default. This is possible and 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.

C++ static member variables can be placed at an absolute address or in named segments, just like any other static variable.

Variables and functions can also be placed into named segments using the `--segment` option, in which case you can override the default segment base name. Note that if you use this method, the object does not need to be declared neither `__no_init` nor `const` as it is only the segment name that will be modified.

Data placement at an absolute location

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.

Example

```
__no_init char alpha @ 0x2000;      /* OK */

#pragma location=0x2002
const int beta=5;                  /* OK */

const int gamma @ 0x2004 = 3;      /* OK */

int delta @ 0x2006;                /* Error, neither */
                                   /* "__no_init" nor "const".*/
```

See *Located data*, page 46 for information about how to handle this in the linker command file.

Note: A variable placed in an absolute location should be defined in an include file, to be included in every module that uses the variable. An unused definition in a module will be ignored. A normal `extern` declaration—one that does not use an absolute placement directive—can refer to a variable at an absolute address; however, optimizations based on the knowledge of the absolute address cannot be performed.

Data placement into named segments

It is possible to place variables into named segments using either the `@` operator or the `#pragma location` directive. A string should be used for specifying the segment name.

Example

```

__no_init int alpha @ "MYSEGMENT"; /* OK */

#pragma location="MYSEGMENT"
const int beta=5;                  /* OK */

const int gamma @ "MYSEGMENT" = 3; /* OK */

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

```

Function placement into named segments

It is possible to place functions into named segments using either the @ operator or the #pragma location directive. When placing functions into segments, the segment is specified as a string literal.

Example

```

void f(void) @ "MYSEGMENT";
void g(void) @ "MYSEGMENT"
{
}

#pragma location="MYSEGMENT"
void h(void);

```

Declaring located variables extern

Using IAR extensions in C, read-only SFRs—for instance, in header files—can be declared like this:

```
volatile const __no_init int x @ 0x100;
```

In C++, const variables are static (module local), which means that each module with this declaration will contain a separate variable. When you link an application with several such modules, the linker will report that there are more than one variable located at address 0x100.

To avoid this problem and have it work the same way in C and C++, you should declare these SFRs extern, for example:

```
extern volatile const __no_init int x @ 0x100;
```

USING USER-DEFINED SEGMENTS

In addition to the predefined segments, you can use your own segments. This is useful if you need to have precise control of placement of individual variables or functions.

A typical situation where this can be useful is if you need to optimize accesses to code and data that is frequently used, and place it in a different physical memory.

To use your own segments, use the `#pragma location` directive, or the `--segment` option.

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

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

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

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

LINKER MAP FILE

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

- A segment map which lists all segments in dump order
- A module map which lists all segments, local symbols, and entries (public symbols) for every module in the program. All symbols not included in the output can also be listed
- 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 IAR Embedded Workbench, or the option `-x` on the command line, and one of their suboptions to generate a linker listing.

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

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

MANAGING MULTIPLE MEMORY SPACES

Output formats that do not support more than one memory space—like `MOTOROLA` and `INTEL-HEX`—may 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 the `DATA` memory space, the output format cannot represent this, and the linker issues the following 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, make a copy of the linker command file for the derivative and memory model you are using, and put it in your project directory. Use this copy in your project and add the following 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 have chosen does not support more than one memory space). Use the XLINK option `-O` for this purpose.

For each additional output file, you have to 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 AVR IAR C/C++ Compiler to generate multiple output files. 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 modify it—setting options, overriding default library modules, or building your own library—to optimize it for your application.

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

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

For information about the CLIB runtime environment, 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. The IAR DLIB runtime environment can be used as is together with the IAR C-SPY Debugger. However, to be able to run the application on hardware, you must adapt the runtime environment.

This section gives an overview of:

- The runtime environment and its components
- Library selection.

RUNTIME ENVIRONMENT FUNCTIONALITY

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

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

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

- Support for hardware features:
 - Direct access to low-level processor operations by means of *intrinsic* functions, such as functions for register handling
 - Peripheral unit registers and interrupt definitions in include files
 - Special compiler support for accessing strings in flash memory, see *AVR-specific library functions*, page 249
- Runtime environment support, that is, startup and exit code and low-level interface to some library functions.

Some parts, like the startup and exit code and the size of the heaps must be tailored for the specific hardware and application requirements.

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

LIBRARY SELECTION

To configure the most code-efficient runtime environment, you must determine your application and hardware requirements. The more functionality you need, the larger your code will get.

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

- Setting library options, for example, for choosing `scanf` input and `printf` output formatters, and for specifying the size of the stack and the heap
- Overriding certain library functions, for example `cstartup.s90`, with your own customized versions
- Choosing the level of support for certain standard library functionality, for example, locale, file descriptors, and multibytes, by choosing a *library configuration*: normal or full.

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

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

SITUATIONS THAT REQUIRE LIBRARY BUILDING

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

You must build your own library when:

- 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.

For information about how to build a customized library, see *Building and using a customized library*, page 62.

LIBRARY CONFIGURATIONS

It is possible to configure the level of support for, for example, locale, file descriptors, multibytes. The runtime library configuration is defined in the *library configuration file*. It contains information about what functionality is part of the runtime environment. The configuration file is used for tailoring a build of a runtime library, as well as tailoring the system header files used when compiling your application. The less functionality you need in the runtime environment, the smaller it is.

The following DLIB library configurations are available:

Library configuration	Description
Normal DLIB	No locale interface, C locale, no file descriptor support, no multibyte characters in <code>printf</code> and <code>scanf</code> , and no hex floats 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 hex floats in <code>strtod</code> .

Table 13: Library configurations

In addition to these configurations, you can define your own configurations, which means that you must modify the configuration file. Note that the library configuration file describes how a library was built and thus cannot be changed unless you rebuild the library. For further information, see *Building and using a customized library*, page 62.

The prebuilt libraries are based on the default configurations, see Table 15, *Prebuilt libraries*, page 57. There is also a ready-made library project template that you can use if you want to rebuild the runtime library.

DEBUG SUPPORT IN THE RUNTIME LIBRARY

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

The following table describes the different levels of debugging support:

Debugging support	Linker option in IAR Embedded Workbench	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 <code>stdin</code> and <code>stdout</code> 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 14: Levels of debugging support in runtime libraries

If you build your application project with the XLINK options **With runtime control modules** or **With I/O emulation modules**, certain functions in the library will be replaced by functions that communicate with the IAR C-SPY Debugger. For further information, see *C-SPY Debugger runtime interface*, page 76.



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

Using a prebuilt library

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

- 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.

For the AVR IAR C/C++ Compiler and the Normal library configuration, there are prebuilt runtime libraries for all combinations of these options. For the Full library configuration there is one prebuilt runtime library delivered. The following table shows the names of the libraries and how they reflect the used settings:

Library file	Generic processor option	Generic processor option	Memory model	Enhanced core	Small flash	64-bit doubles	Library configuration
<code>dlavr-3s-ec-sf-n.r90</code>	<code>-v3</code>	<code>-v3</code>	Small	X	X	--	Normal
<code>dlavr-3s-ec-64-f.r90</code>	<code>-v3</code>	<code>-v3</code>	Small	X	--	X	Full

Table 15: Prebuilt libraries

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

```
<library><target>-<cpu><memory_model>-<enhanced_core>-<small_flash>-<64-bit_doubles>-<library_configuration>.r90
```

where

- `<library>` is `dl` for the IAR DLIB Library, or `cl` for the IAR CLIB Library, respectively (for a list of CLIB library files, see *Runtime environment*, page 85)
- `<target>` is `avr`
- `<cpu>` is a value from 0 to 6, matching the `-v` option
- `<memory_model>` is either `t`, `s`, or `l` for Tiny, Small, or Large memory model, respectively
- `<enhanced_core>` is `ec` when enhanced core is used. When the enhanced core is not used, this value is not specified
- `<small_flash>` is `sf` when the small flash memory is available. When small flash memory is not available, this value is not specified
- `<64-bit_doubles>` is `64` when 64-bit doubles are used. When 32-bit doubles are used, this value is not specified
- `<library_configuration>` is one of `n` or `f` for normal and full, respectively.

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



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



On the command line, you must specify the following items:

- Specify which library object file to use on the XLINK command line, for instance:
`dlavr-3s-ec-64-f.r90`
- Specify the include paths for the compiler and assembler:
`-I avr\inc`
- Specify the library configuration file for the compiler:
`--dlib_config C:\...\dlavr-3s-ec-64-f.h`

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

CUSTOMIZING A PREBUILT LIBRARY WITHOUT REBUILDING

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

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

The following items can be customized:

Items that can be customized	Described on page
Formatters for <code>printf</code> and <code>scanf</code>	<i>Choosing formatters for <code>printf</code> and <code>scanf</code></i> , page 59
Startup and termination code	<i>System startup and termination</i> , page 64
Low-level input and output	<i>Standard streams for input and output</i> , page 67
File input and output	<i>File input and output</i> , page 70
Low-level environment functions	<i>Environment interaction</i> , page 73
Low-level signal functions	<i>Signal and raise</i> , page 73
Low-level time functions	<i>Time</i> , page 74
Size of heaps, stacks, and segments	<i>Placing code and data</i> , page 33

Table 16: Customizable items

For a description about how to override library modules, see *Overriding library modules*, page 61.

Choosing formatters for printf and scanf

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

Note:

- If you rebuild the library, it is possible to optimize these functions even further, see *Configuration symbols for printf and scanf*, page 69
- For information about how to choose formatter for the AVR-specific functions `printf_P` and `scanf_P`, see *AVR-specific library functions*, page 249.

CHOOSING PRINTF FORMATTER

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

The following table summarizes the capabilities of the different formatters:

Formatting capabilities	<code>_PrintfFull</code>	<code>_PrintfLarge</code>	<code>_PrintfSmall</code> (default)	<code>_PrintfTiny</code>
Basic specifiers <code>c</code> , <code>d</code> , <code>i</code> , <code>o</code> , <code>p</code> , <code>s</code> , <code>u</code> , <code>X</code> , <code>x</code> , and <code>%</code>	Yes	Yes	Yes	Yes
Multibyte support	*	*	*	No
Floating-point specifiers <code>a</code> , and <code>A</code>	Yes	No	No	No
Floating-point specifiers <code>e</code> , <code>E</code> , <code>f</code> , <code>F</code> , <code>g</code> , and <code>G</code>	Yes	Yes	No	No
Conversion specifier <code>n</code>	Yes	Yes	No	No
Format flag space, <code>+</code> , <code>-</code> , <code>#</code> , and <code>0</code>	Yes	Yes	Yes	No
Length modifiers <code>h</code> , <code>l</code> , <code>L</code> , <code>s</code> , <code>t</code> , and <code>Z</code>	Yes	Yes	Yes	No
Field width and precision, including <code>*</code>	Yes	Yes	Yes	No
<code>long long</code> support	Yes	Yes	No	No

Table 17: Formatters for printf

* Depends on which library configuration is used.

For information about how to fine-tune the formatting capabilities even further, see *Configuration symbols for printf and scanf*, page 69.



Specifying print formatter in IAR Embedded Workbench

To specify the `printf` formatter in IAR Embedded Workbench, choose **Project>Options** and select the **General Options** category. Select the appropriate option on the **Library options** page.



Specifying printf formatter from the command line

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

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

CHOOSING SCANF FORMATTER

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

The following table summarizes the capabilities of the different formatters:

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

Table 18: Formatters for `scanf`

* Depends on which library configuration that is used.

For information about how to fine-tune the formatting capabilities even further, see *Configuration symbols for printf and scanf*, page 69.



Specifying scanf formatter in IAR Embedded Workbench

To specify the `scanf` formatter in IAR Embedded Workbench, choose **Project>Options** and select the **General Options** category. Select the appropriate option on the **Library options** page.



Specifying scanf formatter from the command line

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

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

Overriding library modules

The library contains modules which you probably need to override with your own customized modules, for example functions for character-based I/O and `cstartup`. 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 IAR Embedded Workbench

This procedure is applicable to any source file in the library, which means `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 `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, include paths, and library configuration file as for the rest of the project:

```
iccavr library_module
```

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

- 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 dlavr-3s-ec-64-n.r90
```

Make sure that `library_module` is located 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

In some situations, see *Situations that require library building*, page 55, it is necessary to rebuild the library. In those cases you need to:

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

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

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

SETTING UP A LIBRARY PROJECT

IAR Embedded Workbench provides a library project template which can be used for customizing the runtime environment configuration. This library template has full library configuration, see Table 13, *Library configurations*, page 55.



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

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

MODIFYING THE LIBRARY FUNCTIONALITY

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

The library functionality is determined by a set of *configuration symbols*. The default values of these symbols are defined in the file `Dlib_defaults.h`. This read-only file describes the configuration possibilities. In addition, your library has its own library configuration file `dlavrCustom.h`, which sets up that specific library with full library configuration. For more information, see Table 16, *Customizable items*, page 58.

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

Modifying the library configuration file

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



In IAR Embedded Workbench you must perform the following steps:

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

System startup and termination

This section describes the runtime environment actions performed during startup and termination of applications. The following figure gives a graphical overview of the startup and exit sequences:

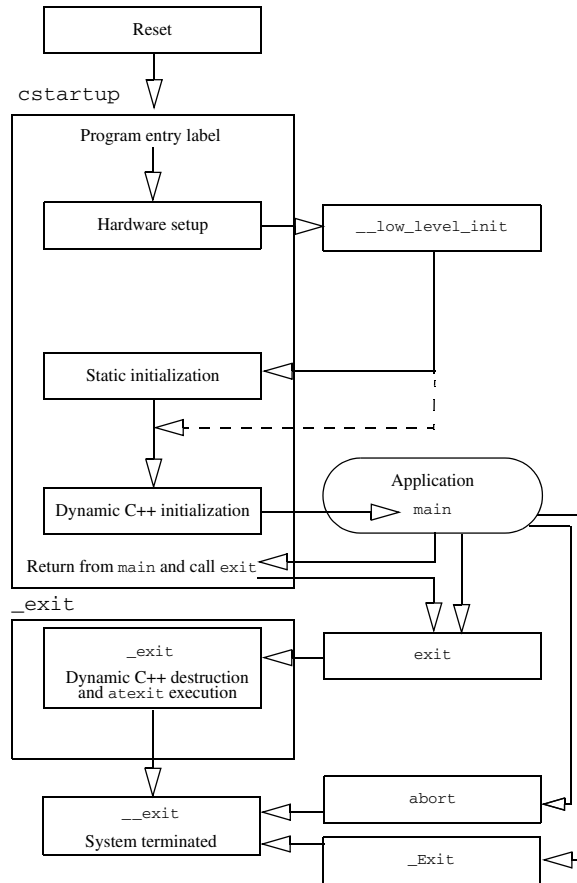


Figure 1: Startup and exit sequences

The code for handling startup and termination is located in the source files `cstartup.s90` and `__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:

- When the cpu is reset it will jump to the program entry label `__program_start` in the system startup code.
- Enables the external data and address buses if needed
- Initializes the stack pointers to the end of `CSTACK` and `RSTACK`, respectively
- The function `__low_level_init` is called, giving the application a chance to perform early initializations
- 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 rest of the initialized variables depending on the return value of `__low_level_init`
- Static C++ objects are constructed
- The `main` function is called, which starts the application.

SYSTEM TERMINATION

An application can terminate normally in two different ways:

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

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

The default `exit` function is written in C. It calls a small function `_exit`, also written in C, that will perform the following operations:

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

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

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

C-SPY interface to system termination

If your project is linked with the XLINK options **With runtime control modules** or **With I/O emulation modules**, the normal `__exit` and `abort` functions are replaced with special ones. C-SPY will then recognize when those functions are called and can take appropriate actions to simulate program termination. For more information, see *C-SPY Debugger runtime interface*, page 76.

Customizing system initialization

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

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

Note: Normally, there is no need for customizing either of the files `cmain.s90` or `cexit.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 62.

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

Some applications may need to initialize I/O registers, omit the default initialization of data segments performed by the system startup code, or set up for use of external memory.

You can do this by providing a customized version of 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 61.

Standard streams for input and output

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

There are primitive I/O functions, which are the fundamental functions through which C 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` 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 62. Note that customizing the low-level routines for input and output does not require you to rebuild the library.

Note: If you write your own variants of `__read` or `__write`, special considerations for the C-SPY runtime interface are needed, see *C-SPY Debugger runtime interface*, page 76.

Example of using `__write` and `__read`

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

```
__no_init volatile unsigned char LCD_IO @ address;

size_t __write(int Handle, const unsigned char * Buf,
              size_t Bufsize)
```

```

{
  int nChars = 0;
  /* Check for stdout and stderr
     (only necessary if file descriptors are enabled.) */
  if (Handle != 1 && Handle != 2)
  {
    return -1;
  }
  for (/*Empty */; Bufsize > 0; --Bufsize)
  {
    LCD_IO = * Buf++;
    ++nChars;
  }
  return nChars;
}

```

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

```

__no_init volatile unsigned char KB_IO @ 0xD2;

size_t __read(int Handle, unsigned char *Buf, size_t BufSize)
{
  int nChars = 0;
  /* Check for stdin
     (only necessary if FILE descriptors are enabled) */
  if (Handle != 0)
  {
    return -1;
  }
  for (/*Empty*/; BufSize > 0; --BufSize)
  {
    int c = KB_IO;
    if (c < 0)
      break;
    *Buf++ = c;
    ++nChars;
  }
  return nChars;
}

```

For information about the @ operator, see *Controlling data and function placement*, page 47.

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 59.

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

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

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

Printf configuration symbols	Includes support for
<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 floats
<code>_DLIB_PRINTF_SPECIFIER_N</code>	Output count (%n)
<code>_DLIB_PRINTF_QUALIFIERS</code>	Qualifiers h, l, L, v, t, and z
<code>_DLIB_PRINTF_FLAGS</code>	Flags -, +, #, and 0
<code>_DLIB_PRINTF_WIDTH_AND_PRECISION</code>	Width and precision
<code>_DLIB_PRINTF_CHAR_BY_CHAR</code>	Output char by char or buffered

Table 19: Descriptions of printf configuration symbols

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

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

Table 20: Descriptions of scanf configuration symbols

CUSTOMIZING FORMATTING CAPABILITIES

To customize the formatting capabilities, you need to set up a library project, see *Building and using a customized library*, page 62. Define the configuration symbols according to your application requirements.

File input and output

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

Note that file I/O capability in the library is only supported by libraries with full library configuration, see *Library configurations*, page 55. In other words, file I/O is supported when the configuration symbol `__DLIB_FILE_DESCRIPTOR` is enabled. If not enabled, functions taking a *FILE* * argument cannot be used.

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

I/O function	File	Description
<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 21: Low-level I/O files

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

Note: If you link your library with I/O debugging support, C-SPY variants of the low-level I/O functions will be linked for interaction with C-SPY. For more information, see *Debug support in the runtime library*, page 56.

Locale

Locale is a part of the C language that allows language- and country-specific settings for a number of areas, such as currency symbols, date and time, and multibyte 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 major 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 supports the C locale only
- 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 encoding during runtime.
- Libraries with *normal library configuration* do not have support for the locale interface.

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

CUSTOMIZING THE LOCALE SUPPORT

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

- The standard C locale
- The POSIX locale
- A wide range of international 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.

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 62.

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 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 encoding:

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

Environment interaction

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

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

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

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

```
key=value\0
```

The last string must be empty. Assign the created sequence of strings to the `__environ` variable.

For example:

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

If you need a more sophisticated environment variable handling, you should implement your own `getenv`, and possibly `putenv` function. This does not require that you rebuild the library. You can find source templates in the files `getenv.c` and `environ.c` in the `avr\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 61.

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

If you decide to rebuild the library, you can find source templates in the library project template. For further information, see *Building and using a customized library*, page 62.

Note: If you link your application with support for I/O debugging, the functions `getenv` and `system` will be replaced by C-SPY variants. For further information, see *Debug support in the runtime library*, page 56.

Signal and raise

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

This does not require that you rebuild the library. You can find source templates in the files `Signal.c` and `Raise.c` in the `avr\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 61.

If you decide to rebuild the library, you can find source templates in the library project template. For further information, see *Building and using a customized library*, page 62.

Time

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

This does not require that you rebuild the library. You can find source templates in the files `Clock.c` and `Time.c`, and `Getzone.c` in the `avr\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 61.

If you decide to rebuild the library, you can find source templates in the library project template. For further information, see *Building and using a customized library*, page 62.

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

Note: If you link your application with support for I/O debugging, the functions `clock` and `time` will be replaced by C-SPY variants that return the host clock and time respectively. For further information, see *C-SPY Debugger runtime interface*, page 76.

Strtod

The function `strtod` does not accept hexadecimal floating-point strings in libraries with the normal library configuration. To make a library do so, you need to rebuild the library, see *Building and using a customized library*, page 62. Enable the configuration symbol `_DLIB_STRTOD_HEX_FLOAT` in the library configuration file.

Assert

If you have linked your application with support for runtime debugging, C-SPY will be notified about failed asserts. If this is not the behavior you require, you must add the source file `xReportAssert.c` to your application project. Alternatively, you can rebuild the library. The `__ReportAssert` function generates the assert notification. You can find template code in the `avr\src` directory. For further information, see *Building and using a customized library*, page 62. To turn off assertions, you must define the symbol `NDEBUG`.



In the IAR Embedded Workbench, 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.

Heaps

The runtime environment supports heaps in the following memory types:

Memory type	Segment name	Extended keyword	Used by default in memory model
Tiny	TINY_HEAP	<code>__tiny</code>	Tiny
Near	NEAR_HEAP	<code>__near</code>	Small
Far	FAR_HEAP	<code>__far</code>	Medium
Huge	HUGE_HEAP	<code>__huge</code>	Large

Table 22: Heaps and memory types

See *The heap*, page 44 for information about how to set the size for each heap. To use a specific heap, the prefix in the table is the extended keyword to use in front of `malloc`, `free`, `calloc`, and `realloc`. 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 114.

C-SPY Debugger runtime interface

To include support for runtime and I/O debugging, you must link your application with the XLINK options **With runtime control modules** or **With I/O emulation modules**, see *Debug support in the runtime library*, page 56. In this case, C-SPY variants of the following library functions will be linked to the application:

Function	Description
<code>abort</code>	C-SPY notifies that the application has called <code>abort</code> *
<code>__exit</code>	C-SPY notifies that the end of the application has been reached *
<code>__read</code>	<code>stdin</code> , <code>stdout</code> , and <code>stderr</code> will be directed to the Terminal I/O window; all other files will read the associated host file
<code>__write</code>	<code>stdin</code> , <code>stdout</code> , and <code>stderr</code> will be directed to the Terminal I/O window, all other files will write to the associated host file
<code>__open</code>	Opens a file on the host computer
<code>__close</code>	Closes the associated host file on the host computer
<code>__seek</code>	Seeks in the associated host file on the host computer
<code>remove</code>	Writes a message to the Debug Log window and returns -1
<code>rename</code>	Writes a message to the Debug Log window and returns -1
<code>time</code>	Returns the time on the host computer
<code>clock</code>	Returns the clock on the host computer
<code>system</code>	Writes a message to the Debug Log window and returns -1
<code>_ReportAssert</code>	Handles failed asserts *

Table 23: Functions with special meanings when linked with debug info

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

LOW-LEVEL DEBUGGER RUNTIME INTERFACE

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

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

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

THE DEBUGGER TERMINAL I/O WINDOW

To make the Terminal I/O window available, the application must be linked with support for I/O debugging, see *Debug support in the runtime library*, page 56. 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 *AVR® IAR Embedded Workbench™ IDE User Guide* for more information about the Terminal I/O window.

Checking module consistency

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

When developing an application, it is important to ensure that incompatible modules are not used together. For example, in the AVR IAR C/C++ 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 use a set of predefined runtime model attributes. You can use these predefined attributes or define your own to perform any type of consistency check.

RUNTIME MODEL ATTRIBUTES

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

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

Example

In the following table, the object files could (but do not have to) define the two runtime attributes `color` and `taste`. In this case, `file1` cannot be linked with any of the other files, since the runtime attribute `color` does not match. Also, `file4` and `file5` cannot be linked together, because the `taste` runtime attribute does not match.

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

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

Table 24: Example of runtime model attributes

USING RUNTIME MODEL ATTRIBUTES

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

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

For detailed syntax information, see *#pragma rtmodel*, page 223.

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

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

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

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 AVR IAR C/C++ Compiler. These can be included in assembler code or in mixed C or C++ and assembler code.

Runtime model attribute	Value	Description
<code>__rt_version</code>	2.30	This runtime key is always present in all modules generated by the AVR IAR C/C++ 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.
<code>__cpu_name</code>	<i>derivative</i>	Corresponds to the processor derivative name, as it appears in Table 2, <i>Mapping of processor options</i> , page 6, for example AT90S2343 or ATmega8515. Note that for the FpSLic derivative, the value is AT94Kxx.
<code>__double_size</code>	32 or 64	States the size of double. The default size is 32. Use the compiler option <code>--64bit_doubles</code> to override the default.
<code>__enhanced_core</code>	Enabled	Available only if the compiler option <code>--enhanced_core</code> or <code>--cpu</code> for a target processor with enhanced core is used.
<code>__memory_model</code>	1–3	Corresponds to the used memory model, where the value can be 1, 2, or 3 for Tiny, Small, or Large, respectively.
<code>__no_rampd</code>	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

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

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

Examples

For an example of using the runtime model attribute `__rt_version` for checking module consistency on used calling convention, see *Hints for using the new calling convention*, page 100.

The following assembler source code provides a function, `part2`, that counts the number of times it has been called by increasing the register `R4`. The routine assumes that the application does not use `R4` for anything else, that is, the register has been locked for usage. To ensure this, a runtime module attribute, `__reg_r4`, has been defined with a value `counter`. This definition will ensure that this specific module can only be linked with either other modules containing the same definition, or with modules that do not set this attribute. Note that the compiler sets this attribute to `free`, unless the register is locked.

```

                                RTMODEL    "__reg_r4", "counter"
                                MODULE      myCounter
                                PUBLIC      myCounter
                                RSEG        CODE:CODE:NOROOT(1)
myCounter: INC                    R4
                                RET
                                ENDMOD
                                END

```

If this module is used in an application that contains modules where the register `R4` has not been locked, an error is issued by the linker:

```

Error[e117]: Incompatible runtime models. Module myCounter
specifies that '__reg_r4' must be 'counter', but module part1
has the value 'free'

```

USER-DEFINED RUNTIME MODEL ATTRIBUTES

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

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

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

Implementation of system startup code

This section presents some general techniques used in the system startup code, including background information that might be useful if you need to modify it.

Note: Do not modify the file `cstartup.s90` unless required by your application. Your first option should be to use a customized version of `__low_level_init` for initialization code.

The source files are well commented and are not described in detail in this guide.

For information about assembler source files, see the *AVR® IAR Assembler Reference Guide*.

MODULES AND SEGMENT PARTS

To understand how the startup code is designed, you must have a clear understanding of modules and segment parts, and how the IAR XLINK Linker treats them.

An assembler module starts with a `MODULE` directive and ends with an `ENDMOD` directive. Each module is logically divided into segment parts, which are the smallest linkable units. There will be segment parts for constants, code bytes, and for reserved space for data. Each segment part begins with an `RSEG` directive.

When XLINK builds an application, it starts with a small number of modules that have either been declared using the `__root` keyword or have the program entry label `__program_start`. The linker then continues to include all modules that are referred from the already included modules. XLINK then discards unused segment parts.

Segment parts, REQUIRE, and the falling-through trick

The system startup code has been designed to use segment parts so that as little as possible of unused code will be included in the linked application.

A piece of code or data is not included if it is not used or referred to. To make the linker always include a piece of code or data, the assembler directive `REQUIRE` can be used.

The segment parts defined in the system startup code are guaranteed to be placed immediately after each other. There are two reasons for this. First, the alignment requirement of the segment parts is every two bytes. Because the size of all assembler instructions are multiples of two, this does not allow pad bytes to be placed between code sections. Second, XLINK will not change the order of the segment parts or modules, because the segments holding the system startup code are placed using the `-z` option.

This lets the system startup code specify code in subsequent segment parts and modules that are designed so that some of the parts may not be included by XLINK. The code simply falls through to the next piece of code not discarded by the linker. The following example shows this technique:

```

MODULE doSomething

RSEG MYSEG:CODE:NOROOT(1) // First segment part.
PUBLIC ?do_something
EXTERN ?end_of_test
REQUIRE ?end_of_test

?do_something: // This will be included if someone refers to
...           // ?do_something. If this is included then
              // the REQUIRE directive above ensures that
              // the JUMP instruction below is included.

JMP ?end_of_test
RSEG MYSEG:CODE:NOROOT(1) // Second segment part.
PUBLIC ?do_something_else

?do_something_else:
... // This will only be included in the linked
    // application if someone outside this function
    // refers to or requires ?do_something_else

RSEG MYSEG:CODE:NOROOT(1) // Third segment part.
PUBLIC ?end_of_test

?end_of_test:
RET // This is included if ?do_something above
    // is included.

ENDMOD

```

Added C functionality

The IAR DLIB Library includes some added C functionality, partly taken from the C99 standard.

The following include files provide these features:

- `stdint.h`
- `stdbool.h`
- `math.h`
- `stdio.h`

- `stdlib.h`

STDINT.H

This include file provides integer characteristics.

STDBOOL.H

This include file makes the `bool` type available if the **Allow IAR extensions** (`-e`) option is used.

MATH.H

In `math.h` all functions exist in a `float` variant and a `long double` variant, suffixed by `f` and `l` respectively. For example, `sinf` and `sinl`.

STDIO.H

In `stdio.h`, the following functions have been added from the C99 standard:

<code>vscanf,</code> <code>vfscanf,</code> <code>vsscanf,</code> <code>vsnprintf</code>	Variants that have a <code>va_list</code> as argument.
<code>snprintf</code>	Same as <code>sprintf</code> , but writes to a size limited array.

The following functions have been added to provide I/O functionality for libraries built without `FILE` support:

<code>__write_array</code>	Corresponds to <code>fwrite</code> on <code>stdout</code> .
<code>__ungetchar</code>	Corresponds to <code>ungetc</code> on <code>stdout</code> .
<code>__gets</code>	Corresponds to <code>fgets</code> on <code>stdin</code> .

STDLIB.H

In `stdlib.h`, the following functions have been added:

<code>_exit</code>	Exits without closing files et cetera.
<code>__qsorttbl</code>	A <code>qsort</code> function that uses the bubble sort algorithm. Useful for applications that have limited stack.

PRINTF, SCANF AND STRTOD

The functions `printf`, `scanf` and `strtod` have added functionality from the C99 standard. For reference information about these functions, see the library reference available from the **Help** menu.

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 *AVR® IAR Embedded Workbench Migration Guide*.

Runtime environment

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 detailed reference information about the runtime libraries, see the chapter *Library functions*.

IAR Embedded Workbench comes with a set of prebuilt runtime libraries, which are configured for different combinations of the following 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`).

For the AVR IAR C/C++ Compiler, this means there are prebuilt runtime libraries for different combination of these options. The following table shows the names of the libraries and how they reflect the used settings:

Library file	Generic processor option	Memory model	Enhanced core	Small flash	64-bit doubles
<code>c10t.r90</code>	<code>-v0</code>	Tiny	--	--	--
<code>c11s-64.r90</code>	<code>-v1</code>	Small	--	--	X
<code>c161-ec-64.r90</code>	<code>-v6</code>	Large	X	--	X

Table 26: Prebuilt libraries

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

```
<library><cpu><memory_model>-<enhanced_core>-<small_flash>-<64-bit_doubles>.r90
```

where

- `<library>` is `c1` for the IAR CLIB Library, or `d1` for the IAR DLIB Library, respectively (for a list of DLIB library files, see *Using a prebuilt library*, page 56)
- `<cpu>` is a value from 0 to 6, matching the `-v` option
- `<memory_model>` is either `t`, `s`, or `l` for Tiny, Small, or Large memory model, respectively
- `<enhanced_core>` is `ec` when enhanced core is used. When the enhanced core is not used, this value is not specified
- `<small_flash>` is `sf` when the small flash memory is available. When small flash memory is not available, this value is not specified
- `<64-bit_doubles>` is `64` when 64-bit doubles are used. When 32-bit doubles are used, this value is not specified.



IAR Embedded Workbench includes the correct runtime library based on the options you select. See the *AVR® IAR Embedded Workbench™ IDE User Guide* for additional information.



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

```
c10t.r90
```

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

The creation of new I/O routines is based on the following 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 DEV_IO @ address;

int putchar(int outchar)
{
    DEV_IO = 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 61.

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 IAR Embedded Workbench

- 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 command 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 may 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 61.

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 IAR Embedded Workbench

- 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 the following line in the linker command 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.s90` and `_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
- Enables the external data and address buses if needed
- Initializes the stack pointers 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 ISO/ANSI 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` in order 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 61 in the chapter *The DLIB runtime environment*.

Customizing system initialization

For information about how to customize system initialization, see *Customizing system initialization*, page 66.

Implementation of cstartup

For information about `cstartup` implementation, see *Implementation of system startup code*, page 81 in the chapter *The DLIB runtime environment*.

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 has typed some input and pressed 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 *AVR® IAR Embedded Workbench™ IDE User 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 77 in the chapter *The DLIB runtime environment*.

Assembler language interface

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

This chapter describes the available methods for this, as well as some C alternatives, with their pros and cons. 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 AVR IAR C/C++ Compiler provides several ways to mix C or C++ and assembler:

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

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

INTRINSIC FUNCTIONS

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

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

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

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

MIXING C AND ASSEMBLER MODULES

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

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

The first issue is discussed in this section. The following two are covered in the section *Calling convention*, page 99.

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 108.

It is possible to write parts of your application in assembler and mix them with your C or C++ modules. There are several benefits with this:

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

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

The recommended method for mixing C or C++ and assembler modules is described in *Calling assembler routines from C*, page 96, and *Calling assembler routines from C++*, page 98, respectively.

INLINE ASSEMBLER

It is possible to insert assembler code directly into a C or C++ function. The `asm` keyword assembles and inserts the supplied assembler statement, or statements, in-line. The following example shows how to use inline assembler to insert assembler instructions directly in the C source code. This example also shows the risks of using inline assembler.

```
bool flag;

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

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

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

Inline assembler sequences have no well-defined interface with the surrounding code generated from your C 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. In addition, there are several limitations to using inline assembler:

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

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

For reference information about the `asm` keyword, see *asm*, *__asm*, page 205.

Calling assembler routines from C

An assembler routine that is to be called from C must:

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

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

or

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

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

CREATING SKELETON CODE

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

The following example shows how to create skeleton code to which you can easily add the functional body of the routine. The skeleton source code only needs to declare the variables required and perform simple accesses to them. In this example, the assembler routine takes an `int` and a `double`, and then returns an `int`:

```
extern int gInt;
extern double gDouble;

int func(int arg1, double arg2)
{
    int locInt = arg1;
    gInt = arg1;
    gDouble = arg2;
    return locInt;
}

int main()
{
    int locInt = gInt;
    gInt = func(locInt, gDouble);
    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 IAR Embedded Workbench, 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 the following options to compile the skeleton code:

```
iccavr skeleton -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 as well as a low level of optimization.

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 IAR Embedded Workbench, select **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 IAR C-SPY™ Debugger.

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 may therefore be called from C++ when declared in the following manner:

```
extern "C"
{
    int my_routine(int x);
}
```

Memory access layout of non-PODs ("plain old data structures") is not defined, and may change between compiler versions. Therefore, we do not recommend that you access non-PODs from assembler routines.

To achieve the equivalent to a non-static member function, the implicit `this` pointer has to be made explicit:

```
class X;

extern "C"
{
    void doit(X *ptr, int arg);
}
```

It is possible to “wrap” the call to the assembler routine in a member function. Using an inline member function removes the overhead of the extra call—provided that function inlining is enabled:

```
class X
{
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 AVR IAR C/C++ 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 AVR IAR C/C++ Compiler. The following items are looked upon:

- 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

There are two calling conventions to choose between:

- 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.

The new calling convention is used by default. However, you can choose a calling convention by using the `--version1_calls` command line option. You can also declare individual functions to use the old calling convention by using the `__version_1` function attribute, for example:

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

For details about the `--version1_calls` option and the `__version_1` attribute, see `--version1_calls`, page 199 and `__version_1`, page 214, respectively.



Hints for using the new calling convention

The new 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 96.

If you intend to use the new 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 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 as the linker will produce an error if there is a mismatch between the values.

For more information about checking module consistency, see *Checking module consistency*, page 77.

FUNCTION DECLARATIONS

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

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

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

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

C AND C++ LINKAGE

In C++, a function can have either C or C++ linkage. Only functions with C linkage can be implemented in assembler.

The following 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++. The following 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 may 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, the following 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. Any function may use a preserved register for other purposes, but must save the value prior to use and restore it at the exit of the function.

For both calling conventions, the following 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 186 and `__regvar`, page 212.

Special registers

For some registers there are certain prerequisites that you must consider:

- 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 that 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 utilizes any of the registers `EIND`, `RAMPX`, or `RAMPZ`, these registers are treated as scratch registers.

FUNCTION CALL

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 utilize registers as much as possible. The remaining parameters not passed in registers are passed on the stack.

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 the following table:

Parameters	Alignment	Passed in registers
8-bit values	1	<code>R16, R17, R18, R19, R20, R21, R22, R23</code>

Table 27: Registers used for passing parameters

Parameters	Alignment	Passed in registers
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

Table 27: Registers used for passing parameters (Continued)

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.

The following 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.

Register assignment using the new calling convention

In the new 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 new 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 96.

Below follows some examples of combinations of register assignment.

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

```
void foo(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.

Stack parameters

There is only a limited number of registers that can be used for passing parameters; when no more registers are available, the remaining parameters are passed on the stack. In addition, the parameters are passed on the stack in the following cases:

- Structure types: `struct`, `union`, and classes if larger than 4 bytes
- The data type `double` (64-bit floating-point numbers)
- Unnamed parameters to variable length functions; in other words, functions declared as `foo(param1, ...)`, for example `printf`.

Stack layout

A function call creates a stack frame as follows:

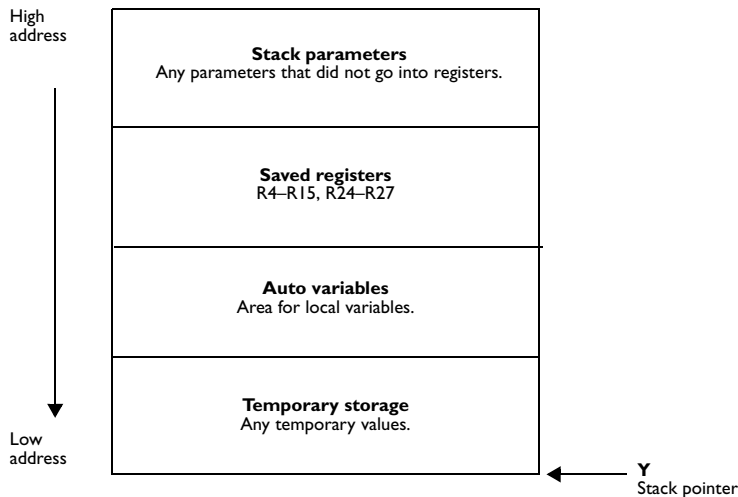


Figure 2: Storing stack parameters in memory

Note that only the registers that are used will be saved.

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 four 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 called by the assembler function.
- 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.

FUNCTION EXIT

The called function exits by deallocating auto variables, restoring registers, deallocating stack parameters, and finally performing a `RET` instruction which pops the return address from the return address stack. Note that an interrupt function returns by performing a `RETI` function.

Return values

A function can return a value to the function or program that called it, or it can be of the type `void`. The return value of a function, if any, can be scalar (such as integers and pointers), floating-point, or a structure. A return value can be passed via register or via the stack.

For both calling conventions, the following details are valid:

- The *return address stack* (RSTACK) and the *data stack* (CSTACK) are two separate stacks. The RSTACK uses the internal I/O port `SP`, which is declared in the `ioderivative.h` include files provided with the product.
- `struct` and `union` values larger than 4 bytes are passed using a pointer; this pointer is returned via register
- The implicit first parameter passed to a function, pointing to the memory to be used for storing the return value, is always passed on the stack if the value is larger than 4 bytes.

Registers used for returning values

For both calling conventions, the registers available for returning values are R16–R19.

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

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 handling

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 returns by using the `RETI` function.

For more information about interrupt functions, see *Interrupt functions*, page 29.

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 30.

EXAMPLES

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

Example 1

Assume that we have the following 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`.

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

```
SUBI    R16,FF
SBCI    R17,FF
```

Example 2

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

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

The calling function must reserve six 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 `struct`.

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

It is the responsibility of the calling function to allocate a memory location for the return value and pass a pointer to it as a hidden first parameter. The pointer to the location where the return value should be stored is passed in the first suitable register/register pair, which is R16, R17:R16, and R18:R17:R16 for the Tiny, Small, and Large memory model, respectively. The parameter *x* is passed in R19:R18, R19:R18, and R21:R20 for the Tiny, Small, and Large memory model, respectively.

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

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

In this case, the return value is a scalar, so there is no hidden parameter. The parameter *x* is passed in R17:R16. The return value is returned in R16, R17:R16, and R18:R17:R16 for the Tiny, Small, and Large memory model, respectively.

Call frame information

When debugging an application using C-SPY, it is possible to view the *call stack*, that is, the functions that have called the current function. The compiler makes this possible by supplying debug information that describes the layout of the call frame, in particular information about where the return address is stored.

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

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

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

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

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

Using C++

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

Overview

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

STANDARD EMBEDDED C++

The following C++ features are supported:

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

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

Standard Embedded C++ lacks the following 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
- 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 `typeid`) are excluded.

Note: The library is not in the `std` namespace, because Embedded C++ does not support namespaces.

IAR EXTENDED EMBEDDED C++

IAR Extended EC++ is a slightly larger subset of C++ which adds the following features to the standard EC++:

- Full template support
- Multiple and virtual inheritance
- Namespace support
- 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 does neither use exceptions and multiple inheritance, nor does it use runtime type information (`rtti`). Moreover, the library is not in the `std` namespace.

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

ENABLING C++ SUPPORT



In the AVR IAR C/C++ Compiler, the default language is C. To be able to compile files written in Embedded C++, you must use the `--ec++` compiler option. See `--ec++`, page 179. You must also use the IAR DLIB runtime library.

To take advantage of *Extended* Embedded C++ features in your source code, you must use the `--eec++` compiler option. See `--eec++`, page 180.



To set the equivalent option in IAR Embedded Workbench, select **Project>Options>C/C++ Compiler>Language**.

Feature descriptions

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

CLASSES

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

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

For further information about attributes, see *Type and object attributes*, page 144.

Example

```
class A {
public:
    static __near int i @ 600; //Located in near at address 600
    static __nearfunc void f(); //Located in nearfunc memory
    __nearfunc void g(); //Located in nearfunc memory
    virtual __nearfunc void h(); //Located in nearfunc memory
};
```

The `this` pointer used for referring to 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 casted to a default data pointer.

Example

```
class B {
public:
    void f();
    int i;
};
```

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 f();           // Has a this pointer of type C __far *
    void f() 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 i;
};
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
C __huge Cc;          // Not allowed, __huge pointer can't be
                        // implicitly casted into a __far pointer
void h()
{
    C Cd;              // Resides on the stack
}
C * Cp;               // Creates a pointer to far memory
C __near * Cp;        // Creates a pointer to near memory
```

Note: 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.

There is a built-in operator that 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 g();
    int j;
};

class __near E : public C { // OK, near memory is inside far
public:
    void g() // Has a this pointer pointing into near memory
    {
        f(); // Gets a this pointer into far memory
    }
    int j;
};

class __huge F : public C { // Not OK, huge memory isn't inside
                           // far memory
public:
    void g();
    int j;
};

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

A `new` expression on the class will allocate memory in the heap residing in 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++.

If a pointer to class memory cannot be implicitly casted into a default pointer type, no temporaries can be created for that class.

For more information about memory types, see *Memory types and memory attributes*, page 18.

FUNCTIONS

A function with `extern "C"` linkage is compatible with a function that has C++ linkage.

Example

```
extern "C" {
    typedef void (*fpC)(void); // A C function typedef
};
void (*fpCpp)(void);          // A C++ function typedef

fpC f1;
fpCpp f2;
void f(fpC);

f(f1);                        // Always works
f(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.

These examples assume that there is a heap in both near and far memory.

```
void __near * operator new __near (__near_size_t);
void __tiny * operator new __tiny (__tiny_size_t);
void operator delete(void __near *);
void operator delete(void __tiny *);
```

And correspondingly for array `new` and `delete` operators:

```
void __near * operator new[] __near (__near_size_t);
void __tiny * operator new[] __tiny (__tiny_size_t);
void operator delete[](void __near *);
void operator delete[](void __tiny *);
```


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,

```
//Calls operator new __near(__near_size_t)
int __tiny *p = new __tiny int;

//Calls operator new __near(__near_size_t)
int __near *q = new int __near;

//Calls operator new[] __data16(__data16_size_t)
int __near *r = new __near int[10];

//Calls operator new __tiny(__tiny_size_t)
class __tiny S{...};
S *s = new S;
```

A `delete` expression calls the `operator delete` function that corresponds to the argument given. For example,

```
delete p; //Calls operator delete(void __near *)
delete s; //Calls operator delete(void __tiny *)
```

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. For example,

```
int __near * t = new __tiny int;
delete t; //Error: Causes a corrupt heap
```

TEMPLATES

Extended EC++ supports templates according to the C++ standard, except for the support of the `export` keyword. The implementation uses a two-phase lookup which means that the keyword `typename` has to 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 have to be in include files or in the actual source file.

Templates and data memory attributes

For data memory attributes to work as expected in templates, two elements of the standard C++ template handling have been 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

```
template<typename T> void fun(T *);

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

Note that line 3 above gets a different result than the analogous situation with class template specializations.

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. In order 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

```
#pragma basic_template_matching
template<typename T> void fun(T *);

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 __io x;

template<__io int &y>
void foo()
{
    y = 17;
}

void bar()
{
    foo<x>();
}
```

The standard template library

The STL (standard template library) delivered with the product is tailored for Extended EC++, as described in *IAR Extended Embedded C++*, page 110.

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

```
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
vector<int __near> __huge z; // Illegal
```

Note 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>` where `mem` is the memory type of `T`. Supplying a key with a memory type is not useful.

Note that two containers that only differ by the data memory attribute they use cannot be assigned to each other.

Example

```
vector<int __near> x;
vector<int __huge> y;

x = y; // Illegal
y = x; // Illegal
```

However, the templated assign member method will work:

```
x.assign(y.begin(), y.end());
y.assign(x.begin(), x.end());
```

STL and the IAR C-SPY Debugger

C-SPY has built-in display support for the STL containers.

VARIANTS OF CASTS

In Extended EC++ the following additional C++ cast variants can be used:

```
const_cast<t2>(t), static_cast<t2>(t), reinterpret_cast<t2>(t).
```

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 // Nothing here
```

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;
```

USING INTERRUPTS AND EC++ DESTRUCTORS

If interrupts are enabled and the interrupt functions use class objects that have destructors, there may be problems if the program exits either by using `exit` or by returning from `main`. If an interrupt occurs after an object has been destroyed, there is no guarantee that the program will work properly.

To avoid this, you must override the function `exit(int)`.

The standard implementation of this function (located in the file `exit.c`) looks like this:

```
extern void _exit(int arg);
void exit(int arg)
{
    _exit(arg);
}
```

`_exit(int)` is responsible for calling the destructors of global class objects before ending the program.

To avoid interrupts, place a call to the intrinsic function `__disable_interrupt` before the call to `_exit`.

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.

This chapter gives an overview about how to write code that compiles to efficient code for an embedded application. The issues discussed are:

- Taking advantage of the compilation system
- Selecting data types and placing data in memory
- Writing efficient code.

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

Taking advantage of the compilation system

Largely, the compiler determines what size the executable code for the application will be. The compiler performs many transformations on a program in order to generate the best possible code. Examples of such transformations are storing values in registers instead of memory, removing superfluous code, reordering computations in a more efficient order, and replacing arithmetic operations by cheaper operations.

The linker should also be considered an integral part of the compilation system, since there are some optimizations that are performed by the linker. For instance, all unused functions and variables are removed and not included in the final object file. It is also as input to the linker you specify the memory layout. For detailed information about how to design the linker command file to suit the memory layout of your target system, see the chapter *Placing code and data*.

CONTROLLING COMPILER OPTIMIZATIONS

The AVR IAR C/C++ Compiler allows you to specify whether generated code should be optimized for size or for speed, at a selectable optimization level. The purpose of optimization is to reduce the code size and to improve the execution speed. When only one of these two goals can be reached, the compiler prioritizes according to the settings you specify. Note that one optimization sometimes enables other optimizations to be performed, and an application may become smaller even when optimizing for speed rather than size.

The following table describes the optimization levels:

Optimization level	Description
None (Best debug support)	Variables live through their entire scope
Low	Dead code elimination Redundant label elimination Redundant branch elimination
Medium	Live-dead analysis and optimization Code hoisting Register content analysis and optimization Code motion Common subexpression elimination Clustering of variables
High (Maximum optimization)	Peephole optimization Cross jumping Cross call (when optimizing for size) Function inlining Type-based alias analysis

Table 30: Compiler optimization levels

By default, the same optimization level for an entire project or file is used, but you should consider using different optimization settings for different files in a project. For example, put code that must execute very quickly into a separate file and compile it for minimal execution time (maximum speed), and the rest of the code for minimal code size. This will give a small program, which is still fast enough where it matters. The `#pragma optimize` directive allows you to fine-tune the optimization for specific functions, such as time-critical functions.

A high level of optimization will result in increased compile time, and may also make debugging more difficult, since it will be less clear how the generated code relates to the source code. At any time, if you experience difficulties when debugging your code, try lowering the optimization level.

Both compiler options and pragma directives are available for specifying the preferred type and level of optimization. The chapter *Compiler options* contains reference information about the command line options used for specifying optimization type and level. Refer to the *AVR® IAR Embedded Workbench™ IDE User Guide* for information about the compiler options available in IAR Embedded Workbench. Refer to *#pragma optimize*, page 221, for information about the pragma directives that can be used for specifying optimization type and level.

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 IAR Embedded Workbench IDE **Function inlining**, or the `#pragma optimize` directive. The following transformations can be disabled:

- Common subexpression elimination
- 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 **Low** and **None**.

To read more about the command line option, see `--no_cse`, page 190.

Function inlining

Function inlining means that a simple function, whose definition is known at compile time, is integrated into the body of its caller to eliminate the overhead of the call. This optimization, which is performed at optimization level **High**, normally reduces execution time, but increases code size. The resulting code might also be difficult to debug.

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

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

To read more about the command line option, see `--no_inline`, page 190.

Code motion

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

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

Type-based alias analysis

When two or more pointers reference the same memory location, these pointers are said to be *aliases* for each other. The existence of aliases makes optimization more difficult because it is not necessarily known at compile time whether a particular value is being changed.

Type-based alias analysis optimization assumes that all accesses to an object will take place using its declared type or as a `char` type. This assumption lets the compiler detect whether pointers may reference the same memory location or not.

Type-based alias analysis is performed at optimization level **High**. For ISO/ANSI standard-conforming C or C++ application code, this optimization can reduce code size and execution time. However, non-standard-conforming C or C++ code might result in the compiler producing code that leads to unexpected behavior. Therefore, it is possible to turn this optimization off.

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

To read more about the command line option, see `--no_tbaa`, page 191.

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. By using explicit casts, you can also force pointers of different pointer types to point to the same memory location.

Clustering of variables

When clustering of variables is enabled, static and global variables 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**.

Cross call

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

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

To read more about related command line options, see `--no_cross_call`, page 189, `--do_cross_call`, page 179, and `--cross_call_passes`, page 174.

Selecting data types and placing data in memory

For efficient treatment of data, you should consider the data types used and the most efficient placement of the data. This section provides useful information for efficient treatment of data:

- Locating strings in ROM, RAM and flash
- Using efficient data types
- Memory model and memory attributes for data
- Using the best pointer type
- Anonymous structs and unions
- Saving stack space and RAM memory.

LOCATING STRINGS IN ROM, RAM AND FLASH

With the AVR IAR C/C++ Compiler 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 39.

Putting 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 put 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 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 249.

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 to const data tells the calling function that the data pointed to will not change, which opens for better optimizations.

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

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. The AVR IAR C/C++ 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 AVR IAR C/C++ Compiler, the floating-point type `float` always uses the 32-bit format. The format used by the `double` floating-point type depends on the compiler option `--64bit_doubles` (**Use 64-bit doubles**).

Unless the application requires the extra precision that 64-bit floating-point numbers give, we recommend using 32-bit floats instead. Also consider replacing code using floating-point operations with code using integers because these are more efficient.

Note that 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`, `1` is added and the result is converted back to a `float`:

```
float test(float a)
{
    return a+1.0;
}
```

To treat a floating-point constant as a `float` rather than as a `double`, add an `f` to it, for example:

```
float test(float a)
{
    return a+1.0f;
}
```

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 AVR IAR C/C++ Compiler 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 *Extended keywords for data*, page 16.

Efficient usage of memory type attributes can significantly reduce the application size.

For details about the memory types, see *Memory types and memory attributes*, page 18.

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 performing 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 141.

ANONYMOUS STRUCTS AND UNIONS

When declaring a structure or union 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 AVR IAR C/C++ Compiler they can be used in C if language extensions are enabled.



In IAR Embedded Workbench, language extensions are enabled by default.



Use the `-e` compiler option to enable language extensions. See `-e`, page 179, for additional information.

Example

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

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

void f(void)
{
    st.l = 5;
}
```

The member names must be unique in the surrounding scope. Having an anonymous struct or union at file scope, as a global, external, or static variable is also allowed. This could for instance be used for declaring I/O registers, as in the following example:

```
__no_init volatile
union
{
    unsigned char IOPORT;
    struct
    {
        unsigned char way: 1;
        unsigned char out: 1;
    };
} @ 0x1234;
```

This declares an I/O register byte `IOPORT` at the address `0x1234`. The I/O register has 2 bits declared, `way` and `out`. Note that both the inner structure and the outer union are anonymous.

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

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

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

Writing efficient code

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

The following is a list of programming techniques that will, when followed, enable the compiler to better optimize the application.

- Local variables—auto variables and parameters—are preferred over static or global variables. The reason is that the optimizer must assume, for example, that called functions may modify non-local variables. When the life spans for local variables end, the previously occupied memory can then be reused. Globally declared variables will occupy data memory during the whole program execution.
- Avoid taking the address of local variables using the `&` operator. There are two main reasons why this is inefficient. First, the variable must be placed in memory, and thus cannot be placed in a processor register. This results in larger and slower code. Second, the optimizer can no longer assume that the local variable is unaffected over function calls.
- Module-local variables—variables that are declared static—are preferred over global variables. Also avoid taking the address of frequently accessed static variables.
- The compiler is capable of inlining functions. This means that instead of calling a function, the compiler inserts the content of the function at the location where the function was called. The result is a faster, but often larger, application. Also, inlining may enable further optimizations. The compiler often inlines small functions declared static. The use of the `#pragma inline` directive and the C++ keyword `inline` gives you fine-grained control, and it is the preferred method compared to the traditional way of using preprocessor macros. This feature can be disabled using the `--no_inline` command line option; see `--no_inline`, page 190.
- Avoid using inline assembler. Instead, try writing the code in C or C++, use intrinsic functions, or write a separate module in assembler language. For more details, see *Mixing C and assembler*, page 93.

SAVING STACK SPACE AND RAM MEMORY

The following is a list of programming techniques that will, when followed, save memory and stack space:

- If stack space is limited, avoid long call chains and recursive functions.
- Avoid using large non-scalar types, such as structures, as parameters or return type; in order to save stack space, you should instead pass them as pointers 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. In addition, using the prototyped style will make it possible to generate more efficient code, since type promotion (implicit casting) is not needed. The K&R style is only supported for compatibility reasons.

To make the compiler verify that all functions have proper prototypes, use the compiler option **Require prototypes** (`--require_prototypes`).

Prototyped style

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

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

Kernighan & Ritchie style

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

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

INTEGER TYPES AND BIT NEGATION

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

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

```
void f1(unsigned char c1)
{
    if (c1 == ~0x80)
        ;
}
```

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

PROTECTING SIMULTANEOUSLY ACCESSED VARIABLES

Variables that are accessed from multiple threads, for example from `main` or an interrupt, 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.

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



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 a number of AVR derivatives are included in the AVR IAR C/C++ Compiler delivery. The header files are named *ioderivative.h* and define the processor-specific special function registers (SFRs).

In IAR Embedded Workbench, enable the bit definitions by selecting the option **General Options>System>Enable bit definitions in I/O include files**.

SFRs with bitfields are declared in the header file. The following 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 to 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.PORTE_Bit0 = 1;
```

You can also use the header files as templates when you create new header files for other AVR derivatives. For details about the `@` operator, see *Located data*, page 46.

NON-INITIALIZED VARIABLES

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

The compiler supports the declaration of variables that will not be initialized, using the `__no_init` type modifier. They can be specified either as a keyword or using the `#pragma object_attribute` directive. The compiler places such variables in separate segments, 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.

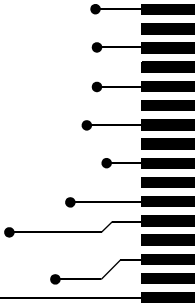
Variables declared using the `__no_init` keyword could, for example, be large input buffers or mapped to special RAM that keeps its content even when the application is turned off.

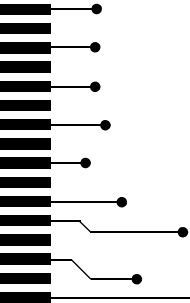
For information about the `__no_init` keyword, see page 211. Note that to use this keyword, language extensions must be enabled; see *-e*, page 179. For information about the `#pragma object_attribute`, see page 220.

Part 2. Compiler reference

This part of the AVR® IAR C/C++ Compiler Reference Guide contains the following chapters:

- Data representation
- Segment reference
- Compiler options
- Extended keywords
- Pragma directives
- The preprocessor
- Intrinsic functions
- Library functions
- Implementation-defined behavior
- IAR language extensions
- Diagnostics.





Data representation

This chapter describes the data types, pointers, and structure types supported by the AVR IAR C/C++ 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, four, it must be stored on an address that is divisible by four.

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 inherit the alignment from its components.

All objects must have a size that is a multiple of the alignment. Otherwise, only the first element of an array would be placed in accordance with the alignment requirements.

In the following example, the alignment of the structure is 4, under the assumption that `long` has alignment 4. Its size is 8, even though only 5 bytes are effectively used.

```
struct str {
    long a;
    char b;
};
```

In standard C, the size of an object can be determined by using the `sizeof` operator.

ALIGNMENT IN THE AVR IAR C/C++ COMPILER

The AVR microcontroller does not have any alignment restrictions.

Basic data types

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

INTEGER TYPES

The following table gives the size and range of each integer data type:

Data type	Size	Range	Alignment
<code>bool</code>	8 bits	0 to 1	1
<code>char</code>	8 bits	0 to 255	1
<code>signed char</code>	8 bits	-128 to 127	1
<code>unsigned char</code>	8 bits	0 to 255	1
<code>signed short</code>	16 bits	-32768 to 32767	1
<code>unsigned short</code>	16 bits	0 to 65535	1
<code>signed int</code>	16 bits	-32768 to 32767	1
<code>unsigned int</code>	16 bits	0 to 65535	1
<code>signed long</code>	32 bits	-2^{31} to $2^{31}-1$	1
<code>unsigned long</code>	32 bits	0 to $2^{32}-1$	1
<code>signed long long</code>	64 bits	-2^{63} to $2^{63}-1$	1
<code>unsigned long long</code>	64 bits	0 to $2^{64}-1$	1

Table 31: Integer types

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

Bool

The `bool` data type is supported by default in the C++ language. If you have enabled language extensions, the `bool` type can also be used in C source code if you include the file `stdbool.h`. This will also enable the boolean values `false` and `true`.

The enum type

ISO/ANSI C specifies that constants defined using the `enum` construction should be representable using the type `int`. The compiler will use the shortest signed or unsigned type required to contain the values.

When IAR Systems language extensions are enabled, and in C++, the `enum` constants and types can also be of the type `long` or `unsigned long`.

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

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

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

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

FLOATING-POINT TYPES

In the AVR IAR C/C++ Compiler, floating-point values are represented in standard IEEE format.

The ranges and sizes for the different floating-point types are:

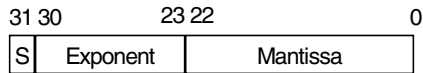
Type	Size	Range (+/-)	Exponent	Mantissa
<code>float</code>	32 bits	$\pm 1.18\text{E-}38$ to $\pm 3.39\text{E}+38$	8 bits	23 bits
<code>double</code> *	32 bits (default)	$\pm 1.18\text{E-}38$ to $\pm 3.39\text{E}+38$	8 bits	23 bits
<code>double</code> *	64 bits	$\pm 2.23\text{E-}308$ to $\pm 1.79\text{E}+308$	11 bits	52 bits
<code>long double</code> *	32 bits	$\pm 1.18\text{E-}38$ to $\pm 3.39\text{E}+38$	8 bits	23 bits
<code>long double</code> *	64 bits	$\pm 2.23\text{E-}308$ to $\pm 1.79\text{E}+308$	11 bits	52 bits

Table 32: Floating-point types

* Depends on whether the `--64bit_doubles` option is used, see `--64bit_doubles`, page 201. The type `long double` use the same precision as `double`.

32-bit floating-point format

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



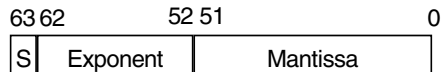
The value of the number is:

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

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 value of the number is:

$$(-1)^S * 2^{(\text{Exponent}-1023)} * 1.\text{Mantissa}$$

The precision of the float operators (+, -, *, and /) is approximately 15 decimal digits.

Special cases

The following applies to both 32-bit and 64-bit floating-point formats:

- Zero is represented by zero mantissa and exponent. The sign bit signifies positive or negative zero.
- Infinity is represented by setting the exponent to the highest value and the mantissa to zero. The sign bit signifies positive or negative infinity.
- Not a number (NaN) is represented by setting the exponent to the highest positive value and the mantissa to a non-zero value. The value of the sign bit is ignored.
- Subnormal numbers are used for representing values smaller than what can be represented by normal values. The drawback is that the precision will decrease with smaller values. The exponent is set to 0 to signify that the number is denormalized, even though the number is treated as if the exponent would have been 1. Unlike normal numbers, denormalized numbers do not have an implicit 1 as the most significant bit (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.

Pointer types

The AVR IAR C/C++ 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 code pointer is the actual address it refers to divided by two.

The following function pointers are available:

Keyword	Address range	Pointer size	Index type	Description
<code>__nearfunc</code>	0-0x1FFFE	2 bytes	signed int	Can be called from any part of the code memory, but must reside in the first 128 Kbytes of that space.
<code>__farfunc</code>	0-0x7FFFE	3 bytes	signed long	No restrictions on code placement.

Table 33: Function pointers

DATA POINTERS

Data pointers have three sizes: 8, 16, or 24 bits. The following data pointers are available:

Keyword	Pointer size	Memory space	Index type	Range
<code>__tiny</code>	1 byte	Data	signed char	0x0-0xFF
<code>__near</code>	2 bytes	Data	signed int	0x0-0xFFFF
<code>__far</code>	3 bytes	Data	signed int	0x0-0xFFFFFFFF (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

Table 34: Data pointers

Keyword	Pointer size	Memory space	Index type	Range
<code>__farflash</code>	3 bytes	Code	signed int	0x0-0xFFFFFFFF (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 3 bytes	Data/Code	signed int 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 34: Data pointers

CASTING

Casts between pointers have the following characteristics:

- Casting a *value* of an integer type to a pointer of a smaller type is performed by truncation
- Casting a *value* of an integer type to a pointer of a larger type is performed by zero extension
- Casting a *pointer type* to a smaller integer type is performed by truncation
- Casting a *pointer type* to a larger integer type is performed 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.

size_t

`size_t` is the unsigned integer type required to hold the maximum size of an object. The following table shows the typedef of `size_t` depending on the processor option:

Generic processor option	Typedef
<code>-v0</code> and <code>-v1</code>	unsigned int
<code>-v2</code> , <code>-v3</code> , <code>-v4</code> , <code>-v5</code> , and <code>-v6</code>	unsigned long

Table 35: `size_t` typedef

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. The following 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 36: `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 AVR IAR C/C++ Compiler, the size of `intptr_t` is `long int` when using the Large 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` (fields) are always allocated in the order given in the declaration. The members are placed in memory according to the given alignment (offsets), which always is 1.

Example

```

struct {
    short s; /* stored in byte 0 and 1 */
    char c; /* stored in byte 2 */
    long l; /* stored in byte 4, 5, and 6 */
    char c2; /* stored in byte 7 */
} s;

```

The following diagram shows the layout in memory:

s.s 2 bytes	s.c 1 byte	s.l 4 bytes	s.c2 1 byte
----------------	---------------	----------------	----------------

The alignment of the structure is 1 byte, and its size is 8 bytes.

Type and object attributes

The AVR IAR C/C++ Compiler provides a set of attributes that support specific features of the AVR microcontroller. There are two basic types of attributes—*type attributes* and *object attributes*.

Type attributes affect the *external functionality* of the data object or function. For instance, how an object is placed in memory, or in other words, how it is accessed.

Object attributes affect the *internal functionality* of the data object or function.

To understand the syntax rules for the different attributes, it is important to be familiar with the concepts of the type attributes and the object attributes.

For information about how to use attributes to modify data, see the chapter *Data storage*. For information about how to use attributes to modify functions, see the chapter *Functions*. For detailed information about each attribute, see *Descriptions of extended keywords*, page 204.

TYPE ATTRIBUTES

Type attributes define how a function is called, or how a data object is accessed. This means that type attributes must be specified both when they are defined and in the declaration.

You can either place the type attributes directly in your source code, or use the pragma directive `#pragma type_attribute`.

Type attributes can be further divided into *memory attributes* and *general type attributes*.

Memory attributes

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

- Available *memory attributes for functions*: `__nearfunc`, and `__farfunc`
- Available *memory attributes for data objects*: `__tiny`, `__near`, `__far`, `__huge`, `__regvar`, `__eeprom`, `__tinyflash`, `__flash`, `__farflash`, `__hugeflash`, `__generic`, `__io`, and `__ext_io`

For each level of indirection, you can only specify one memory attribute.

General type attributes

The following general type attributes are available:

- *Function type attributes* change the calling convention of a function: `__interrupt`, `__task`, and `__version_1`
- *Data type attributes*: `const`, and `volatile`

For each level of indirection, you can specify as many type attributes as required.

To read more about volatile, see *Declaring objects volatile*, page 146.

OBJECT ATTRIBUTES

Object attributes affect functions and data objects, but not how the function is called or how the data is accessed. This means that an object attribute does not need to be present in the declaration of an object.

The following object attributes are available:

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

Note: The `__intrinsic` attribute is reserved for compiler internal use only.

You can specify as many object attributes as required.

DECLARING OBJECTS IN SOURCE FILES

When declaring objects, note that the IAR-specific attributes work exactly like `const`. One exception to this is that attributes that are declared in front of the type specifier apply to all declared objects.

See *More examples*, page 21.

DECLARING OBJECTS VOLATILE

There are three main reasons for declaring an object `volatile`:

- Shared access; the object is shared between several tasks in a multitasking environment
- Trigger access; as for a memory-mapped SFR where the fact that an access occurs has an effect
- Modified access; where the contents of the object can change in ways not known to the compiler.

Definition of access to volatile objects

The ISO/ANSI standard defines an abstract machine, which governs the behavior of accesses to `volatile` declared objects. The AVR IAR C/C++ Compiler considers each read and write access to an object that has been declared `volatile` as an access. The unit for the access is either the entire object or, for accesses to an element in a composite object—such as an array, struct, class, or union—the element. For example:

```
char volatile a;
a = 5; /* A write access */
a += 6; /* First a read then a write access */
```

An access to a bitfield is treated as an access to the underlying type.

Rules for accesses

Accesses to `volatile` declared objects are subject to the following rules:

- 1 All accesses are preserved
- 2 All accesses are complete, that is, the whole object is accessed
- 3 All accesses are performed in the same order as given in the abstract machine
- 4 All accesses are atomic, that is, non-interruptable.

The AVR IAR C/C++ Compiler adheres to these rules for all 8-bit types.

The following object types are treated in a special way:

Type of object	Treated as
Global register variables	Treated as a non-triggering volatile
<code>__io</code> declared variables located in <code>0x-0x1F</code>	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 37: Volatile objects with special handling

For all combinations of object types not listed, only rule number one applies.

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 legal to write assembler code that accesses class members.

Segment reference

The AVR IAR C/C++ 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 information about how to define segments in the linker command file, see *Customizing the linker command file*, page 35.

Summary of segments

The table below lists the segments that are available in the AVR IAR C/C++ Compiler. Note that *located* denotes absolute location using the @ operator or the #pragma location directive. The XLINK segment memory type CODE, DATA, or XDATA indicates whether the segment should be placed in ROM or RAM memory areas; see Table 9, *XLINK segment memory types*, page 34.

Segment	Description	Type
CODE	Holds program code declared <code>__nearfunc</code> .	CODE
CSTACK	Holds the stack used by C or C++ programs.	DATA
DIFUNCT	Holds pointers to constructor blocks that should be executed by the system startup code before <code>main</code> is called.	CODE
EEPROM_AN	Holds initialized located <code>__eeprom</code> variables.	XDATA
EEPROM_I	Holds non-zero initialized static and global <code>__eeprom</code> variables.	XDATA
EEPROM_N	Holds <code>__no_init</code> static and global <code>__eeprom</code> variables.	XDATA
FARCODE	Holds program code declared <code>__farfunc</code> .	CODE
FAR_C	Holds <code>__far</code> declared constant data, including literal strings.	DATA
FAR_F	Holds static and global <code>__farflash</code> variables.	CODE
FAR_HEAP	Holds the heap used for dynamically allocated data in far memory using the DLIB library.	DATA
FAR_I	Holds non-zero initialized static and global <code>__far</code> variables.	DATA
FAR_ID	Holds initial values for the variables located in the <code>FAR_I</code> segment.	CODE
FAR_N	Holds <code>__no_init</code> static and global <code>__far</code> variables.	DATA
FAR_Z	Holds zero-initialized static and global <code>__far</code> variables.	DATA

Table 38: Segment summary

Segment	Description	Type
HEAP	Holds the heap used for dynamically allocated data using the CLIB library.	DATA
HUGE_C	Holds <code>__huge</code> declared constant data, including literal strings.	DATA
HUGE_F	Holds static and global <code>__hugeflash</code> variables.	CODE
HUGE_HEAP	Holds the heap used for dynamically allocated data in huge memory using the DLIB library.	DATA
HUGE_I	Holds non-zero initialized static and global <code>__huge</code> variables.	DATA
HUGE_ID	Holds initial values for the variables located in the HUGE_I segment.	CODE
HUGE_N	Holds <code>__no_init</code> static and global <code>__huge</code> variables.	DATA
HUGE_Z	Holds zero-initialized static and global <code>__huge</code> variables.	DATA
INITTAB	Contains compiler-generated table entries that describe the segment initialization that will be performed at system start up.	CODE
INTVEC	Contains the reset and interrupt vectors.	CODE
NEAR_C	Used for storing <code>__tiny</code> and <code>__near</code> constant data, including literal strings.	DATA
NEAR_F	Holds static and global <code>__flash</code> variables.	CODE
NEAR_HEAP	Holds the heap used for dynamically allocated data in near memory using the DLIB library.	DATA
NEAR_I	Holds non-zero initialized static and global <code>__near</code> variables.	DATA
NEAR_ID	Holds initial values for the variables located in the NEAR_I segment.	CODE
NEAR_N	Holds <code>__no_init</code> static and global <code>__near</code> variables.	DATA
NEAR_Z	Holds zero-initialized static and global <code>__near</code> variables.	DATA
RSTACK	Holds the internal return stack.	DATA
SWITCH	Holds switch tables for all functions.	CODE
TINY_F	Holds static and global <code>__tinyflash</code> variables.	CODE
TINY_HEAP	Holds the heap used for dynamically allocated data in tiny memory using the DLIB library.	DATA
TINY_I	Holds non-zero initialized static and global <code>__tiny</code> variables.	DATA
TINY_ID	Holds initial values for the variables located in the TINY_I segment.	CODE
TINY_N	Holds <code>__no_init</code> static and global <code>__tiny</code> variables.	DATA
TINY_Z	Holds zero-initialized static and global <code>__tiny</code> variables.	DATA

Table 38: Segment summary (Continued)

Descriptions of segments

The following section gives reference information about each segment. For detailed information about the extended keywords mentioned here, see the chapter *Extended keywords*.

CODE Holds `__nearfunc` program code.

XLINK segment memory type

CODE

Memory space

Flash. The address range is `0x0-0x01FFFE`.

Description

By default, functions are placed in the CODE segment when using any of the processor options `-v0`, `-v1`, `-v2`, `-v3`, or `-v4`. A function is also placed in the CODE segment if you are using any of the processor options `-v5` or `-v6` and explicitly declare the function `__nearfunc`.

CSTACK Holds the internal data stack.

XLINK segment memory type

DATA

Memory space

Data. The address range depends on the memory model:

Memory model	Address range	Comment
Tiny	<code>0x0-0xFF</code>	
Small	<code>0x0-0xFFFF</code>	
Large	<code>0x0-0xFFFFFFFF</code>	Maximum 64 Kbytes stack. CSTACK must not cross a 64-Kbyte boundary.

Table 39: Memory models

Description

Holds the internal data stack. This segment and its length is normally defined in the linker command file with the following command:

```
-Z (DATA) CSTACK+nn=start
```

or

```
-Z (DATA) CSTACK=start-end
```

where *nn* is the length, *start* is the first memory location, and *end* is the last memory location.

DIFUNCT Holds pointers to constructor blocks in C++ code.

XLINK segment memory type

CODE

Memory space

Flash. The address range is 0x0-0xFFFF.

Description

When using C++ and global objects, it is necessary to call the constructor methods of these global objects before `main` is called.

EEPROM_AN Variables with an absolute location in the built-in EEPROM.

XLINK segment memory type

XDATA

Memory space

EEPROM.

Description

This segment is not copied to EEPROM during system startup. Instead it is used for programming the EEPROM during the download of the code.

Use the command line option `--eeprom_size` to set the address range for this segment; see `--eeprom_size`, page 180, for additional information.

EEPROM_I Holds non-zero initialized static and global `__eeprom` variables.

XLINK segment memory type

XDATA

Memory space

EEPROM.

Description

Holds static and global `__eeprom` variables that have been defined with non-zero initial values. This segment is not copied to EEPROM during system startup. Instead it is used for programming the EEPROM during the download of the code.

Use the command line option `--eeprom_size` to set the address range for this segment; see *--eeprom_size*, page 180, for additional information.

EEPROM_N Holds `__no_init` static and global `__eeprom` variables.

XLINK segment memory type

XDATA

Memory space

EEPROM.

Description

Holds static and global `__eeprom` variables that will not be initialized during the download of the code to the EEPROM. These variables have been declared `__no_init`.

Use the command line option `--eeprom_size` to set the address range for this segment; see *--eeprom_size*, page 180, for additional information.

FARCODE Holds `__farfunc` program code.

XLINK segment memory type

CODE

Memory space

Flash. The address range is `0x0-0x7FFFFE`.

Description

Holds user program code that has been declared `__farfunc`. The `__farfunc` memory attribute is available when using the `-v5` and `-v6` options, in which case the `__farfunc` is implicitly used for all functions.

`FAR_C` Holds `__far` constant data, including string literals.

XLINK segment memory type

CONST

Memory space

Data. The address range is `0x0-0xFFFFF`.

Description

Holds `__far` constant data, including string literals.

Note: This segment is located in external ROM. Systems without external ROM may not use this segment.

When the `-y` compiler option is used, `__far` constant data is located in the `FAR_I` segment.

`FAR_F` Holds static and global `__farflash` variables.

XLINK segment memory type

CODE

Memory space

Flash. The address range is `0x0-0x7FFFFF`.

Description

Holds static and global `__farflash` variables and aggregate initializers.

`FAR_HEAP` Holds the heap used for dynamically allocated data in far memory when using the DLIB library.

XLINK segment memory type

DATA

Memory space

Data. The address range is 0x0-0xFFFFFFFF.

Description

This segment holds dynamically allocated data in far memory, in other words data used by `far_malloc` and `far_free`, and in C++, `new` and `delete`.

This segment and its length is normally defined in the linker command file by the command:

```
-Z (DATA) FAR_HEAP+nn=start
```

where `nn` is the length and `start` is the location.

For more information about dynamically allocated data and the heap, see *The return address stack*, page 43. For information about using the `new` and `delete` operators for a heap in far memory, see *New and Delete operators*, page 114.

`FAR_I` Holds non-zero initialized static and global `__far` variables.

XLINK segment memory type

DATA

Memory space

Data. The address range is 0x0-0xFFFFFFFF.

Description

Holds static and global `__far` variables that have been defined with non-zero initial values.

When the `-y` compiler option is used, `__far` constant data is located in this segment.

FAR_ID Holds `__far` variable initializers.

XLINK segment memory type

CODE

Memory space

Flash. The address range is 0x0-0x7FFFFFFF.

Description

Holds initial values for the variables located in the FAR_I segment. These values are copied from FAR_ID to FAR_I during system initialization.

FAR_N Holds `__no_init` static and global `__far` variables.

XLINK segment memory type

DATA

Memory space

Data. The address range is 0x0-0xFFFFFFFF.

Description

Holds static and global `__far` variables that will not be initialized at system startup, for example variables that are to be placed in non-volatile memory. These variables have been declared `__no_init`.

FAR_Z Holds zero-initialized static and global `__far` variables.

XLINK segment memory type

DATA

Memory space

Flash. The address range is 0x0-0xFFFFFFFF.

Description

Holds static and global `__far` variables that have been declared without initial values or with zero initial values.

HEAP Used for the heap when using the CLIB library.

XLINK segment memory type

DATA

Memory space

Data. The address range depends on the memory model:

Memory model	Address range
Tiny	0x0–0xFF
Small	0x0–0xFFFF
Large	0x0–0xFFFFFFFF

Table 40: Heap memory address range

Description

Holds the heap used for dynamically allocated data.

This segment and its length is normally defined in the linker command file by the command:

```
-Z (DATA) HEAP+nn=start
```

where *nn* is the length and *start* is the location.

HUGE_C Holds `__huge` constant data, including string literals.

XLINK segment memory type

CONST

Memory space

Data. The address range is 0x0–0xFFFFFFFF.

Description

Holds `__huge` constant data, including string literals.

Note: This segment is located in external ROM. Systems without external ROM may not use this segment.

HUGE_F Holds static and global `__hugeflash` variables.

XLINK segment memory type

CODE

Memory space

Flash. The address range is `0x0-0xFFFFFFFF`.

Description

Holds static and global `__hugeflash` variables and aggregate initializers.

HUGE_HEAP Holds the heap used for dynamically allocated data in huge memory when using the DLIB library.

XLINK segment memory type

DATA

Memory space

Data. The address range is `0x0-0xFFFFFFFF`.

Description

This segment holds dynamically allocated data in huge memory, in other words data used by `huge_malloc` and `huge_free`, and, in C++, `new` and `delete`.

This segment and its length is normally defined in the linker command file by the command:

```
-Z (DATA) HUGE_HEAP+nn=start
```

where *nn* is the length and *start* is the location.

For more information about dynamically allocated data and the heap, see *The return address stack*, page 43. For information about using the `new` and `delete` operators for a heap in huge memory, see *New and Delete operators*, page 114.

HUGE_I Holds non-zero initialized static and global `__huge` variables.

XLINK segment memory type

DATA

Memory space

Data. The address range is 0x0-0xFFFFFFFF.

Description

Holds static and global `__huge` variables that have been defined with non-zero initial values.

When the `-y` compiler option is used, `__huge` constant data is located in this segment.

HUGE_ID Holds `__huge` variable initializers.

XLINK segment memory type

CODE

Memory space

Flash. The address range is 0x0-0x7FFFFFFF.

Description

Holds initial values for the variables located in the `HUGE_I` segment. These values are copied from `HUGE_ID` to `HUGE_I` during system initialization.

HUGE_N Holds `__no_init` static and global `__huge` variables.

XLINK segment memory type

DATA

Memory space

Data. The address range is 0x0-0xFFFFFFFF.

Description

Holds static and global `__huge` variables that will not be initialized at system startup, for example variables that are to be placed in non-volatile memory. Variables defined using the `__no_init` keyword will be placed in this segment.

HUGE_Z Holds zero-initialized static and global `__huge` variables.

XLINK segment memory type

DATA

Memory space

Data. The address range is `0x0-0xFFFFFFFF`.

Description

Holds static and global `__huge` variables that have been declared without initial values or with zero initial values.

INITTAB Segment initialization descriptions.

XLINK segment memory type

CODE

Memory space

Flash. The address range is `0x0-0xFFFF` (`0x7FFFFFFF` if farflash is enabled).

Description

Contains compiler-generated table entries that describe the segment initialization which will be performed at system startup.

INTVEC Holds the interrupt vector table.

XLINK segment memory type

CODE

Memory space

Flash. The address range is approximately `0-64`.

Description

Holds the interrupt vector table generated by the use of the `__interrupt` extended keyword.

Note: This segment *must* be placed at address `0` and forwards.

NEAR_C Holds `__tiny` and `__near` constant data, including string literals.

XLINK segment memory type

CONST

Memory space

Data. The address range is `0x0-0xFFFF`.

Description

Holds `__tiny` and `__near` constant data, including string literals.

Note: This segment is located in external ROM. Systems without external ROM may not use this segment.

NEAR_F Holds static and global `__flash` variables.

XLINK segment memory type

CODE

Memory space

Flash. The address range is `0x0-0xFFFF`.

Description

Holds static and global `__flash` variables and aggregate initializers.

NEAR_HEAP Holds the heap used for dynamically allocated data in near memory when using the DLIB library.

XLINK segment memory type

DATA

Memory space

Data. The address range is `0x0-0xFFFF`.

Description

This segment holds dynamically allocated data in near memory, in other words data used by `near_malloc` and `near_free`, and in C++, `new` and `delete`.

This segment and its length is normally defined in the linker command file by the command:

```
-Z (DATA) NEAR_HEAP+nn=start
```

where *nn* is the length and *start* is the location.

For more information about dynamically allocated data and the heap, see *The return address stack*, page 43. For information about using the `new` and `delete` operators for a heap in near memory, see *New and Delete operators*, page 114.

NEAR_I Holds non-zero initialized static and global `__near` variables.

XLINK segment memory type

DATA

Memory space

Data. The address range is 0x0-0xFFFF.

Description

Holds static and global `__near` variables that have been defined with non-zero initial values.

When the `-y` compiler option is used, NEAR_C data (`__near` or `__tiny`) is located in this segment.

NEAR_ID Holds `__near` variable initializers.

XLINK segment memory type

CODE

Memory space

Flash. The address range is 0x0-0x7FFFFFFF.

Description

Holds initial values for the variables located in the NEAR_I segment. These values are copied from NEAR_ID to NEAR_I during system initialization.

NEAR_N Holds `__no_init` static and global `__near` variables.

XLINK segment memory type

DATA

Memory space

Data. The address range is `0x0-0xFFFF`.

Description

Holds static and global `__near` variables that will not be initialized at system startup, for example variables that are to be placed in non-volatile memory. These variables have been declared `__no_init`.

NEAR_Z Holds zero-initialized static and global `__near` variables.

XLINK segment memory type

DATA

Memory space

Data. The address range is `0x0-0xFFFF`.

Description

Holds static and global `__near` variables that have been declared without initial values or with zero initial values.

RSTACK Holds the internal return stack.

XLINK segment memory type

DATA

Memory space

Data. The address range is `0x0-0xFFFF`.

Description

Holds the internal return stack.

SWITCH Holds switch tables for all functions.

XLINK segment memory type

CODE

Memory space

Flash. The address range is 0x0-0xFFFF. If the `__farflash` extended keyword and the `--enhanced_core` option are used, the range is 0x0-0x7FFFFFFF. This segment must not cross a 64-Kbyte boundary.

Description

The `SWITCH` segment is for compiler internal use only and should always be defined. The segment allocates, if necessary, jump tables for C switch statements.

TINY_F Holds static and global `__tinyclash` variables.

XLINK segment memory type

CODE

Memory space

Flash. The address range is 0x0-0xFF.

Description

Holds static and global `__tinyclash` variables and aggregate initializers.

TINY_HEAP Holds the heap used for dynamically allocated data in tiny memory when using the DLIB library.

XLINK segment memory type

DATA

Memory space

Data. The address range is 0x0-0xFF.

Description

This segment holds dynamically allocated data in tiny memory, in other words data used by `tiny_malloc` and `tiny_free`, and in C++, `new` and `delete`.

This segment and its length is normally defined in the linker command file by the command:

```
-Z (DATA) TINY_HEAP+nn=start
```

where *nn* is the length and *start* is the location.

For more information about dynamically allocated data and the heap, see *The return address stack*, page 43. For information about using the `new` and `delete` operators for a heap in tiny memory, see *New and Delete operators*, page 114.

TINY_I Holds non-zero initialized static and global `__tiny` variables.

XLINK segment memory type

DATA

Memory space

Data. The address range is 0x0-0xFF.

Description

Holds static and global `__tiny` variables that have been defined with non-zero initial values.

When the `-y` compiler option is used, `FAR_C` data is located in this segment.

TINY_ID Holds `__tiny` variable initializers.

XLINK segment memory type

CODE

Memory space

Flash. The address range is 0x0-0x7FFFFFFF.

Description

Holds initial values for the variables located in the TINY_I segment. These values are copied from TINY_ID to TINY_I during system initialization.

TINY_N Holds `__no_init` static and global `__tiny` variables.

XLINK segment memory type

DATA

Memory space

Data. The address range is `0x0-0xFF`.

Description

Holds static and global `__tiny` variables that will not be initialized at system startup, for example variables that are to be placed in non-volatile memory. These variables have been declared `__no_init`.

TINY_Z Holds zero-initialized static and global `__tiny` variables.

XLINK segment memory type

DATA

Memory space

Data. The address range is `0x0-0xFF`.

Description

Holds static and global `__tiny` variables that have been declared without initial values or with zero initial values.

Compiler options

This chapter explains how to set the compiler options from the command line, and gives detailed reference information about each option.



Refer to the *AVR® IAR Embedded Workbench™ IDE User Guide* for information about the compiler options available in IAR Embedded Workbench and how to set them.

Setting command line options

To set compiler options from the command line, include them on the command line after the `iccavr` command, either before or after the source filename. For example, when compiling the source `prog.c`, use the following command to generate an object file with debug information:

```
iccavr prog --debug
```

Some options accept a filename, included after the option letter with a separating space. For example, to generate a listing to the file `list.lst`:

```
iccavr prog -l list.lst
```

Some other options accept a string that is not a filename. The string is included after the option letter, but without a space. For example, to define a symbol:

```
iccavr prog -DDEBUG=1
```

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.

Note that a command line option has a *short* name and/or a *long* name:

- A short option name consists of one character, with or without parameters. You specify it with a single dash, for example `-e`
- A long name consists of one or several words joined by underscores, and it may have parameters. You specify it with double dashes, for example `--char_is_signed`.

SPECIFYING PARAMETERS

When a parameter is needed for an option with a short name, it can be specified either immediately following the option or as the next command line argument.

For instance, an include file path of `\usr\include` can be specified either as:

```
-I\usr\include
```

or as:

```
-I \usr\include
```

Note: `/` can be used instead of `\` as the directory delimiter.

Additionally, output file options can take a parameter that is a directory name. The output file will then receive a default name and extension.

When a parameter is needed for an option with a long name, it can be specified either immediately after the equal sign (=) or as the next command line argument, for example:

```
--diag_suppress=Pe0001
```

or

```
--diag_suppress Pe0001
```

The option `--preprocess`, however, is an exception, as the filename must be preceded by a space. In the following example, comments are included in the preprocessor output:

```
--preprocess=c prog
```

Options that accept multiple values may be repeated, and may also have comma-separated values (without a space), for example:

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

The current directory is specified with a period (`.`), for example:

```
iccavr prog -l .
```

A file parameter specified by `'-'` represents standard input or output, whichever is appropriate.

Note: When an option takes a parameter, the parameter cannot start with a dash (`-`) followed by another character. Instead, you can prefix the parameter with two dashes; the following example will create a list file called `-r`:

```
iccavr prog -l ---r
```

SPECIFYING ENVIRONMENT VARIABLES

Compiler options can also be specified in the `QCCAVR` environment variable. The compiler automatically appends the value of this variable to every command line, so it provides a convenient method of specifying options that are required for every compilation.

The following environment variables can be used with the AVR IAR C/C++ Compiler:

Environment variable	Description
<code>C_INCLUDE</code>	Specifies directories to search for include files; for example: <code>C_INCLUDE=c:\program files\iar systems\embedded workbench 4.n\avr\inc;c:\headers</code>
<code>QCCAVR</code>	Specifies command line options; for example: <code>QCCAVR=-lA asm.lst -z9</code>

Table 41: Environment variables

ERROR RETURN CODES

The AVR IAR C/C++ Compiler returns status information to the operating system which can be tested in a batch file.

The following command line error codes are supported:

Code	Description
0	Compilation successful, but there may have been warnings.
1	There were warnings, provided that the option <code>--warnings_affect_exit_code</code> was used.
2	There were non-fatal errors or fatal compilation errors making the compiler abort.
3	There were fatal errors.

Table 42: Error return codes

Options summary

The following table summarizes the compiler command line options:

Command line option	Description
<code>--char_is_signed</code>	Treats <code>char</code> as signed
<code>--cpu=cpu</code>	Specifies a processor variant
<code>--cross_call_passes=N</code>	Cross-call optimization

Table 43: Compiler options summary

Command line option	Description
<code>-Dsymbol [=value]</code>	Defines preprocessor symbols
<code>--debug</code>	Generates debug information
<code>--dependencies [= [i m]] {filename directory}</code>	Lists file dependencies
<code>--diag_error=tag, tag, ...</code>	Treats these as errors
<code>--diag_remark=tag, tag, ...</code>	Treats these as remarks
<code>--diag_suppress=tag, tag, ...</code>	Suppresses these diagnostics
<code>--diag_warning=tag, tag, ...</code>	Treats these as warnings
<code>--diagnostics_tables {filename directory}</code>	Lists all diagnostic messages
<code>--disable_direct_mode</code>	Disables direct addressing mode
<code>-dlib_config filename</code>	Determines the library configuration file
<code>--do_cross_call</code>	Forces cross-call optimization
<code>-e</code>	Enables language extensions
<code>--ec++</code>	Enables Embedded C++ syntax
<code>--eec++</code>	Enables Extended Embedded C++ syntax
<code>--eecr_address</code>	Defines the EECR address
<code>--enable_multibytes</code>	Enables support for multibyte characters
<code>--eeprom_size=N</code>	Specifies EEPROM size
<code>--enable_external_bus</code>	Adds code to enable external data bus
<code>--enhanced_core</code>	Enables enhanced instruction set
<code>--error_limit=n</code>	Specifies the allowed number of errors before compilation stops
<code>-f filename</code>	Extends the command line
<code>--force_switch_type [0 1 2]</code>	Forces the switch type
<code>--header_context</code>	Lists all referred source files

Table 43: Compiler options summary (Continued)

Command line option	Description
<code>-Ipath</code>	Specifies include file path
<code>--initializers_in_flash</code>	Places aggregate initializers in flash memory
<code>-l[a A b B c C D][N][H] {filename directory}</code>	Creates a list file
<code>--library_module</code>	Creates a library module
<code>--lock_regs N</code>	Locks registers
<code>-mname</code>	Memory model
<code>--memory_model=name</code>	Memory model
<code>--migration_preprocessor_extensions</code>	Extends the preprocessor
<code>--misrac</code>	Enables MISRA C-specific error messages
<code>--misrac_verbose</code>	Enables verbose logging of MISRA C checking
<code>--module_name=name</code>	Sets object module name
<code>--no_clustering</code>	Disables clustering of variables
<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_rampd</code>	Uses RAMPZ instead of RAMPD
<code>--no_ubrof_messages</code>	Minimizes object file size
<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_warnings</code>	Disables all warnings
<code>--no_wrap_diagnostics</code>	Disables wrapping of diagnostic messages
<code>-o {filename directory}</code>	Sets object filename

Table 43: Compiler options summary (Continued)

Command line option	Description
<code>--omit_types</code>	Excludes type information
<code>--only_stdout</code>	Uses standard output only
<code>--preinclude includefile</code>	Includes an include file before reading the source file
<code>--preprocess[=[c][n][l]] {filename directory}</code>	Generates preprocessor output
<code>--public_equ symbol[=value]</code>	Defines a global named assembler label
<code>-r</code>	Generates debug information
<code>--remarks</code>	Enables remarks
<code>--require_prototypes</code>	Verifies that prototypes are proper
<code>--root_variables</code>	Specifies variables as <code>__root</code>
<code>-s[2 3 6 9]</code>	Optimizes for speed
<code>--segment memory_attr=segment_name</code>	Changes segment name base
<code>--separate_cluster_for_initialized_variables</code>	Separates initialized and non-initialized variables
<code>--silent</code>	Sets silent operation
<code>--spmcr_address</code>	Defines the SPMCR address
<code>--strict_ansi</code>	Checks for strict compliance with ISO/ANSI C
<code>--string_literals_in_flash</code>	Puts "string" in the <code>__nearflash</code> or <code>__farflash</code> segment
<code>-v[0 1 2 3 4 5 6]</code>	Processor variant
<code>--version1_calls</code>	Uses ICCA90 calling convention
<code>--warnings_affect_exit_code</code>	Warnings affects exit code
<code>--warnings_are_errors</code>	Warnings are treated as errors
<code>-y</code>	Places constants and literals
<code>-z[2 3 6 9]</code>	Optimizes for size

Table 43: Compiler options summary (Continued)

Command line option	Description
<code>--zero_register</code>	Specifies register R15 as zero register
<code>--64bit_doubles</code>	Use 64-bit doubles
<code>--64k_flash</code>	Specifies a maximum of 64 Kbytes flash memory.

Table 43: Compiler options summary (Continued)

Descriptions of options

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

`--char_is_signed` `--char_is_signed`

By default, the compiler interprets the `char` type as unsigned. The `--char_is_signed` option causes the compiler to interpret the `char` type as signed instead. This can be useful when you, for example, want to maintain compatibility with another compiler.

Note: The runtime library is compiled without the `--char_is_signed` option. If you use this option, you may get type mismatch warnings from the IAR XLINK Linker, because the library uses unsigned chars.



To set the equivalent option in IAR Embedded Workbench, select **Project>Options>C/C++ Compiler>Language**.

`--cpu` `--cpu=cpu`

Use this option to select the processor for which the code is to be generated.

For example, use the following command to specify the AT90S4414 derivative:

```
--cpu=4414
```

See *Processor configuration*, page 5, for a summary of the available processor variants.

Note that to specify the processor, you can use either the `--cpu` option or the `-v` option. The `--cpu` option is, however, more precise because it contains more information about the intended target than the more generic `-v` option. For information about the implicit assumptions using the `-v` option, see `-v`, page 197.



This option is related to the **Processor configuration** option in the **General Options** category in IAR Embedded Workbench.

```
--cross_call_passes --cross_call_passes=N
```

Use this option to decrease the `RSTACK` usage by running the cross-call optimizer N times, where N can be 1–5. The default is to run it until no more improvements can be made.

For additional information, see `--no_cross_call`, page 189.

Note: Use this option if you have a target processor with a hardware stack or a small internal return stack segment, `RSTACK`.



This option is related to the **Optimizations** options in the **C/C++ Compiler** category in IAR Embedded Workbench.

```
-D -Dsymbol[=value]
-D symbol[=value]
```

Use this option to define a preprocessor symbol with the name `symbol` and the value `value`. If no value is specified, 1 is used.

The option `-D` has the same effect as a `#define` statement at the top of the source file:

```
-Dsymbol
```

is equivalent to:

```
#define symbol 1
```

In order to get the equivalence of:

```
#define FOO
```

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

```
-DFOO=
```

This option can be used one or more times on the command line.

Example

You may want to arrange your source to produce either the test or production version of your program, depending on whether the symbol `TESTVER` was defined. To do this, you would use include sections such as:

```
#ifdef TESTVER
... additional code lines for test version only
#endif
```

Then, you would select the version required on the command line as follows:

Production version: `iccavr prog`
 Test version: `iccavr prog -DTESTVER`



To set the equivalent option in IAR Embedded Workbench, select **Project>Options>C/C++ Compiler>Preprocessor**.

`--debug, -r` `--debug`
`-r`

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

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



To set the equivalent option in IAR Embedded Workbench, select **Project>Options>C/C++ Compiler>Output**.

`--dependencies` `--dependencies=[i|m] {filename|directory}`

Use this option to make the compiler write information to a file about each source code file opened by the compiler. The following modifiers are available:

Option modifier	Description
<code>i</code>	Lists only the names of files (default)
<code>m</code>	Lists in makefile style

Table 44: Generating a list of dependencies (`--dependencies`)

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 extension `i`. The filename will be the same as the name of the compiled source file, unless a different name has been specified with the option `-o`, in which case that name will be used.

To specify the working directory, replace *directory* with a period (`.`).

If `--dependencies` or `--dependencies=i` is used, the name of each opened source file, including the full path, if available, is output on a separate line. For example:

```
c:\iar\product\include\stdio.h
d:\myproject\include\foo.h
```

If `--dependencies=m` is used, the output uses makefile style. For each source file, one line containing a makefile dependency rule is produced. Each line consists of the name of the object file, a colon, a space, and the name of a source file.

For example:

```
foo.r90: c:\iar\product\include\stdio.h
foo.r90: d:\myproject\include\foo.h
```

Example 1

To generate a listing of file dependencies to the file `listing.i`, use:

```
iccavr prog --dependencies=i listing
```

Example 2

To generate a listing of file dependencies to a file called `listing.i` in the `mypath` directory, you would use:

```
iccavr prog --dependencies mypath\listing
```

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

Example 3

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, besides 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.

```
--diag_error --diag_error=tag, tag, ...
```

Use this option to classify 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, and the exit code will be non-zero.

Example

The following example classifies warning `Pe117` as an error:

```
--diag_error=Pe117
```



To set the equivalent option in IAR Embedded Workbench, select **Project>Options>C/C++ Compiler>Diagnostics**.

```
--diag_remark --diag_remark=tag, tag, ...
```

Use this option to classify diagnostic messages as remarks. A remark is the least severe type of diagnostic message and indicates a source code construct that may cause strange behavior in the generated code.

Example

The following example classifies the warning Pe177 as a remark:

```
--diag_remark=Pe177
```



To set the equivalent option in IAR Embedded Workbench, select **Project>Options>C/C++ Compiler>Diagnostics**.

```
--diag_suppress --diag_suppress=tag, tag, ...
```

Use this option to suppress diagnostic messages.

Example

The following example suppresses the warnings Pe117 and Pe177:

```
--diag_suppress=Pe117, Pe177
```



To set the equivalent option in IAR Embedded Workbench, select **Project>Options>C/C++ Compiler>Diagnostics**.

```
--diag_warning --diag_warning=tag, tag, ...
```

Use this option to classify 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.

Example

The following example classifies the remark Pe826 as a warning:

```
--diag_warning=Pe826
```



To set the equivalent option in IAR Embedded Workbench, select **Project>Options>C/C++ Compiler>Diagnostics**.

```
--diagnostics_tables --diagnostics_tables {filename|directory}
```

Use this option to list all possible diagnostic messages in a named file. This can be very 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.

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 name `diagnostics_tables.txt`. To specify the working directory, replace *directory* with a period (`.`).

Example 1

To output a list of all possible diagnostic messages to the file `diag.txt`, use:

```
--diagnostics_tables diag
```

Example 2

If you want to generate a table to a file `diagnostics_tables.txt` in the working directory, you could use:

```
--diagnostics_tables .
```

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

```
--disable_direct_mode --disable_direct_mode
```

This option prevents the compiler from generating the direct addressing mode instructions `LDS` and `STS`.

Using this option may in some cases reduce the size of the object code.

```
--dlib_config --dlib_config filename
```

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

All prebuilt runtime libraries are delivered with corresponding configuration files. You can find the library object files and the library configuration files in the directory `avr\lib\dlib`. For examples and a list of prebuilt runtime libraries, see *Using a prebuilt library*, page 56.

If you build your own customized runtime library, you should also create a corresponding customized library configuration file, which must be specified to the compiler. For more information, see *Building and using a customized library*, page 62.

Note: This option only applies to the IAR DLIB runtime environment.



To set the related option in IAR Embedded Workbench, select **Project>Options>General Options>Library Configuration**.

`--do_cross_call` `--do_cross_call`

Using this option forces 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 (z9).



To set the related option in IAR Embedded Workbench, select **Project>Options>C/C++ Compiler>Optimization**.

`-e` `-e`

In the command line version of the AVR IAR C/C++ Compiler, language extensions are disabled by default. If you use language extensions such as AVR-specific keywords and anonymous structs and unions in your source code, you must enable them by using this option.

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

For additional information, see *Special support for embedded systems*, page 12.



To set the equivalent option in IAR Embedded Workbench, select **Project>Options>C/C++ Compiler>Language**.

`--ec++` `--ec++`

In the AVR IAR C/C++ 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++.



To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>C/C++ Compiler>Language**.

```
--eec++ --eec++
```

In the AVR IAR C/C++ Compiler, the default language is C. If you take advantage of Extended Embedded C++ features like namespaces or the standard template library in your source code, you must use this option to set the language the compiler uses to Extended Embedded C++. See *IAR Extended Embedded C++*, page 110.



To set the equivalent option in IAR Embedded Workbench, select **Project>Options>C/C++ Compiler>Language**.

```
--eecr_address --eecr_address address
```

If you use the `-v` processor option, the `--eecr_address` option can be used for modifying the value of the `EECR` address, where `0x1C` is the default.

If you use the `--cpu` processor option, the `--eecr_address` option is implicitly set, which means you should not use these options together.

```
--enable_multibytes --enable_multibytes
```

By default, multibyte characters cannot be used in C or C++ source code. If you use this option, multibyte characters in the source code are 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.



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

```
--eeprom_size --eeprom_size=N
```

Use this option to enable the `__eeprom` extended keyword by specifying the size of the built-in EEPROM. The value *N* can be 0-65536.

Note: To use the `__eeprom` extended keyword, the language extensions must be enabled. For additional information, see *-e*, page 179, and *#pragma language*, page 220.



This option is related to the **Code** options in the **C/C++ Compiler** category in IAR Embedded Workbench.

```
--enable_external_bus --enable_external_bus
```

This option will make the compiler add the special `__require` statement which will make XLINK include the code in `cstartup.s90` that enables the external data bus. Use this option if you intend to place RSTACK in external RAM.

Note: The code in `cstartup.s90` that enables the external data bus is preferably placed in `low_level_inti` instead.



To set the equivalent option in IAR Embedded Workbench, choose **Project>Options>General Options>System**.

```
--enhanced_core --enhanced_core
```

Use this option to allow the compiler to generate instructions from the enhanced instruction set that is available in some AVR derivatives, for example ATmega161.

The enhanced instruction set consists of the following instructions:

```
MUL
MOVW
MULS
MULSU
FMUL
FMULS
FMULSU
LPM Rd, Z
LPM Rd, Z+
ELPM Rd, Z
ELPM Rd, Z+
SPM
```



This option corresponds to the **Enhanced core** option in the **General Options** category in IAR Embedded Workbench.

```
--error_limit --error_limit=n
```

Use the `--error_limit` option to specify the number of errors allowed before the compiler stops the compilation. By default, 100 errors are allowed. *n* must be a positive number; 0 indicates no limit.

`-f filename`

Reads command line options from the named file, with the default extension `.xcl`.

By default, the compiler accepts command parameters only from the command line itself and the `QCCAVR` environment variable. To make long command lines more manageable, and to avoid any operating system command line length limit, you can use the `-f` option to specify a command file, from which the compiler reads command line items as if they had been entered at the position of the option.

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 as in the Microsoft Windows command line environment.

Example

For example, you could replace the command line:

```
iccavr prog -r "-DUsername=John Smith" -DUserid=463760
```

with

```
iccavr prog -r -f userinfo
```

if the file `userinfo.xcl` contains:

```
"-DUsername=John Smith"
-DUserid=463760
```

`--force_switch_type` `--force_switch_type[0|1|2]`

Sets the switch type:

Option modifier	Switch type
0	Library call with switch table
1	Inline code with switch table
2	Inline compare/jump logic

Table 45: Specifying switch type

```
--header_context --header_context
```

Occasionally, to find the cause of a problem it is necessary to know which header file 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.

```
-I -Ipath
-I path
```

Use this option to specify the search path for `#include` files. This option may be used more than once on a single command line.

Following is the full description of the compiler's `#include` file search procedure:

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

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

it searches the following directories for the file to include:

- 1 The directories specified with the `-I` option, in the order that they were specified.
- 2 The directories specified using the `C_INCLUDE` environment variable, if any.

- If the compiler encounters the name of an `#include` file in double quotes, for example:

```
#include "vars.h"
```

it searches the directory of the source file in which the `#include` statement occurs, and then performs the same sequence as for angle-bracketed filenames.

If there are nested `#include` files, the compiler starts searching the directory of the file that was last included, iterating upwards for each included file, searching the source file directory last. For example:

```
src.c in directory dir\src
#include "src.h"
...
src.h in directory dir\include
#include "config.h"
...
```

When `dir\exe` is the current directory, use the following command for compilation:

```
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.
<code>dir\src</code>	File including current file.
<code>dir\include</code>	As specified with the first <code>-I</code> option.
<code>dir\debugconfig</code>	As specified with the second <code>-I</code> option.

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

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



To set the equivalent option in IAR Embedded Workbench, select **Project>Options>C/C++ Compiler>Preprocessor**.

`--initializers_in_flash` `--initializers_in_flash`

Use this option to place aggregate initializers in flash memory. These initializers are otherwise placed either in the external segments `_C` or in the initialized data segments if the compiler option `-y` was also specified.

For example:

```
void foo ()
{
    char buf[4] = { 'l', 'd', 'g', 't' };
    ...
}
```

In other words: 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.

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.

Local variables with aggregate initializers are copied from the segments:

Data	Default	<code>-y</code>	<code>--initializers_in_flash</code>
auto aggregates	NEAR_C	NEAR_F	NEAR_F

Table 46: Accessing variables with aggregate initializers

See `-y`, page 200, and the chapter *Initialization of local aggregates at function invocation*, page 40 for additional information.



This option is related to the **Code** options in the **C/C++ Compiler** category in IAR Embedded Workbench.

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

By default, the compiler does not generate a listing. Use this option to generate a listing to a file.

The following modifiers are available:

Option modifier	Description
a	Assembler list file
A	Assembler file with C or C++ source as comments
b	Basic assembler list file. This file has the same contents as a list file produced with <code>-la</code> , except that none of the extra compiler generated information (runtime model attributes, call frame information, frame size information) is included *
B	Basic assembler list file. This file has the same contents as a list file produced with <code>-lA</code> , except that none of the 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

Table 47: Generating a compiler list file (-l)

*** This makes the list file less useful as input to the assembler, but more useful for reading by a human.**

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 extension `lst`. The filename will be the same as the name of the compiled source file, unless a different name has been specified with the option `-o`, in which case that name will be used.

To specify the working directory, replace *directory* with a period (`.`).

Example 1

To generate a listing to the file `list.lst`, use:

```
iccavr prog -l list
```

Example 2

If you compile the file `mysource.c` and want to generate a listing to a file `mysource.lst` in the working directory, you could use:

```
iccavr mysource -l .
```

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



To set the equivalent option in IAR Embedded Workbench, select **Project>Options>C/C++ Compiler>List**.

```
--library_module --library_module
```

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.



To set the equivalent option in IAR Embedded Workbench, select **Project>Options>C/C++ Compiler>Output**.

```
--lock_regs --lock_regs N
```

Use this option to lock registers that are to be used for global register variables. The value *N* can be 0–12 where 0 means that no registers are locked. When you use this option, the registers R15 and downwards will be locked.

In order to maintain module consistency, make sure to lock the same number of registers in all modules.

Note: Locking more than nine registers may cause linking to fail. Because the pre-built libraries delivered with the product do not lock any registers, a library function may 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.



This option is related to the **Code** options in the **C/C++ Compiler** category in IAR Embedded Workbench.

```
-m, --memory_model -m[tiny|t|small|s|large|l]
--memory_model=[tiny|t|small|s|large|l]
```

Specifies the memory model for which the code is to be generated.

By default the compiler generates code for the Tiny memory model for all processor options except `-v4` and `-v6` where the Small memory model is the default.

Use the `-m` or the `--memory_model` option if you want to generate code for a different memory model.

For example, to generate code for the Large memory model, give the command:

```
iccavr filename -ml
```

or:

```
iccavr filename --memory_model=large
```

To read more about the available memory models, see *Memory model*, page 9.



These options are related to the **Memory model** option in the **General Options** category in IAR Embedded Workbench.

```
--migration_preprocessor_extensions
```

If you need to migrate code from an earlier IAR C or C/C++ compiler, you may want to use this option. With this option, the following can be used in preprocessor expressions:

- Floating-point expressions
- Basic type names and `sizeof`
- All symbol names (including typedefs and variables).

Note: If you use this option, not only will the compiler accept code that is not standard conformant, but it will also reject some code that *does* conform to the standard.

Important! Do not depend on these extensions in newly written code, as support for them may be removed in future compiler versions.

```
--misrac --misrac[={tag1,tag2-tag3,...|all|required}]
```

Use this option to enable the compiler to check for deviations from the rules described in the *MISRA Guidelines for the Use of the C Language in Vehicle Based Software*. By using one or more arguments with the option, you can restrict the checking to a specific subset of the MISRA C rules. The possible arguments are described in this table:

Command line option	Description
<code>--misrac</code>	Enables checking for all MISRA C rules
<code>--misrac=<i>n</i></code>	Enables checking for the MISRA C rule with number <i>n</i>
<code>--misrac=<i>m</i>,<i>n</i></code>	Enables checking for the MISRA C rules with numbers <i>m</i> and <i>n</i>

Table 48: Enabling MISRA C rules (`--misrac`)

Command line option	Description
<code>--misrac=k-n</code>	Enables checking for all MISRA C rules with numbers from k to n
<code>--misrac=k, m, r-t</code>	Enables checking for MISRA C rules with numbers k , m , and from r to t
<code>--misrac=all</code>	Enables checking for all MISRA C rules
<code>--misrac=required</code>	Enables checking for all MISRA C rules categorized as required

Table 48: Enabling MISRA C rules (`--misrac`) (Continued)

If the compiler is unable to check for a rule, specifying the option for that rule has no effect. For instance, MISRA C rule 15 is a documentation issue, and the rule is not checked by the compiler. As a consequence, specifying `--misrac=15` has no effect.



To set the equivalent option in IAR Embedded Workbench, select **Project>Options>General Options>MISRA C** or **Project>Options>C/C++ Compiler>MISRA C**.

`--misrac_verbose` `--misrac_verbose`

Use this option to generate a MISRA C log during compilation and linking. This is a list of the rules that are enabled—but not necessarily checked—and a list of rules that are actually checked.

If this option is enabled, the compiler displays a text at sign-on that shows both enabled and checked MISRA C rules.



To set the equivalent option in IAR Embedded Workbench, select **Project>Options>General Options>MISRA C**.

`--module_name` `--module_name=name`

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.

To set the object module name explicitly, use the option `--module_name=name`, for example:

```
iccavr prog --module_name=main
```

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.

Example

The following example—in which %1 is an operating system variable containing the name of the source file—will give duplicate name errors from the linker:

```
preproc %1.c temp.c                ; preprocess source,
                                   ; generating temp.c
iccavr temp.c                       ; module name is
                                   ; always 'temp'
```

To avoid this, use `--module_name=name` to retain the original name:

```
preproc %1.c temp.c                ; preprocess source,
                                   ; generating temp.c
iccavr temp.c --module_name=%1     ; use original source
                                   ; name as module name
```

Note: In this example, `preproc` is an external utility.



To set the equivalent option in IAR Embedded Workbench, select **Project>Options>C/C++ Compiler>Output**.

`--no_clustering` `--no_clustering`

This option will disable clustering of variables. Clustered variables may be accessed through a common base pointer which will, in most cases, reduce the size of the generated code.

`--no_code_motion` `--no_code_motion`

Use this option to disable optimizations that move code. These optimizations, which are performed at optimization levels 6 and 9, normally reduce code size and execution time. However, the resulting code may be difficult to debug.

Note: This option has no effect at optimization levels below 6.



To set the equivalent option in IAR Embedded Workbench, select **Project>Options>C/C++ Compiler>Code**.

`--no_cross_call` `--no_cross_call`

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`, since this option reduces the usage of `RSTACK`.

This optimization is performed at size optimization, level 7–9. Note that, although it can drastically reduce the code size, this option increases the execution time.

For additional information, see `--cross_call_passes`, page 174.



This option is related to the **Optimization** options in the **C/C++ Compiler** category in IAR Embedded Workbench.

`--no_cse` `--no_cse`

Use `--no_cse` to disable common subexpression elimination.

At optimization levels 6 and 9, the compiler avoids calculating the same expression more than once. This optimization normally reduces both code size and execution time. However, the resulting code may be difficult to debug.

Note: This option has no effect at optimization levels below 6.



To set the equivalent option in IAR Embedded Workbench, select **Project>Options>C/C++ Compiler>Code**.

`--no_inline` `--no_inline`

Use `--no_inline` to disable function inlining.

Function inlining means that a simple function, whose definition is known at compile time, is integrated into the body of its caller to eliminate the overhead of the call.

This optimization, which is performed at optimization level 9, normally reduces execution time and increases code size. The resulting code may also be difficult to debug.

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

Note: This option has no effect at optimization levels below 9.



To set the equivalent option in IAR Embedded Workbench, select **Project>Options>C/C++ Compiler>Code**.

`--no_rampd` `--no_rampd`

Specifying this option makes 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`).

```
--no_ubrof_messages --no_ubrof_messages
```

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.



This option is related to the **Output** options in the **C/C++ Compiler** category in IAR Embedded Workbench.

```
--no_tbaa --no_tbaa
```

Use `--no_tbaa` to disable type-based alias analysis. When this options is not used, the compiler is free to assume that objects are only accessed through the declared type or through `unsigned char`. See *Type-based alias analysis*, page 124 for more information.



This option is related to the **Optimization** options in the **C/C++ Compiler** category in IAR Embedded Workbench.

```
--no_typedefs_in_diagnostics --no_typedefs_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. For example,

```
typedef int (*MyPtr)(char const *);
MyPtr p = "foo";
```

will give an error message like the following:

```
Error[Pe144]: a value of type "char *" cannot be used to
initialize an entity of type "MyPtr"
```

If the `--no_typedefs_in_diagnostics` option is specified, the error message will be like this:

```
Error[Pe144]: a value of type "char *" cannot be used to
initialize an entity of type "int (*)(char const *)"
```

```
--no_warnings --no_warnings
```

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



To set the equivalent option in IAR Embedded Workbench, select **Project>Options>C/C++ Compiler>Diagnostics**.

```
--no_wrap_diagnostics --no_wrap_diagnostics
```

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

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

Use the `-o` option to specify an output file for object code.

If a *filename* is specified, the compiler stores the object code in that file.

If a *directory* is specified, the compiler stores the object code in that directory, in a file with the same name as the name of the compiled source file, but with the extension `.r90`. To specify the working directory, replace *directory* with a period (`.`).

Example 1

To store the compiler output in a file called `obj.r90` in the `mypath` directory, you would use:

```
iccavr mysource -o mypath\obj
```

Example 2

If you compile the file `mysource.c` and want to store the compiler output in a file `mysource.r90` in the working directory, you could use:

```
iccavr mysource -o .
```

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



To set the equivalent option in IAR Embedded Workbench, select **Project>Options>General Options>Output**.

```
--omit_types --omit_types
```

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. 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, which is useful when you build a library that should not contain type information.

```
--only_stdout --only_stdout
```

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

```
--preinclude --preinclude includefile
```

Use this option to make the compiler include the specified include file before it starts to read the source file. This is useful if you want to change something in the source code for the entire application, for instance if you want to define a new symbol.

```
--preprocess --preprocess[=[c][n][l]] {filename|directory}
```

Use this option to direct preprocessor output to a named file.

The following table shows the mapping of the available preprocessor modifiers:

Command line option	Description
<code>--preprocess=c</code>	Preserve comments
<code>--preprocess=n</code>	Preprocess only
<code>--preprocess=l</code>	Generate <code>#line</code> directives

Table 49: Directing preprocessor output to file (`--preprocess`)

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 extension `i`. The filename will be the same as the name of the compiled source file, unless a different name has been specified with the option `-o`, in which case that name will be used.

To specify the working directory, replace *directory* with a period (`.`).

Example 1

To store the compiler output with preserved comments to the file `output.i`, use:

```
iccavr prog --preprocess=c output
```

Example 2

If you compile the file `mysource.c` and want to store the compiler output with `#line` directives to a file `mysource.i` in the working directory, you could use:

```
iccavr mysource --preprocess=l .
```

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



To set the equivalent option in IAR Embedded Workbench, select **Project>Options>C/C++ Compiler>Preprocessor**.

`--public_equ` `--public_equ symbol [=value]`

This option is equivalent to defining a label in assembler language by using the `EQU` directive and exporting it using the `PUBLIC` directive.

`-r`, `--debug` `-r`
`--debug`

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

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



To set the equivalent option in IAR Embedded Workbench, select **Project>Options>C/C++ Compiler>Output**.

`--remarks` `--remarks`

The least severe diagnostic messages are called remarks (see *Severity levels*, page 279). 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.



To set the equivalent option in IAR Embedded Workbench, select **Project>Options>C/C++ Compiler>Diagnostics**.

`--require_prototypes` `--require_prototypes`

This option forces 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.



To set the equivalent option in IAR Embedded Workbench, select **Project>Options>C/C++ Compiler>Language**.

```
--root_variables --root_variables
```

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.

Notice that the `--root_variables` option is always available, even if you do not specify the compiler option `-e`, language extensions.



This option is related to the **Code** options in the **C/C++ Compiler** category in IAR Embedded Workbench.

```
-s -s[2|3|6|9]
```

Use this option to make the compiler optimize the code for maximum execution speed.

If no optimization option is specified, the compiler will use the size optimization `-z2` by default. If the `-s` option is used without specifying the optimization level, speed optimization at level 2 is used by default.

The following table shows how the optimization levels are mapped:

Option modifier	Optimization level
2	None* (Best debug support)
3	Low*
6	Medium
9	High (Maximum optimization)

Table 50: Specifying speed optimization (`-s`)

***The most important difference between `-s2` and `-s3` is that at level 2, all non-static variables will live during their entire scope.**

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.

Note: The `-s` and `-z` options cannot be used at the same time.



To set the equivalent option in IAR Embedded Workbench, select **Project>Options>C/C++ Compiler>Code**.

```
--segment --segment memory_attribute=segment_name
```

Use this option to place all variables or functions with the memory attribute `memory_attribute` in segments with names that begin with `segment_name`.

For example, the following command places the `__near int a;` variable in the `FOO_Z` segment:

```
--segment __near=FOO
```

For descriptions of the available memory attributes, see Table 7, *Memory attributes for data* on page 19 and Table 8, *Memory attributes for functions* on page 28. For a description of the segment name suffixes, see *Segment naming*, page 37.

`--separate_cluster_for_initialized_variables`

`--separate_cluster_for_initialized_variables`

Separates initialized and non-initialized variables when using variable clustering. Makes the `*_ID` segments smaller but can generate bigger code.

`--silent` `--silent`

By default, the compiler issues introductory messages and a final statistics report. Use `--silent` to make the compiler operate without sending these messages to the standard output stream (normally the screen).

This option does not affect the display of error and warning messages.

`--spmcr_address` `--spmcr_address address`

This option sets the SPMCR address, where `0x37` is the default.

`--strict_ansi` `--strict_ansi`

By default, the compiler accepts a relaxed superset of ISO/ANSI C (see the chapter *IAR language extensions*). Use `--strict_ansi` to ensure that the program conforms to the ISO/ANSI C standard.

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



To set the equivalent option in IAR Embedded Workbench, select **Project>Options>C/C++ Compiler>Language**.

`--string_literals_in_flash` `--string_literals_in_flash`

Puts "string" in the `__nearflash` or `__farflash` segment depending on the processor option.

When this option is used, library functions taking a string literal as a parameter will no longer be type-compatible. Use the `*_P` library function variants (for example `printf_P`). See *AVR-specific library functions*, page 249.

`-v` `-v[0|1|2|3|4|5|6]`

Use this option to select the processor derivative for which the code is to be generated. The following processor variants are available:

Generic processor option	Processor variant	Description
<code>-v0</code>	AT90S2313	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.
	AT90S2323	
	AT90S2333	
	AT90S2343	
	AT90S4433	
	ATtiny13	
ATtiny26		
ATtiny2313		
<code>-v1</code>	AT90S4414	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.
	AT90S4434	
	AT90S8515	
	AT90S8534	
	AT90S8535	
	ATmega8	
	ATmega48	
	ATmega88	
ATmega8515		
ATmega8535		

Table 51: Processor variant command line options

Generic processor option	Processor variant	Description
-v2	Reserved for future derivatives	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.
-v3	ATmega16 ATmega32 ATmega64 ATmega103 ATmega128 ATmega161 ATmega162 ATmega163 ATmega168 ATmega169 ATmega323 FpSLic (at94k)	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.
-v4	Reserved for future derivatives	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.

Table 51: Processor variant command line options (Continued)

Generic processor option	Processor variant	Description
-v5	Reserved for future derivatives	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 <code>_C</code> segments.
-v6	Reserved for future derivatives	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 <code>_C</code> segments.

Table 51: Processor variant command line options (Continued)

See also `--cpu`, page 173, and *Processor configuration*, page 5.



This option is related to the **Processor configuration** option in the **General Options** category in IAR Embedded Workbench.

`--version1_calls` `--version1_calls`

This option is provided for backward compatibility. It makes all functions and function calls use the calling convention of the A90 IAR Compiler, ICCA90, which is described in *Calling convention*, page 99.

To change the calling convention of a single function, use the `__version_1` function type attribute. See `__version_1`, page 214, for detailed information.



This option is related to the **Code** options in the **C/C++ Compiler** category in IAR Embedded Workbench.

`--warnings_affect_exit_code` `--warnings_affect_exit_code`

By default, the exit code is not affected by warnings, as only errors produce a non-zero exit code. With this option, warnings will generate a non-zero exit code.



To set the equivalent option in IAR Embedded Workbench, select **Project>Options>C/C++ Compiler>Diagnostics**.

`--warnings_are_errors` `--warnings_are_errors`

Use this option to make the compiler treat all warnings as errors. If the compiler encounters an error, no object code is generated. Warnings that have been changed into remarks are not treated as errors.

Note: Any diagnostic messages that have been reclassified as warnings by the compiler option `--diag_warning` or the `#pragma diag_warning` directive will also be treated as errors when `--warnings_are_errors` is used.

For additional information, see `--diag_warning`, page 177 and `#pragma diag_warning`, page 219.



To set the equivalent option in IAR Embedded Workbench, select **Project>Options>C/C++ Compiler>Diagnostics**.

`-y` `-y`

Use this option to override the default placement of constants and literals.

Without this option, constants and literals are placed in an external `const` segment, `segmentbasename_C`. *With* this option, constants and literals will instead be placed in the initialized `segmentbasename_I` data segments that are copied from the `segmentbasename_ID` segments by the system startup code.

Note that `-y` is implicit in the tiny memory model.

This option can be combined with the option `--initializers_in_flash`; see `--initializers_in_flash`, page 184 for additional information.



This option is related to the **Code** options in the **C/C++ Compiler** category in IAR Embedded Workbench.

`-z` `-z [2 | 3 | 6 | 9]`

Use this option to make the compiler optimize the code for minimum size. If no optimization option is specified, `-z3` is used by default.

The following table shows how the optimization levels are mapped:

Option modifier	Optimization level
2	None* (Best debug support)
3	Low*
6	Medium
9	High (Maximum optimization)

Table 52: Specifying size optimization (`-z`)

***The most important difference between `-z2` and `-z3` is that at level 2, all non-static variables will live during their entire scope.**

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.

Note: The `-s` and `-z` options cannot be used at the same time.



To set the equivalent option in IAR Embedded Workbench, select **Project>Options>C/C++ Compiler>Code**.

`--zero_register` `--zero_register`

Enabling this option will make the compiler use register R15 as zero register, that is register R15 is assumed to always contain zero.

This option can in some cases reduce the size of the generated code, especially in the Large memory model. The option may be incompatible with the supplied runtime libraries. The linker will issue a link-time error if any incompatibilities arise.

`--64bit_doubles` `--64bit_doubles`

Use this option to force the compiler to use 64-bit doubles instead of 32-bit doubles which is the default. For additional information, see *Floating-point types*, page 139.



This option is related to the **Target** options in the **General Options** category in IAR Embedded Workbench.

`--64k_flash` `--64k_flash`

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 `RAMPZ` register or the `ELPM` instruction. This option can only be used together with the `-v2`, `-v3`, and `-v4` processor options.



To set the equivalent option in IAR Embedded Workbench, select **Project>Options>General Options>Target (No RAMPZ register)**.

Extended keywords

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

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

Summary of extended keywords

Some extended keywords are used on data, some on functions, and some can be used on both data and functions.

The keywords and the @ operator are only available when language extensions are enabled in the AVR IAR C/C++ Compiler.



In IAR Embedded Workbench, language extensions are enabled by default.



Use the `-e` compiler option to enable language extensions. See `-e`, page 179 for additional information.

The following table summarizes the extended keywords that can be used on functions:

Extended keywords for functions	Description
@	Controls the storage of variables and functions
asm, __asm	Inserts an assembler instruction
__farfunc	Controls the storage of functions in code memory space
__interrupt	Creates an interrupt function
__intrinsic	Reserved for compiler internal use only
__monitor	Supports atomic execution of a function
__nearfunc	Controls the storage of functions in code memory space
__noreturn	Informs the compiler that the declared function will not return
__root	Ensures that a function or variable is included in the object code even if unused
__task	Allows functions to exit without restoring registers
__version_1	Uses old calling convention; available for backward compatibility.

Table 53: Summary of extended keywords for functions

The following table summarizes the extended keywords that can be used on data:

Extended keywords for data	Description
@	Controls the storage of data objects and functions
__eeprom	Controls the storage of data objects in code memory space
__ext_io	Controls the storage of data objects in I/O memory space Supports I/O instructions; used for SFRs
__far	Controls the storage of data objects in data memory space
__farflash	Controls the storage of data objects in code memory space
__flash	Controls the storage of data objects in code memory space
__generic	Declares a generic pointer
__huge	Controls the storage of data objects in data memory space
__hugeflash	Controls the storage of data objects in code memory space
__io	Controls the storage of data objects in I/O memory space Supports I/O instructions; used for SFRs
__near	Controls the storage of data objects in data memory space
__no_init	Supports non-volatile memory
__regvar	Places a data object in a register
__root	Ensures that a function or data object is included in the object code even if unused
__tiny	Controls the storage of data objects in data memory space
__tinyflash	Controls the storage of data objects in code memory space

Table 54: Summary of extended keywords for data

Descriptions of extended keywords

The following sections give detailed information about each extended keyword.

- @ The @ operator can be used for placing global and static variables at absolute addresses. The syntax can also be used for placing variables and functions in named segments.

For more information about the @ operator, see *Efficient usage of segments and memory*, page 47.

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

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

Syntax

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

The string can be a valid assembler instruction or an 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 93.

`__eeprom` Controls the storage of data objects in eeprom memory space.

The `__eeprom` memory attribute allows you to place 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.

You can also override the supplied support functions to make the `__eeprom` 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.

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 `eeprom.s90` file in the `avr\src\lib\` directory (visible register use, the same entry points and the same semantics).

Note: Variables declared `__eeprom` are initialized only when a downloadable linker output file is downloaded to the system, and not every time the system startup code is executed.

Address range	Max object size	Pointer size	Memory space
0-0xFF (255 bytes)	255 bytes	8 bits	Eeprom ¹⁾
0-0xFFFF (64 kbytes)	65535 bytes	16 bits	Eeprom ²⁾

Table 55: EEPROM address ranges

1) `--eeprom_size ≤ 256 bytes`

2) `--eeprom_size ≥ 512 bytes`

`__ext_io` Controls the storage of data objects in data memory space.

The `__ext_io` memory attribute implies that objects are `__no_init` and volatile, and allows objects to be accessed by use of the special I/O instructions in the AVR microcontroller.

Address range	Max object size	Pointer size	Memory space
0x100-0xFFFF	4 bytes (32 bits)	Pointers not allowed	Data

Table 56: Near address ranges

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 `__ext_io` memory attribute should be included in the code.

`__far` Controls the storage of data objects in data memory space.

The `__far` memory attribute overrides the default data storage of variables given by the selected memory model.

Address range	Max object size	Pointer size	Memory space
0-0xFFFFFFFF (16 Mbytes)	65535 bytes	24 bits	Data

Table 57: Far address ranges

Note: When the `__far` memory attribute is used, the object cannot cross a 64-Kbyte boundary. Arithmetics will only be performed on the two lower bytes, except comparison which is always performed on the entire 24-bit address.

`__farflash` Controls the storage of data objects in flash (code) memory space.

The `__farflash` memory attribute places objects in flash (code) memory.

Note that it is preferable to declare flash objects as `const`. The `__farflash` memory attribute is only available for AVR chips with at least 64 Kbytes of flash memory.

The index type uses 16-bit arithmetic, which implies that a 64-Kbyte boundary cannot be crossed.

Address range	Max object size	Pointer size	Memory space
0-0x7FFFFFFF (8 Mbytes)	65535 bytes	24 bits	Code

Table 58: Farflash address ranges

Note: When the `__farflash` memory attribute is used, the object cannot cross a 64-Kbyte boundary. Arithmetics will only be performed on the two lower bytes, except comparison which is always performed on the entire 24-bit address.

`__farfunc` Controls the storage of functions in code memory space.

The `__farfunc` memory attribute places a function in farfunc code memory space.

Functions declared `__farfunc` have no restrictions on code placement, and can be placed anywhere in code memory.

The default code pointer for the `-v5` and `-v6` processor options is `__farfunc`, and it only affects the size of the function pointers.

Address range	Pointer size
0-0x7FFFFE (8 Mbytes)	24 bits

Table 59: Farfunc pointer size

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.

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

`__flash` Controls the storage of data objects in flash (code) memory space.

The `__flash` memory attribute places objects in flash (code) memory.

Note that it is preferable to declare flash objects as constant. The `__flash` keyword is available for all AVR chips.

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.

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.

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);
```

Address range	Max object size	Pointer size	Memory space
0-0xFFFF (64 Kbytes)	65535 bytes	16 bits	Code

Table 60: Flash address ranges

`__generic` Declares a generic pointer.

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:

Processor option	Generic pointer size	Memory space
-v0, -v1	l + 15 bits	Data/Code
-v2 to -v6	l + 23 bits	Data/Code

Table 61: Generic pointer size

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.

`__huge` Controls the storage of data objects in data memory space.

The `__huge` memory attribute overrides the default data storage of variables given by the selected memory model.

Address range	Max object size	Pointer size	Memory space
0-0xFFFFFFFF (16 Mbytes)	16 Mbytes	24 bits	Data

Table 62: Huge address ranges

`__hugeflash` Controls the storage of data objects in flash (code) memory space.

The `__hugeflash` memory attribute places objects in flash (code) memory.

Note that it is preferable to declare flash objects as constant. The `__hugeflash` memory attribute is only available for AVR chips with at least 64 Kbytes of flash memory.

Address range	Max object size	Pointer size	Memory region
0-0x7FFFFFFF (8 Mbytes)	8 Mbytes	24 bits	Code

Table 63: Hugeflash address ranges

`__interrupt` Creates an interrupt function.

The `__interrupt` keyword specifies interrupt functions. The `#pragma vector` directive can be used for specifying the interrupt vector(s), see *#pragma vector*, page 225. An interrupt function must have a `void` return type and cannot have any parameters.

The following example declares an interrupt function with an interrupt vector with offset 0x14 in the `INTVEC` segment:

```
#pragma vector=0x14
__interrupt void my_interrupt_handler(void);
```

An interrupt function cannot be called directly from a C program. It can only be executed as a response to an interrupt request.

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. For additional information, see *INTVEC*, page 160.

The range of the interrupt vectors depends on the device used.

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

For additional information, see *Interrupt functions*, page 29.

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

`__io` Controls the storage of data objects in I/O memory space, alternatively data memory space.

The `__io` memory attribute implies that objects are `__no_init` and volatile, and allows objects to be accessed by use of the special I/O instructions in the AVR microcontroller.

Address range	Max object size	Pointer size	Memory space
0-0x3F	4 bytes (32 bits)	Pointers not allowed	I/O
0x60-0xFF	4 bytes (32 bits)	Pointers not allowed	Data

Table 64: I/O address ranges

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 `__io` memory attribute should be included in the code.

`__monitor` Supports atomic execution of a function.

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. This keyword can be specified using the `#pragma object_attribute` directive.

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

For additional information, see the intrinsic functions `__disable_interrupt`, page 238, and `__enable_interrupt`, page 238.

Read more about monitor functions in *Monitor functions*, page 30.

`__near` Controls the storage of data objects in data memory space.

The `__near` memory attribute overrides the default data storage of variables given by the selected memory model.

Address range	Max object size	Pointer size	Memory space
0-0xFFFF (64 Kbytes)	65535 bytes	16 bits	Data

Table 65: Near address ranges

`__nearfunc` Controls the storage of functions in code memory space.

The `__nearfunc` memory attribute allows you to define the memory range where a function will be located.

Functions declared `__nearfunc` can be called from the entire code memory area, but must reside in the first 128 Kbytes of the code memory.

The default for the `-v0` to `-v4` processor options is `__nearfunc`, and it only affects the size of the function pointers.

Address range	Pointer size
0-0x1FFFE (128 Kbytes)	16 bits

Table 66: Nearfunc pointer size

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.

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

`__no_init` The `__no_init` keyword is used for suppressing initialization of a variable at system startup.

The `__no_init` keyword is placed in front of the type. In this example, `myarray` is placed in a non-initialized segment:

```
__no_init int myarray[10];
```

The `#pragma object_attribute` directive can also be used. The following declaration is equivalent to the previous one:

```
#pragma object_attribute=__no_init
int myarray[10];
```

Note: The `__no_init` keyword cannot be used in combination with the `typedef` keyword.

`__noreturn` The `__noreturn` keyword can be used on a function to inform the compiler that the function will not return. If you use this keyword on such functions, the compiler can optimize more efficiently. Examples of functions that do not return are `abort` and `exit`.

The `__noreturn` keyword is an object attribute, which means the `#pragma object_attribute` directive can be used for specifying it. For more information about object attributes, see *Object attributes*, page 145.

`__regvar` The `__regvar` extended keyword is used for declaring that a *global* or *static* 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 `--lock_regs` compiler option; see `--lock_regs`, page 186 for additional information. Also see *Preserved registers*, page 101.

To declare a global register variable, use the following syntax:

```
__regvar __no_init type_name variable_name @ Rlocation
```

This will create a variable called *variable_name* of type *type_name*, located in registers starting from *Rlocation*, for example:

```
__regvar __no_init int counter @ 14;
```

This will create the 16-bit integer variable `counter`, which will always be available in R15:R14. At least two registers must be locked. And, as noted in `-m, --memory_model`, page 186, if you lock any registers in your code, the library must be rebuilt with the same set of locked registers.

The maximum object size is 4 bytes (32 bits).

Note: It is *not* possible to point to an object that has been declared `__regvar`. An object declared `__regvar` cannot have an initial value.

`__root` The `__root` attribute can be used on a function or a variable to ensure that, when the module containing the function or variable is linked, the function or variable is also included, whether or not it is referenced by the rest of the program.

By default, only the part of the runtime library calling `main` and any interrupt vectors are root. All other functions and variables are included in the linked output only if they are referenced by the rest of the program.

The `__root` keyword is placed in front of the type, for example to place `myarray` in non-volatile memory:

```
__root int myarray[10];
```

The `#pragma object_attribute` directive can also be used. The following declaration is equivalent to the previous one:

```
#pragma object_attribute=__root
int myarray[10];
```

Note: The `__root` keyword cannot be used in combination with the `typedef` keyword.

`__task` Allows functions to exit without restoring registers. This keyword is typically used for the `main` function.

By default, functions save the contents of used non-scratch registers (permanent registers) on the stack upon entry, and restore them at exit. Functions declared as `__task` do not save any registers, and therefore require less stack space. Such functions should only be called from assembler routines.

The function `main` may be declared `__task`, unless it is explicitly called from the application. In real-time applications with more than one task, the root function of each task may be declared `__task`.

The keyword is placed in front of the return type, for instance:

```
__task void my_handler(void);
```

The `#pragma type_attribute` directive can also be used. The following declaration of `my_handler` is equivalent with the previous one:

```
#pragma type_attribute=__task
void my_handler(void);
```

The `__task` keyword must be specified both in the function declaration and when the function is defined.

`__tiny` Controls the storage of data objects in data memory space.

The `__tiny` memory attribute overrides the default data storage of variables given by the selected memory model.

Address range	Max object size	Pointer size	Memory space
0-0xFF (256 bytes)	255 bytes	8 bits	Data

Table 67: Tiny address ranges

`__tinyflash` Controls the storage of data objects in flash (code) memory space.

The `__tinyflash` memory attribute places objects in flash (code) memory.

Note that it is preferable to declare flash objects as constant.

Address range	Max object size	Pointer size	Memory space
0-0xFF (256 bytes)	255 bytes	8 bits	Code

Table 68: Tinyflash address ranges

`__version_1` The `__version_1` keyword is available for backward compatibility. It makes a function use the calling convention of the A90 IAR C Compiler instead of the default calling convention, both which are described in *Calling convention*, page 99.

This calling convention is preferred when calling assembler functions from C.S

Pragma directives

This chapter describes the pragma directives of the AVR IAR C/C++ Compiler.

The pragma directives control the behavior of the compiler, for example how it allocates memory, whether it allows extended keywords, and whether it outputs warning messages. The pragma directives are preprocessed, which means that macros are substituted in a pragma directive.

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

Summary of pragma directives

The following table shows the pragma directives of the compiler:

Pragma directive	Description
<code>#pragma</code>	Makes a template function fully memory-aware
<code>basic_template_matching</code>	
<code>#pragma bitfields</code>	Controls the order of bitfield members
<code>#pragma constseg</code>	Places constant variables in a named segment
<code>#pragma data_alignment</code>	Gives a variable a higher (more strict) alignment
<code>#pragma dataseg</code>	Places variables in a named segment
<code>#pragma diag_default</code>	Changes the severity level of diagnostic messages
<code>#pragma diag_error</code>	Changes the severity level of diagnostic messages
<code>#pragma diag_remark</code>	Changes the severity level of diagnostic messages
<code>#pragma diag_suppress</code>	Suppresses diagnostic messages
<code>#pragma diag_warning</code>	Changes the severity level of diagnostic messages
<code>#pragma include_alias</code>	Specifies an alias for an include file
<code>#pragma inline</code>	Inlines a function
<code>#pragma language</code>	Controls the IAR language extensions
<code>#pragma location</code>	Specifies the absolute address of a variable
<code>#pragma message</code>	Prints a message

Table 69: Pragma directives summary

Pragma directive	Description
<code>#pragma object_attribute</code>	Changes the definition of a variable or a function
<code>#pragma optimize</code>	Specifies type and level of optimization
<code>#pragma pack</code>	Specifies the alignment of structures and union members
<code>#pragma required</code>	Ensures that a symbol which is needed by another symbol is present in the linked output
<code>#pragma rtmodel</code>	Adds a runtime model attribute to the module
<code>#pragma segment</code>	Declares a segment name to be used by intrinsic functions
<code>#pragma type_attribute</code>	Changes the declaration and definitions of a variable or function
<code>#pragma vector</code>	Specifies the vector of an interrupt function

Table 69: Pragma directives summary (Continued)

Note: For portability reasons, some old-style pragma directives are recognized but will give a diagnostic message. It is important to be aware of this if you need to port existing code that contains any of those pragma directives. For additional information, see the *AVR® IAR Embedded Workbench Migration Guide*.

Descriptions of pragma directives

This section gives detailed information about each pragma directive.

All pragma directives using = for value assignment should be entered like:

```
#pragma pragmaname=pragmavalue
```

or

```
#pragma pragmaname = pragmavalue
```

```
#pragma basic_template_matching #pragma basic_template_matching
```

Use this pragma directive in front of a template function declaration to make the function fully memory-aware, in the rare cases where this is useful. That template function will then match the template without the modifications described in *Templates and data memory attributes*, page 116.

Example

```
#pragma basic_template_matching
template<typename T> void fun(T *);

fun((int __near *) 0); // T = int __near
```

```
#pragma bitfields #pragma bitfields={reversed|default}
```

The `#pragma bitfields` directive controls the order of bitfield members.

By default, the AVR IAR C/C++ Compiler places bitfield members from the least significant bit to the most significant bit in the container type. Use the `#pragma bitfields=reversed` directive to place the bitfield members from the most significant to the least significant bit. This setting remains active until you turn it off again with the `#pragma bitfields=default` directive.

```
#pragma constseg
```

The `#pragma constseg` directive places constant variables in a named segment. Use the following syntax:

```
#pragma constseg=MY_CONSTANTS
const int factorySettings[] = {42, 15, -128, 0};
#pragma constseg=default
```

The segment name cannot be a predefined segment; see the chapter *Segment reference* for more information.

The memory in which the segment resides is optionally specified using the following syntax:

```
#pragma constseg=__huge MyOtherSeg
```

All constants defined following this directive will be placed in the segment `MyOtherSeg` and accessed using huge addressing.

```
#pragma data_alignment #pragma data_alignment=expression
```

Use this pragma directive to give a variable a higher (more strict) alignment than it would otherwise have. It can be used on variables with static and automatic storage duration.

The value of the constant *expression* must be a power of two (1, 2, 4, etc.).

When you use `#pragma data_alignment` on variables with automatic storage duration, there is an upper limit on the allowed alignment for each function, determined by the calling convention used.

```
#pragma dataseg
```

The `#pragma dataseg` directive places variables in a named segment. Use the following syntax:

```
#pragma dataseg=MY_SEGMENT
__no_init char myBuffer[1000];
#pragma dataseg=default
```

The segment name cannot be a predefined segment, see the chapter *Segment reference* for more information. The variable `myBuffer` will not be initialized at startup, and can for this reason not have any initializer.

The memory in which the segment resides is optionally specified using the following syntax:

```
#pragma dataseg=__huge MyOtherSeg
```

All variables in `MyOtherSeg` will be accessed using huge addressing.

```
#pragma diag_default #pragma diag_default=tag, tag, ...
```

Changes the severity level back to default, or as defined on the command line for the diagnostic messages with the specified tags. For example:

```
#pragma diag_default=Pe117
```

See the chapter *Diagnostics* for more information about diagnostic messages.

```
#pragma diag_error #pragma diag_error=tag, tag, ...
```

Changes the severity level to `error` for the specified diagnostics. For example:

```
#pragma diag_error=Pe117
```

See the chapter *Diagnostics* for more information about diagnostic messages.

```
#pragma diag_remark #pragma diag_remark=tag, tag, ...
```

Changes the severity level to `remark` for the specified diagnostics. For example:

```
#pragma diag_remark=Pe177
```

See the chapter *Diagnostics* for more information about diagnostic messages.

```
#pragma diag_suppress #pragma diag_suppress=tag, tag, ...
```

Suppresses the diagnostic messages with the specified tags. For example:

```
#pragma diag_suppress=Pe117, Pe177
```

See the chapter *Diagnostics* for more information about diagnostic messages.

```
#pragma diag_warning #pragma diag_warning=tag, tag, ...
```

Changes the severity level to `warning` for the specified diagnostics. For example:

```
#pragma diag_warning=Pe826
```

See the chapter *Diagnostics* for more information about diagnostic messages.

```
#pragma include_alias #pragma include_alias "orig_header" "subst_header"
```

```
#pragma include_alias <orig_header> <subst_header>
```

The `#pragma include_alias` directive makes it possible 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.

The parameter `subst_header` is used for specifying an alias for `orig_header`. This pragma directive must appear before the corresponding `#include` directives and `subst_header` must match its corresponding `#include` directive exactly.

Example

```
#pragma include_alias <stdio.h> <C:\MyHeaders\stdio.h>
#include <stdio.h>
```

This example will substitute the relative file `stdio.h` with a counterpart located according to the specified path.

```
#pragma inline #pragma inline[=forced]
```

The `#pragma inline` directive advises the compiler that the function whose declaration follows immediately after the directive should be inlined—that is, expanded into the body of the calling function. Whether the inlining actually takes place is subject to the compiler's heuristics.

This is similar to the C++ keyword `inline`, but has the advantage of being available in C code.

Specifying `#pragma inline=forced` disables the compiler's heuristics and forces the inlining. If the inlining fails for some reason, for example if it cannot be used with the function type in question (like `printf`), an error message is emitted.

<code>#pragma language</code>	<p><code>#pragma language={extended default}</code></p> <p>The <code>#pragma language</code> directive is used for turning on the IAR language extensions or for using the language settings specified on the command line:</p> <table> <tr> <td style="vertical-align: top;"><code>extended</code></td> <td>Turns on the IAR language extensions and turns off the <code>--strict_ansi</code> command line option.</td> </tr> <tr> <td style="vertical-align: top;"><code>default</code></td> <td>Uses the settings specified on the command line.</td> </tr> </table>	<code>extended</code>	Turns on the IAR language extensions and turns off the <code>--strict_ansi</code> command line option.	<code>default</code>	Uses the settings specified on the command line.
<code>extended</code>	Turns on the IAR language extensions and turns off the <code>--strict_ansi</code> command line option.				
<code>default</code>	Uses the settings specified on the command line.				

<code>#pragma location</code>	<p><code>#pragma location=address</code></p> <p>The <code>#pragma location</code> directive specifies the location—the absolute address—of the variable whose declaration follows the pragma directive. For example:</p> <pre>#pragma location=0x2000 char PORT1; /* PORT1 is located at address 0x2000 */</pre> <p>The directive can also take a string specifying the segment placement for either a variable or a function, for example:</p> <pre>#pragma location="foo"</pre> <p>For additional information and examples, see <i>Located data</i>, page 46.</p>
-------------------------------	---

<code>#pragma message</code>	<p><code>#pragma message(message)</code></p> <p>Makes the compiler print a message on <code>stdout</code> when the file is compiled. For example:</p> <pre>#ifdef TESTING #pragma message("Testing") #endif</pre>
------------------------------	---

<code>#pragma object_attribute</code>	<p><code>#pragma object_attribute=keyword</code></p> <p>The <code>#pragma object_attribute</code> directive affects the definition of the identifier that follows immediately after the directive. The object is modified, not its type.</p> <p>The following keyword can be used with <code>#pragma object_attribute</code> for a variable:</p> <table> <tr> <td style="vertical-align: top;"><code>__no_init</code></td> <td>Suppresses initialization of a variable at startup.</td> </tr> </table> <p>The following keyword can be used with <code>#pragma object_attribute</code> for a variable:</p> <table> <tr> <td style="vertical-align: top;"><code>__monitor</code></td> <td>Supports atomic execution of a function.</td> </tr> </table>	<code>__no_init</code>	Suppresses initialization of a variable at startup.	<code>__monitor</code>	Supports atomic execution of a function.
<code>__no_init</code>	Suppresses initialization of a variable at startup.				
<code>__monitor</code>	Supports atomic execution of a function.				

The following keywords can be used with `#pragma object_attribute` for a function or variable:

<code>__root</code>	Ensures that a function or data object is included in the linked application, even if it is not referenced.
<code>__noreturn</code>	Informs the compiler that the function will not return.

Example

In the following example, the variable `bar` is placed in the non-initialized segment:

```
#pragma object_attribute=__no_init
char bar;
```

Unlike the directive `#pragma type_attribute` that specifies the storing and accessing of a variable, it is not necessary to specify an object attribute in declarations. The following example declares `bar` without a `#pragma object_attribute`:

```
__no_init char bar;
```

```
#pragma optimize #pragma optimize=token_1 token_2 token_3
```

where *token_n* is one of the following:

<code>s</code>	Optimizes for speed
<code>z</code>	Optimizes for size
<code>2 none 3 low 6 medium 9 high</code>	Specifies the level of optimization
<code>no_code_motion</code>	Turns off code motion
<code>no_cse</code>	Turns off common subexpression elimination
<code>no_inline</code>	Turns off function inlining
<code>no_tbaa</code>	Turns off type-based alias analysis

The `#pragma optimize` directive is used for decreasing the optimization level, or for turning off some specific optimizations. This pragma directive only affects the function that follows immediately after the directive.

Note that it is not possible to optimize for speed and size at the same time. Only one of the `s` and `z` tokens can be used. It is also not possible to use macros embedded in this pragma directive. Any such macro will not get expanded by the preprocessor.

Note: If you use the `#pragma optimize` directive to specify an optimization level that is higher than the optimization level you specify using a compiler option, the pragma directive is ignored.

Example

```
#pragma optimize=s 9
int small_and_used_often()
{
    ...
}

#pragma optimize=z 9
int big_and_seldom_used()
{
    ...
}
```

```
#pragma pack #pragma pack([ [{push|pop}, ] [name, ] ] [n])
```

n Packing alignment, one of: 1, 2, 4, 8, or 16

name Pushed or popped alignment label

The `#pragma pack` directive is used for specifying the alignment of structures and union members.

Note: In the AVR IAR Compiler, alignment is always 1 and this pragma directive has no effect. It is available in order to maintain compatibility with other IAR Systems compilers.

`pack(n)` sets the structure alignment to *n*. The `pack(n)` only affects declarations of structures following the pragma directive and to the next `#pragma pack` or end of file.

`pack()` resets the structure alignment to default.

`pack(push [, name] [, n])` pushes the current alignment with the label *name* and sets alignment to *n*. Note that both *name* and *n* are optional.

`pack(pop [, name] [, n])` pops to the label *name* and sets alignment to *n*. Note that both *name* and *n* are optional.

If *name* is omitted, only top alignment is removed. If *n* is omitted, alignment is set to the value popped from the stack.

```
#pragma required #pragma required=symbol
```

Use the `#pragma required` directive to ensure that a symbol which is needed by another symbol is present in the linked output. The *symbol* can be any statically linked function or variable, and the pragma directive must be placed immediately before a symbol definition.

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

```
void * const myvar_entry @ "MYSEG" = &myvar;
...
#pragma required=myvar_entry
long myvar;
```

```
#pragma rtmodel #pragma rtmodel="key", "value"
```

Use the `#pragma rtmodel` directive to add a runtime model attribute to a module. Use a text string to specify *key* and *value*.

This pragma directive is useful to enforce 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 value, or the special value `*`. Using the special value `*` is equivalent to not defining the attribute at all. It can, however, be useful to state explicitly that the module can handle any runtime model.

A module can have several runtime model definitions.

Note: The predefined compiler runtime model attributes start with a double underscore. In order to avoid confusion, this style must not be used in the user-defined attributes.

Example

```
#pragma rtmodel="I2C", "ENABLED"
```

The linker will generate an error if a module that contains this definition is linked with a module that does not have the corresponding runtime model attributes defined.

For more information about runtime model attributes and module consistency, see *Checking module consistency*, page 77.

```
#pragma segment #pragma segment="segment" [memattr] [align]
```

The `#pragma segment` directive declares a segment name that can be used by the intrinsic functions `__segment_begin` and `__segment_end`. All segment declarations for a specific segment must have the same memory type attribute and alignment.

The optional memory attribute `memattr` will be used in the return type of the intrinsic function. The optional parameter `align` can be specified to align the segment part. The value must be a constant integer expression to the power of two.

Example

```
#pragma segment="MYSEG" __huge 4
```

See also `__segment_begin`, page 241.

For more information about segments and segment parts, see the chapter *Placing code and data*.

```
#pragma type_attribute #pragma type_attribute=keyword
```

The `#pragma type_attribute` directive can be used for specifying IAR-specific *type attributes*, which are not part of the ISO/ANSI C language standard. Note however, that a given type attribute may not be applicable to all kind of objects. For a list of all supported type attributes, see *Type and object attributes*, page 144.

The `#pragma type_attribute` directive affects the declaration of the identifier, the next variable, or the next function that follows immediately after the pragma directive.

Example

In the following example, even though IAR-specific type attributes are used, the application can still be compiled by a different compiler. First, a `typedef` is declared; a `char` object with the memory attribute `__near` is defined as `MyCharInNear`. Then a pointer is declared; the pointer is located in far memory and it points to a `char` object that is located in near memory.

```
#pragma type_attribute=__near
typedef char MyCharInNear;
#pragma type_attribute=__far
MyCharInNear * ptr;
```

The following declarations, which use extended keywords, are equivalent. See the chapter *Extended keywords* for more details.

```
char __near * __far ptr;
```

```
#pragma vector #pragma vector=vector1[, vector2, vector3, ...]
```

The `#pragma vector` directive specifies the vector(s) of an interrupt function whose declaration follows the pragma directive.

Example

```
#pragma vector=0x14  
__interrupt void my_handler(void);
```


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 AVR IAR C/C++ Compiler adheres to the ISO/ANSI standard. The compiler also makes the following preprocessor-related features available to you:

- **Predefined preprocessor symbols.** These symbols allow you to inspect the compile-time environment, for example the time and date of compilation. Some of the symbols take arguments and perform more advanced operations than just inspecting the compile-time environment. For details, see *Predefined symbols*, page 227.
- **User-defined preprocessor symbols.** Use the option `-D` to define your own preprocessor symbols, see *-D*, page 174.
- **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 other extensions, see *Preprocessor extensions*, page 235.
- **Preprocessor output.** Use the option `--preprocess` to direct preprocessor output to a named file, see *--preprocess*, page 193.
- **Implementation-defined behavior.** Some parts listed by the ISO/ANSI standard are implementation-defined, for example the character set used in the preprocessor directives and inclusion of bracketed and quoted filenames. To read more about this, see *Preprocessing directives*, page 260.

Predefined symbols

This section first summarizes all predefined symbols and then provides detailed information about each symbol.

SUMMARY OF PREDEFINED SYMBOLS

The following table summarizes the predefined symbols:

Predefined symbol	Identifies
<code>__ALIGNOF__ ()</code>	Accesses the alignment of an object
<code>__BASE_FILE__</code>	Identifies the name of the file being compiled. If the file is a header file, the name of the file that includes the header file is identified.
<code>__CORE__</code>	Identifies the processor variant in use
<code>__CPU__</code>	Identifies the processor variant in use
<code>__cplusplus</code>	Determines whether the compiler runs in C++ mode*
<code>__DATE__</code>	Determines the date of compilation*
<code>__derivative__</code>	Corresponds to the processor specified with the <code>--cpu</code> compiler option
<code>__embedded_cplusplus</code>	Determines whether the compiler runs in C++ mode*
<code>__FILE__</code>	Identifies the name of the file being compiled*
<code>__func__</code>	Expands into a string with the function name as context
<code>__FUNCTION__</code>	Expands into a string with the function name as context
<code>__HAS_EEPROM__</code>	Determines whether there is an EEPROM available
<code>__IAR_SYSTEMS_ICC__</code>	Identifies the IAR compiler platform
<code>__ICCAVR__</code>	Identifies the AVR IAR C/C++ Compiler
<code>__LINE__</code>	Determines the current source line number*
<code>__MEMORY_MODEL__</code>	Identifies the memory model in use
<code>NDEBUG</code>	Determines whether asserts shall be included or not in the built application
<code>__Pragma ()</code>	Can be used in preprocessor defines and has the equivalent effect as the pragma directive
<code>__PRETTY_FUNCTION__</code>	Expands into a string with the function name, including parameter types and return type, as context
<code>__STDC__</code>	Identifies ISO/ANSI Standard C*
<code>__STDC_VERSION__</code>	Identifies the version of ISO/ANSI Standard C in use*
<code>__TID__</code>	Identifies the target processor of the IAR compiler in use
<code>__TIME__</code>	Determines the time of compilation*
<code>__VER__</code>	Identifies the version number of the IAR compiler in use
<code>__VERSION_1_CALLS__</code>	Identifies the calling convention in use

Table 70: Predefined symbols summary

* This symbol is required by the ISO/ANSI standard.

Note: The predefined symbol `__TID__` is available for backward compatibility. We recommend that you use the symbols `__ICCAVR__` and `__MEMORY_MODEL__` instead.

DESCRIPTIONS OF PREDEFINED SYMBOLS

The following section gives reference information about each predefined symbol.

<code>__ALIGNOF__()</code>	<p>The <code>__ALIGNOF__</code> operator is used to access the alignment of an object. It takes one of two forms:</p> <ul style="list-style-type: none"> ● <code>__ALIGNOF__</code> (<i>type</i>) ● <code>__ALIGNOF__</code> (<i>expression</i>) <p>In the second form, the expression is not evaluated.</p>
<hr/>	
<code>__BASE_FILE__</code>	<p>Use this symbol to identify which file is currently being compiled. This symbol expands to the name of that file, unless the file is a header file. In that case, the name of the file that includes the header file is identified.</p> <p>See also, <code>__FILE__</code>, page 230.</p>
<hr/>	
<code>__CORE__</code>	<p>This symbol identifies the used processor variant.</p> <p>This symbol expands to a number which corresponds to the processor option <code>-vn</code> in use.</p>
<hr/>	
<code>__CPU__</code>	<p>Use this symbol to identify the used processor variant.</p> <p>This symbol expands to a number which corresponds to the processor option <code>-vn</code> in use.</p>
<hr/>	
<code>__cplusplus</code>	<p>This predefined symbol expands to the number <code>199711L</code> when the compiler runs in any of the C++ modes. When the compiler runs in ISO/ANSI C mode, the symbol is undefined.</p> <p>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.</p>

<code>__DATE__</code>	Use this symbol to identify when the file was compiled. This symbol expands to the date of compilation, which is returned in the form "Mmm dd yyyy", for example "Jan 30 2002".
<code>__derivative__</code>	Use this symbol to identify the used processor variant. Corresponds to the processor variant that you have specified with the <code>--cpu</code> compiler option. <i>derivative</i> corresponds exactly to the derivative name in Table 2, <i>Mapping of processor options</i> on page 6, except for <code>FpSLic</code> , which uses the predefined symbol <code>__AT94Kxx__</code> . For example, the symbol is <code>__AT90S2313__</code> when the <code>--cpu=2313</code> option is used, and <code>__ATmega163__</code> when <code>--cpu=m163</code> is used.
<code>__embedded_cplusplus</code>	This predefined symbol expands to the number 1 when the compiler runs in any of the C++ modes. When the compiler runs in ISO/ANSI C mode, the symbol is undefined.
<code>__FILE__</code>	Use this symbol to identify which file is currently being compiled. This symbol expands to the name of that file. See also, <code>__BASE_FILE__</code> , page 229.
<code>__func__</code> , <code>__FUNCTION__</code>	Use one of these symbols inside a function body to make it expand into a string with the function name as context. This is useful for assertions and other trace utilities. These symbols require that language extensions are enabled, see <code>-e</code> , page 179. See also, <code>__PRETTY_FUNCTION__</code> , page 233.
<code>__HAS_EEPROM__</code>	This symbol 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>	This symbol 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.

`__HAS_ELPM__` This symbol determines whether the instruction `ELPM` is available or not.
When this symbol is defined, the instruction `ELPM` is available. When this symbol is not defined, the `ELPM` instruction is not available.

`__HAS_ENHANCED_CORE__` This symbol determines whether the enhanced core is available or not.
When this symbol is defined, the enhanced core is available. When this symbol is not defined, the enhanced core is not available.

`__HAS_FISCR__` This symbol determines whether the instruction `FISCR` is available or not.
When this symbol is defined, the instruction `FISCR` is available. When this symbol is not defined, the `FISCR` instruction is not available.

`__HAS_MUL__` This symbol determines whether the instruction `MUL` is available or not.
When this symbol is defined, the instruction `MUL` is available. When this symbol is not defined, the `MUL` instruction is not available.

`__HAS_RAMPD__` This symbol determines whether the register `RAMPD` is available or not.
When this symbol is defined, the register `RAMPD` is available. When this symbol is not defined, the `RAMPD` register is not available.

`__HAS_RAMPX__` This symbol determines whether the register `RAMPX` is available or not.
When this symbol is defined, the register `RAMPX` is available. When this symbol is not defined, the `RAMPX` register is not available.

`__HAS_RAMPY__` This symbol determines whether the register `RAMPY` is available or not.
When this symbol is defined, the register `RAMPY` is available. When this symbol is not defined, the `RAMPY` register is not available.

`__HAS_RAMPZ__` This symbol determines whether the register `RAMPZ` is available or not.
When this symbol is defined, the register `RAMPZ` is available. When this symbol is not defined, the `RAMPZ` register is not available.

`__IAR_SYSTEMS_ICC__` This predefined symbol expands to a number that identifies the IAR compiler platform. The current identifier is 6. Note that the number could be higher in a future version of the product.

This symbol can be tested with `#ifdef` to detect whether the code was compiled by a compiler from IAR Systems.

`__ICCAVR__` This predefined symbol expands to the number 1 when the code is compiled with the AVR IAR C/C++ Compiler.

`__LINE__` This predefined symbol expands to the current line number of the file currently being compiled.

`__MEMORY_MODEL__` Use this symbol to identify the used memory model.

This symbol expands to a value reflecting the selected memory model according to the following table:

Value	Memory model
1	Tiny
2	Small
3	Large

Table 71: Predefined memory model symbol values

`NDEBUG` This preprocessor symbol determines whether any assert code you have written in your application shall be included or not in the built application.

If the symbol is:

- **defined**, the assert code will *not* be included
- **not defined**, the assert code will be included

This means that if you have written any assert code and build your application, you should define this symbol to exclude the assert code from the application.

Note that the assert macro is defined in the `assert.h` standard include file.



In IAR Embedded Workbench, the `NDEBUG` symbol is automatically defined if you build your application in the Release build configuration.

`_Pragma()` The preprocessor operator `_Pragma` can be used in defines and has the equivalent effect of the `#pragma` directive. The syntax is:

```
_Pragma("string")
```

where *string* follows the syntax for the corresponding pragma directive. For example:

```
#if NO_OPTIMIZE
    #define NOOPT _Pragma("optimize=2")
#else
    #define NOOPT
#endif
```

See the chapter *Pragma directives*.

Note: The `-e` option—enable language extensions—is not required.

`__PRETTY_FUNCTION__` Use this symbol inside a function body to make it expand into a string, with the function name including parameter types and return type as context. The result might, for example, look like this:

```
"void func(char) "
```

This symbol is useful for assertions and other trace utilities. These symbols require that language extensions are enabled, see `-e`, page 179.

See also, `__func__`, `__FUNCTION__`, page 230.

`__STDC__` This predefined symbol expands to the number 1. This symbol can be tested with `#ifdef` to detect whether the compiler in use adheres to ISO/ANSI C.

`__STDC_VERSION__` ISO/ANSI C and version identifier.

This predefined symbol expands to 199409L.

Note: This predefined symbol does not apply in EC++ mode.

`__TID__` Target identifier for the AVR IAR C/C++ Compiler.

Expands to the target identifier which contains the following parts:

- A target identifier (t) unique for each IAR compiler. For the AVR microcontroller, the target identifier is 90.
- The value (c) of the `--cpu` or `-v` option.

- 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 corresponds to Large.

The `__TID__` value is constructed as:

```
((t << 8) | (c << 4) | m)
```

You can extract the values as follows:

```
t = (__TID__ >> 8) & 0x7F;    /* target identifier */
c = (__TID__ >> 4) & 0x0F;    /* cpu core */
m = __TID__ & 0x0F;          /* memory model */
```

To find the value of the target identifier for the current compiler, execute:

```
printf("%ld", (__TID__ >> 8) & 0x7F)
```

Note: The use of `__TID__` is not recommended. We recommend that you use the symbols `__ICCAVR__` and `__CORE__` instead.

`__TIME__` Current time.

Expands to the time of compilation in the form `hh:mm:ss`.

`__VER__` Compiler version number.

Expands to an integer representing the version number of the compiler. The value of the number is calculated in the following way:

```
(100 * the major version number + the minor version number)
```

Example

The example below prints a message for version 3.34.

```
#if __VER__ == 334
#pragma message("Compiler version 3.34")
#endif
```

In this example, 3 is the major version number and 34 is the minor version number.

`__VERSION_1_CALLS__` Calling convention.

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 detailed information about the calling conventions, see *Calling convention*, page 99.

Preprocessor extensions

The following section gives reference information about the extensions that are available in addition to the pragma directives and ISO/ANSI directives.

`#warning message` Use this preprocessor directive to produce messages. Typically this is useful for assertions and other trace utilities, similar to the way the ISO/ANSI standard `#error` directive is used. The syntax is:

```
#warning message
```

where *message* can be any string.

`__VA_ARGS__` Variadic macros are the preprocessor macro equivalents of `printf` style functions.

Syntax

```
#define P(...)      __VA_ARGS__
#define P(x,y,...)  x + y + __VA_ARGS__
```

Here, `__VA_ARGS__` will contain all variadic arguments concatenated, including the separating commas.

Example

```
#if DEBUG
    #define DEBUG_TRACE(...) printf(S,__VA_ARGS__)
#else
    #define DEBUG_TRACE(...)
#endif
...
DEBUG_TRACE("The value is:%d\n",value);
```

will result in:

```
printf("The value is:%d\n",value);
```


Intrinsic functions

This chapter gives reference information about the 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.

Intrinsic functions summary

The following table summarizes the intrinsic functions:

Intrinsic function	Description
<code>__delay_cycles</code>	Inserts a time delay
<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 FMULS instruction
<code>__fractional_multiply_signed_with_unsigned</code>	Generates an FMULSU instruction
<code>__fractional_multiply_unsigned</code>	Generates an FMUL instruction
<code>__indirect_jump_to</code>	Generates an IJMP instruction
<code>__insert_opcode</code>	Assigns a value to a processor register
<code>__load_program_memory</code>	Returns one byte from code memory
<code>__multiply_signed</code>	Generates a MULS instruction
<code>__multiply_signed_with_unsigned</code>	Generates a MULSU instruction
<code>__multiply_unsigned</code>	Generates a MUL instruction
<code>__no_operation</code>	Generates a NOP instruction
<code>__require</code>	Sets a constant literal
<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>__segment_begin</code>	Returns the start address of a segment

Table 72: Intrinsic functions summary

Intrinsic function	Description
<code>__segment_end</code>	Returns the end address of a segment
<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

Table 72: Intrinsic functions summary (Continued)

To use intrinsic functions in an application, include the header file `intrinsic.h`.

Note that the intrinsic function names start with double underscores, for example:

`__segment_begin`

Descriptions of intrinsic functions

The following section gives reference information about each intrinsic function.

`__delay_cycles` `__delay_cycles(unsigned long int);`

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.

Note: The specified value must be a constant integer expression and not an expression that is evaluated at runtime.

`__disable_interrupt` `void __disable_interrupt(void);`

Disables interrupts by inserting the `CLI` instruction.

`__enable_interrupt` `void __enable_interrupt(void);`

Enables interrupts by inserting the `SEI` instruction.

`__extended_load_program_memory` `unsigned char __extended_load_program_memory(unsigned char __farflash *);`

Returns one byte from code memory.

Use this intrinsic function to access constant data in code memory.

<code>__fractional_multiply_signed</code>	<pre>signed int __fractional_multiply_signed(signed char, signed char);</pre>	<p>Generates a FMULS instruction.</p>
<code>__fractional_multiply_signed_with_unsigned</code>	<pre>signed int __fractional_multiply_signed_with_unsigned(signed char, unsigned char);</pre>	<p>Generates a FMULSU instruction.</p>
<code>__fractional_multiply_unsigned</code>	<pre>unsigned int __fractional_multiply_unsigned(unsigned char, unsigned char);</pre>	<p>Generates a FMUL instruction.</p>
<code>__indirect_jump_to</code>	<pre>void __indirect_jump_to(unsigned long);</pre>	<p>Jumps to the address specified by the argument by using the IJMP or EIJMP instruction, depending on the generic processor option. -v0 to -v4 use IJMP, and -v5 and -v6 use EIJMP.</p>
<code>__insert_opcode</code>	<pre>void __insert_opcode(unsigned short);</pre>	<p>Inserts a DW unsigned directive.</p>
<code>__load_program_memory</code>	<pre>unsigned char __load_program_memory(unsigned char __flash *);</pre>	<p>Returns one byte from code memory. The constants must be placed within the first 64 Kbytes of memory.</p>
<code>__multiply_signed</code>	<pre>signed int __multiply_signed(signed char, signed char);</pre>	<p>Generates a MULS instruction.</p>
<code>__multiply_signed_with_unsigned</code>	<pre>signed int __multiply_signed_with_unsigned(signed char, unsigned char);</pre>	<p>Generates a MULSU instruction.</p>

```
__multiply_unsigned unsigned int __multiply_unsigned(unsigned char, unsigned char);
```

Generates a MUL instruction.

```
__no_operation void __no_operation(void);
```

Generates a NOP instruction.

```
__require void __require(void *);
```

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 the following example, the copyright message will be included in the generated binary file even though it is not directly used.

```
#include <intrinsic.h>
char copyright[] = "Copyright 2005 by XXXX";
void main(void)
{
    __require(copyright);
    [... the rest of the program ...]
}
```

```
__restore_interrupt void __restore_interrupt(unsigned char oldState);
```

This intrinsic function will restore 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 unsigned int __reverse(unsigned int);
```

This intrinsic function reverses the byte order of the value given as parameter. Avoid using `__reverse` in complex expressions as it might introduce extra register copying.

```

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 > 64k flash memory */
void __hugeflash * __reverse(void __hugeflash *);
/* Only on -v2 through -v6 with > 64k flash memory */

```

```
__save_interrupt unsigned char __save_interrupt(void);
```

This intrinsic function will save 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 `__restore_interrupt` intrinsic function.

Example

```

unsigned char oldState;

oldState = __save_interrupt();
__disable_interrupt();

/* Critical section goes here */

__restore_interrupt(oldState);

```

```
__segment_begin void * __segment_begin(segment);
```

Returns the address of the first byte of the named *segment*. The named *segment* must be a string literal that has been declared earlier with the `#pragma segment` directive. See *#pragma segment*, page 224.

If the segment was declared with a memory attribute *memattr*, the type of the `__segment_begin` function is pointer to *memattr* void. Otherwise, the type is a default pointer to void.

Example

```

#pragma segment="MYSEG" __huge
...
segment_start_address = __segment_begin("MYSEG");

```

Here, the type of the `__segment_begin` intrinsic function is `void __huge *`.

Note: You must have enabled language extensions to use this intrinsic function.

```
__segment_end void * __segment_end(segment);
```

Returns the address of the first byte *after* the named *segment*. The named *segment* must be a string literal that has been declared earlier with the `#pragma segment` directive. See *#pragma segment*, page 224.

If the segment was declared with a memory attribute *memattr*, the type of the `__segment_end` function is pointer to *memattr* void. Otherwise, the type is a default pointer to void.

Example

```
#pragma segment="MYSEG" __huge
...
segment_end_address = __segment_end("MYSEG");
```

Here, the type of the `__segment_end` intrinsic function is `void __huge *`.

Note: You must have enabled language extensions to use this intrinsic function.

```
__sleep void __sleep(void);
```

Inserts a sleep instruction, `SLEEP`. To use this intrinsic function, make sure that the instruction has been enabled in the `MCUCR` register.

```
__swap_nibbles unsigned char __swap_nibbles(unsigned char);
```

Swaps bit 0-3 with bit 4-7 of the parameter and returns the swapped value.

```
__watchdog_reset void __watchdog_reset(void);
```

Inserts a watchdog reset instruction.

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.

Introduction

The AVR IAR C/C++ Compiler provides two different libraries:

- IAR DLIB Library is a complete ISO/ANSI C and C++ library. This library also supports floating-point numbers in IEEE 754 format and it can be configured to include different levels of support for locale, file descriptors, multibytes, et cetera.
- IAR CLIB Library is a light-weight library, which is not fully compliant with ISO/ANSI C. Neither does it fully support floating-point numbers in IEEE 754 format or does it support Embedded C++.

Note that different customization methods are normally needed for these two libraries. For additional 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 additional information about library functions, see the chapter *Implementation-defined behavior* in this guide.

HEADER FILES

Your application program gains access to library definitions through header files, which it incorporates using the `#include` directive. The definitions are divided into a number of different header files, each covering a particular functional area, letting you include just those that are required.

It is essential to include the appropriate header file before making any reference to its definitions. Failure to do so can cause the call to fail during execution, or generate error or warning messages at compile time or link time.

LIBRARY OBJECT FILES

Most of the library definitions can be used without modification, that is, directly from the library object files that are supplied with the product. For information about how to choose a runtime library, see *Basic settings for project configuration*, page 5. The linker will include only those routines that are required—directly or indirectly—by your application.

REENTRANCY

A function that can be simultaneously invoked in the main application and in any number of interrupts is reentrant. A library function that uses statically allocated data is therefore not reentrant. Most parts of the DLIB library are reentrant, but the following functions and parts are not reentrant:

<code>atexit</code>	Needs static data
heap functions	Need static data for memory allocation tables
<code>strerror</code>	Needs static data
<code>strtok</code>	Designed by ISO/ANSI standard to need static data
I/O	Every function that uses files in some way. This includes <code>printf</code> , <code>scanf</code> , <code>getchar</code> , and <code>putchar</code> . The functions <code>sprintf</code> and <code>sscanf</code> are not included.

In addition, some functions share the same storage for `errno`. These functions are not reentrant, since an `errno` value resulting from one of these functions can be destroyed by a subsequent use of the function before it has been read. Among these functions are:

```
exp, exp10, ldexp, log, log10, pow, sqrt, acos, asin, atan2,
cosh, sinh, strtod, strtol, strtoul
```

Remedies for this are:

- Do not use non-reentrant functions in interrupt service routines
- Guard calls to a non-reentrant function by a mutex, or a secure region, etc.

IAR DLIB Library

The IAR DLIB Library provides most of the important C and C++ library definitions that apply to embedded systems. These are of the following types:

- Adherence to a free-standing implementation of the ISO/ANSI standard for the programming language C. For additional information, see the chapter *Implementation-defined behavior* in this guide.

- Standard C library definitions, for user programs.
- Embedded C++ library definitions, for user programs.
- `CSTARTUP`, the module containing the start-up code. It is described in the chapter *The DLIB runtime environment* in this guide.
- Runtime support libraries; for example low-level floating-point routines.
- Intrinsic functions, allowing low-level use of 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 249.

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 *IAR language extensions*.

The following table lists the C header files:

Header file	Usage
<code>assert.h</code>	Enforcing assertions when functions execute
<code>ctype.h</code>	Classifying characters
<code>errno.h</code>	Testing error codes reported by library functions
<code>float.h</code>	Testing floating-point type properties
<code>iso646.h</code>	Using Amendment 1— <code>iso646.h</code> standard header
<code>limits.h</code>	Testing integer type properties
<code>locale.h</code>	Adapting to different cultural conventions
<code>math.h</code>	Computing common mathematical functions
<code>setjmp.h</code>	Executing non-local <code>goto</code> statements
<code>signal.h</code>	Controlling various exceptional conditions
<code>stdarg.h</code>	Accessing a varying number of arguments
<code>stdbool.h</code>	Adds support for the <code>bool</code> data type in C.
<code>stddef.h</code>	Defining several useful types and macros
<code>stdio.h</code>	Performing input and output
<code>stdlib.h</code>	Performing a variety of operations
<code>string.h</code>	Manipulating several kinds of strings
<code>time.h</code>	Converting between various time and date formats
<code>wchar.h</code>	Support for wide characters
<code>wctype.h</code>	Classifying wide characters

Table 73: Traditional standard C header files—DLIB

C++ HEADER FILES

This section lists the C++ header files.

Embedded C++

The following table lists the Embedded C++ header files:

Header file	Usage
<code>complex</code>	Defining a class that supports complex arithmetic
<code>exception</code>	Defining several functions that control exception handling
<code>fstream</code>	Defining several I/O streams classes that manipulate external files
<code>iomanip</code>	Declaring several I/O streams 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 streams classes before they are necessarily defined
<code>iostream</code>	Declaring the I/O streams 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 streams classes that manipulate string containers
<code>stdexcept</code>	Defining several classes useful for reporting exceptions
<code>streambuf</code>	Defining classes that buffer I/O streams operations
<code>string</code>	Defining a class that implements a string container
<code>stringstream</code>	Defining several I/O streams classes that manipulate in-memory character sequences

Table 74: Embedded C++ header files

The following table lists additional C++ header files:

Header file	Usage
<code>fstream.h</code>	Defining several I/O streams classes that manipulate external files
<code>iomanip.h</code>	Declaring several I/O streams manipulators that take an argument
<code>iostream.h</code>	Declaring the I/O streams objects that manipulate the standard streams
<code>new.h</code>	Declaring several functions that allocate and free storage

Table 75: Additional Embedded C++ header files—DLIB

Extended Embedded C++ standard template library

The following table lists the Extended Embedded C++ 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
<code>queue</code>	A queue sequence container
<code>set</code>	A set associative container
<code>slist</code>	A singly-linked list sequence container
<code>stack</code>	A stack sequence container
<code>utility</code>	Defines several utility components
<code>vector</code>	A vector sequence container

Table 76: Standard template library header files

Using standard C libraries in C++

The C++ library works in conjunction with 15 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`.

The following table shows the new header files:

Header file	Usage
<code>cassert</code>	Enforcing assertions when functions execute
<code>cctype</code>	Classifying characters
<code>cerrno</code>	Testing error codes reported by library functions
<code>cfloat</code>	Testing floating-point type properties
<code>climits</code>	Testing integer type properties

Table 77: New standard C header files—DLIB

Header file	Usage
<code>locale</code>	Adapting to different cultural conventions
<code>cmath</code>	Computing common mathematical functions
<code>csetjmp</code>	Executing non-local goto statements
<code>csignal</code>	Controlling various exceptional conditions
<code>cstdarg</code>	Accessing a varying number of arguments
<code>cstddef</code>	Defining several useful types and macros
<code>stdio</code>	Performing input and output
<code>stdlib</code>	Performing a variety of operations
<code>string</code>	Manipulating several kinds of strings
<code>time</code>	Converting between various time and date formats

Table 77: New standard C header files—DLIB (Continued)

LIBRARY FUNCTIONS AS INTRINSIC FUNCTIONS

The following C library functions will under some circumstances be handled as intrinsic functions and will generate inline code instead of an ordinary function call:

```
memcpy
memset
strcat
strcmp
strcpy
strlen
```

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. It is described in 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*.

LIBRARY DEFINITIONS SUMMARY

This section lists the header files. Header files may additionally contain target-specific definitions.

Header file	Description
<code>assert.h</code>	Assertions
<code>ctype.h*</code>	Character handling
<code>iccbutl.h</code>	Low-level routines
<code>math.h</code>	Mathematics
<code>setjmp.h</code>	Non-local jumps
<code>stdarg.h</code>	Variable arguments
<code>stdio.h</code>	Input/output
<code>stdlib.h</code>	General utilities
<code>string.h</code>	String handling

Table 78: 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.**

The following table shows header files that do not contain any functions, but specify various definitions and data types:

Header file	Description
<code>errno.h</code>	Error return values
<code>float.h</code>	Limits and sizes of floating-point types
<code>limits.h</code>	Limits and sizes of integral types
<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>

Table 79: Miscellaneous IAR CLIB Library header files

AVR-specific library functions

This section lists the AVR-specific library functions defined in `pgmspace.h` that allow access to strings in flash. The `_P` functions exist in both the IAR CLIB Library and the IAR DLIB Library, but they allow access to strings in flash only. The `_G` functions use the `__generic` pointer instead, which means that they allow access to both flash and data memory. The `_G` functions are only available in the IAR DLIB Library.

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 interface 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 int memcpy_G(const void *s1, const void __generic *s2, size_t n);
```

Identical to `memcpy` except that `s2` can be located in flash *or* data memory. This function is only available in the DLIB library.

```
memcpy_G void * memcpy_G(void *s1, const void __generic *s2, size_t n);
```

Identical to `memcpy` except that it copies string `s2` in flash *or* data memory to string `s2` in data memory. This function is only available in the DLIB library.

```
memcpy_P void * memcpy_P(void *s1, PGM_P s2, size_t n);
```

Identical to `memcpy` except that it copies string `s2` in flash memory to string `s2` in data memory. This function is available in both the CLIB and the DLIB library.

```
printf_P int printf_P(PGM_P __format,...);
```

Similar to `printf` 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 *Specifying read and write formatters*, page 250.

```
puts_G int puts_G(const char __generic *s);
```

Identical to `puts` except that the string to be written can be in flash *or* data memory. This function is only available in the DLIB library.

```
puts_P int puts_P(PGM_P __s);
```

Identical to `puts` 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 int scanf_P(PGM_P __format,...);
```

Identical to `scanf` 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 *Specifying read and write formatters*, page 250.

```
sprintf_P int sprintf_P(char *__s, PGM_P __format,...);
```

Identical to `sprintf` 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 int sscanf_P(const char *__s, PGM_P __format,...);
```

Identical to `sscanf` 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 char *strcat_G(char *s1, const char __generic *s2);
```

Identical to `strcat` except that string `s2` can be in flash *or* data memory. This function is only available in the DLIB library.

```
strcmp_G int strcmp_G(const char *s1, const char __generic *s2);
```

Identical to `strcmp` except that string `s2` can be in flash *or* data memory. This function is only available in the DLIB library.

```
strcmp_P int strcmp_P(const char *s1, PGM_P s2);
```

Identical to `strcmp` except that string `s2` is in flash memory, not in data memory. This function is available in both the CLIB and the DLIB library.

```
strcpy_G char *strcpy_G(char *s1, const char __generic *s2);
```

Identical to `strcpy` except that the string `s2` being copied can be in flash *or* data memory. This function is only available in the DLIB library.

```
strcpy_P char * strcpy_P(char *s1, PGM_P s2);
```

Identical to `strcpy` except that the string `s2` being copied is in flash memory, not in data memory. This function is available in both the CLIB and the DLIB library.

```
strerror_P PGM_P strerror_P(int errnum);
```

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 size_t strlen_G(const char __generic *s);
```

Identical to `strlen` except that the string being tested can be in flash *or* data memory. This function is only available in the DLIB library.

```
strlen_P size_t strlen_P(PGM_P s);
```

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 char *strncat_G(char *s1, const char __generic *s2, size_t n);
```

Identical to `strncat` except that the string `s2` can be in flash *or* data memory. This function is only available in the DLIB library.

```
strncmp_G int strncmp_G(const char *s1, const char __generic *s2, size_t n);
```

Identical to `strncmp` except that the string `s2` can be in flash *or* data memory. This function is only available in the DLIB library.

```
strncmp_P int strncmp_P(const char *s1, PGM_P s2, size_t n);
```

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 char *strncpy_G(char *s1, const char __generic *s2, size_t n);
```

Identical to `strncpy` except that the source string `s2` can be in flash *or* data memory. This function is only available in the DLIB library.

```
strncpy_P char * strncpy_P(char *s1, PGM_P s2, size_t n);
```

Identical to `strncpy` except that the source string `s2` is in flash memory, not in data memory. This function is available in both the CLIB and the DLIB library.

Implementation-defined behavior

This chapter describes how the AVR IAR C/C++ Compiler handles the implementation-defined areas of the C language.

ISO 9899:1990, the International Organization for Standardization standard - *Programming Languages - C* (revision and redesign of ANSI X3.159-1989, American National Standard), changed by the ISO Amendment 1:1994, *Technical Corrigendum 1*, and *Technical Corrigendum 2*, contains an appendix called *Portability Issues*. The ISO appendix lists areas of the C language that ISO leaves open to each particular implementation.

Note: The AVR IAR C/C++ Compiler adheres to a freestanding implementation of the ISO standard for the C programming language. This means that parts of a standard library can be excluded in the implementation.

Descriptions of implementation-defined behavior

This section follows the same order as the ISO appendix. Each item covered includes references to the ISO chapter and section (in parenthesis) that explains the implementation-defined behavior.

TRANSLATION

Diagnostics (5.1.1.3)

Diagnostics are produced in the form:

```
filename, linenumber level[tag]: message
```

where *filename* is the name of the source file in which the error was encountered, *linenumber* is the line number at which the compiler detected the error, *level* is the level of seriousness of the message (remark, warning, error, or fatal error), *tag* is a unique tag that identifies the message, and *message* is an explanatory message, possibly several lines.

ENVIRONMENT

Arguments to main (5.1.2.2.1)

The function called at program startup is called `main`. There is no prototype declared for `main`, and the only definition supported for `main` is:

```
int main(void)
```

To change this behavior for the IAR CLIB runtime environment, see *Implementation of `cstartup`*, page 90. To change this behavior for the IAR DLIB runtime environment, see *Implementation of system startup code*, page 81.

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 71.

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 71.

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 138, 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 139, 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 142, for information about *size_t*.

Conversion from/to pointers (6.3.4)

See *Casting*, page 142, for information about casting of data pointers and function pointers.

ptrdiff_t (6.3.6, 7.1.1)

See *ptrdiff_t*, page 143, 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 138, for information about the alignment requirement for data objects.

Sign of 'plain' bitfields (6.5.2.1)

A 'plain' `int` bitfield is treated as a signed `int` bitfield. All integer types are allowed as bitfields.

Allocation order of bitfields within a unit (6.5.2.1)

Bitfields are allocated within an integer from least-significant to most-significant bit.

Can bitfields straddle a storage-unit boundary (6.5.2.1)

Bitfields cannot straddle a storage-unit boundary for the chosen bitfield integer type.

Integer type chosen to represent enumeration types (6.5.2.2)

The chosen integer type for a specific enumeration type depends on the enumeration constants defined for the enumeration type. The chosen integer type is the smallest possible.

QUALIFIERS

Access to volatile objects (6.5.3)

Any reference to an object with volatile qualified type is an access.

DECLARATORS

Maximum numbers of declarators (6.5.4)

The number of declarators is not limited. The number is limited only by the available memory.

STATEMENTS

Maximum number of case statements (6.6.4.2)

The number of case statements (case values) in a switch statement is not limited. The number is limited only by the available memory.

PREPROCESSING DIRECTIVES

Character constants and conditional inclusion (6.8.1)

The character set used in the preprocessor directives is the same as the execution character set. The preprocessor recognizes negative character values if a 'plain' character is treated as a `signed` character.

Including bracketed filenames (6.8.2)

For file specifications enclosed in angle brackets, the preprocessor does not search directories of the parent files. A parent file is the file that contains the `#include` directive. Instead, it begins by searching for the file in the directories specified on the compiler command line.

Including quoted filenames (6.8.2)

For file specifications enclosed in quotes, the preprocessor directory search begins with the directories of the parent file, then proceeds through the directories of any grandparent files. Thus, searching begins relative to the directory containing the source file currently being processed. If there is no grandparent file and the file has not been found, the search continues as if the filename was enclosed in angle brackets.

Character sequences (6.8.2)

Preprocessor directives use the source character set, with the exception of escape sequences. Thus, to specify a path for an include file, use only one backslash:

```
#include "mydirectory\myfile"
```

Within source code, two backslashes are necessary:

```
file = fopen("mydirectory\\myfile", "rt");
```

Recognized pragma directives (6.8.6)

The following pragma directives are recognized:

```
alignment
ARGUSED
baseaddr
bitfields
can_instantiate
codeseg
constseg
cspy_support
data_alignment
dataseg
define_type_info
diag_default
diag_error
diag_remark
diag_suppress
diag_warning
do_not_instantiate
function
hdrstop
```

```

include_alias
inline
instantiate
language
location
memory
message
module_name
none
no_pch
NOTREACHED
object_attribute
once
optimize
pack
__printf_args
public_equ
required
rtmodel
__scanf_args
segment
system_include
type_attribute
VARARGS
vector
warnings

```

For a description of the pragma directives, see the chapter *Pragma directives*.

Default `__DATE__` and `__TIME__` (6.8.8)

The definitions for `__TIME__` and `__DATE__` are always available.

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 80: 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.

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 73.

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 70.

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 70.

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 70.

%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 73.

system() (7.10.4.5)

How the command processor works depends on how you have implemented the `system` function. See *Environment interaction*, page 73.

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 81: 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 74.

clock() (7.12.2.1)

From where the system clock starts counting depends on how you have implemented the `clock` function. See *Time*, page 74.

IAR language extensions

This chapter describes IAR language extensions to the ISO/ANSI standard for the C programming language. All extensions can also be used for the C++ programming language.



In the IAR Embedded Workbench™ IDE, language extensions are enabled by default.



See the compiler options `-e` on page 179 and `--strict_ansi` on page 196 for information about how to enable and disable language extensions from the command line.

Why should language extensions be used?

By using language extensions, you gain full control over the resources and features of the target microcontroller, and can thereby fine-tune your application.

If you want to use the source code with different compilers, note that language extensions may require minor modifications before the code can be compiled. A compiler typically supports microcontroller-specific language extensions as well as vendor-specific ones.

Descriptions of language extensions

This section gives an overview of available language extensions.

Memory, type, and object attributes

Entities such as variables and functions may be declared with memory, type, and object attributes. The syntax follows the syntax for qualifiers—such as `const`—but the semantics is different.

- A memory attribute controls the placement of the entity. There can be only one memory attribute.
- A type attribute controls aspects of the object visible to the surrounding context. There can be many different type attributes, and they must be included when the object is declared.
- An object attribute only has to be specified at the definition, but not at the declaration, of an object. The object attribute does not affect the object interface.

See the chapter *Extended keywords* for a complete list of attributes.

Placement at an absolute address or in a named segment

The operator @ or the directive `#pragma location` can be used for placing a variable at an absolute address, or placing a variable or function in a named segment. The named segment can either be a predefined segment, or a user-defined segment.

Example 1

```
__no_init int x @ 0x1000;
```

An absolute declared variable cannot have an initializer, which means the variable must also be `__no_init` or `const` declared.

Example 2

```
void test(void) @ "MYOWNSEGMENT"
{
    ...
}
```

Note that all segments, both user-defined and predefined, must be assigned a location, which is done in the linker command file.

Pragma

For information about the preprocessor operator `_Pragma`, see *_Pragma()*, page 233.

Variadic macros

Variadic macros are the preprocessor macro equivalents of `printf` style functions. For more information, see `__VA_ARGS__`, page 235.

Inline functions

The `inline` keyword can be used on functions. It works just like the C++ keyword `inline` and the `#pragma inline` directive.

Mixing declarations and statements

It is possible to mix declarations and statements within the same scope.

Declaration in for loops

It is possible to have a declaration in the initialization expression of a `for` loop.

Inline assembler

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

The syntax for inline assembler is:

```
asm( "MOVW R4, R7" );
```

In strict ISO/ANSI mode, the use of inline assembler is disabled.

For more details about inline assembler, see *Mixing C and assembler*, page 93.

C++ style comments

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

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

__ALIGNOF__

For information about alignment, see *Alignment*, page 137, and `__ALIGNOF__()`, page 229.

Compound literals

To create compound literals you can use the following syntax:

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

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

Note:

- A compound literal can be modified unless it is declared `const`
- Compound literals are not supported in Embedded C++.

Anonymous structs and unions

C++ includes a feature named anonymous unions. The AVR IAR C/C++ Compiler allow a similar feature for both structs and unions.

An anonymous structure type (that is, one without a name) defines an unnamed object (and not a type) whose members are promoted to the surrounding scope. External anonymous structure types are allowed.

For example, the structure `str` in the following example contains an anonymous union. The members of the union are accessed using the names `b` and `c`, for example `obj.b`.

Without anonymous structure types, the union would have to be named—for example `u`—and the member elements accessed using the syntax `obj.u.b`.

```
struct str
{
    int a;
    union
    {
        int b;
        int c;
    };
};

struct str obj;
```

Bitfields and non-standard types

In ISO/ANSI C, a bitfield must be of the type `int` or `unsigned int`. Using IAR language extensions, any integer types and enums may be used.

For example, in the following structure an `unsigned char` is used for holding three bits. The advantage is that the struct will be smaller.

```
struct str
{
    unsigned char bitOne   : 1;
    unsigned char bitTwo   : 1;
    unsigned char bitThree : 1;
};
```

This matches G.5.8 in the appendix of the ISO standard, *ISO Portability Issues*.

Incomplete arrays at end of structs

The last element of a `struct` may be an incomplete array. This is useful because one chunk of memory can be allocated for the `struct` itself and for the array, regardless of the size of the array.

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

Example

```

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

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

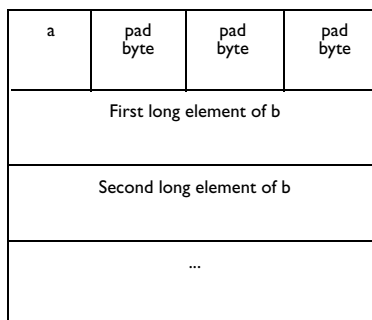
void UseStr(struct str * s)
{
    s->b[10] = 0;
}

```

The `struct` will inherit the alignment requirements from all elements, including the alignment of the incomplete array. The array itself will not be included in the size of the struct. However, the alignment requirements will ensure that the struct will end exactly at the beginning of the array; this is known as padding.

In the example, the alignment of `struct str` will be 4 and the size is also 4. (Assuming a processor where the alignment of `unsigned long` is 4.)

The memory layout of `struct str` is described in the following figure.

**Arrays of incomplete types**

An array may have an incomplete `struct`, `union`, or `enum` type as its element type. The types must be completed before the array is used (if it is), or by the end of the compilation unit (if it is not).

Empty translation units

A translation unit (source file) is allowed to be empty, that is, it does not have to contain any declarations.

In strict ISO/ANSI mode, a warning is issued if the translation unit is empty.

Example

The following source file is only used in a debug build. (In a debug build, the `NDEBUG` preprocessor flag is undefined.) Since the entire contents of the file is conditionally compiled using the preprocessor, the translation unit will be empty when the application is compiled in release mode. Without this extension, this would be considered an error.

```
#ifndef NDEBUG

void PrintStatusToTerminal()
{
    /* Do something */
}

#endif
```

Comments at the end of preprocessor directives

This extension, which makes it legal to place text after preprocessor directives, is enabled, unless strict ISO/ANSI mode is used. This language extension exists to support compilation of old legacy code; we do *not* recommend that you write new code in this fashion.

Example

```
#ifdef FOO

    ... something ...

#endif FOO /* This is allowed but not recommended. */
```

Forward declaration of enums

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

Extra comma at end of enum list

It is allowed to place an extra comma at the end of an `enum` list. In strict ISO/ANSI mode, a warning is issued.

Note: ISO/ANSI C allows extra commas in similar situations, for example after the last element of the initializers to an array. The reason is, that it is easy to get the commas wrong if parts of the list are moved using a normal cut-and-paste operation.

Example

```
enum
{
    kOne,
    kTwo,    /* This is now allowed. */
};
```

Missing semicolon at end of struct or union specifier

A warning is issued if the semicolon at the end of a `struct` or union specifier is missing.

NULL and void

In operations on pointers, a pointer to `void` is always implicitly converted to another type if necessary, and a null pointer constant is always implicitly converted to a null pointer of the right type if necessary. In ISO/ANSI C, some operators allow such things, while others do not allow them.

A label preceding a "}"

In ISO/ANSI C, a label must be followed by at least one statement. Therefore, it is illegal to place the label at the end of a block. In the AVR IAR C/C++ Compiler, a warning is issued.

To create a standard-compliant C program (so that you will not have to see the warning), you can place an empty statement after the label. An empty statement is a single `;` (semi-colon).

Example

```
void test()
{
    if (...) goto end;

    /* Do something */

    end: /* Illegal at the end of block. */
}
```

Note: This also applies to the labels of switch statements.

The following piece of code will generate a warning:

```
switch (x)
{
case 1:
    ...;
    break;

default:
}
```

A good way to convert this into a standard-compliant C program is to place a `break;` statement after the `default:` label.

Empty declarations

An empty declaration (a semicolon by itself) is allowed, but a remark is issued (provided that remarks are enabled).

This is useful when preprocessor macros are used that could expand to nothing. Consider the following example. In a debug build, the macros `DEBUG_ENTER` and `DEBUG_LEAVE` could be defined to something useful. However, in a release build they could expand into nothing, leaving the `;` character in the code:

```
void test()
{
    DEBUG_ENTER();

    do_something();

    DEBUG_LEAVE();
}
```

Single value initialization

ISO/ANSI C requires that all initializer expressions of static arrays, `structs`, and `unions` are enclosed in braces.

Single-value initializers are allowed to appear without braces, but a warning is issued.

Example

In the AVR IAR C/C++ Compiler, the following expression is allowed:

```
struct str
{
    int a;
} x = 10;
```

Casting pointers to integers in static initializers

In an initializer, a pointer constant value may be cast to an integral type if the integral type is large enough to contain it.

In the following example, it is assumed that pointers to `__near` and `__huge` are 16 and 32 bits, respectively. The first initialization is correct, because it is possible to cast the 16-bit address to a 16-bit `unsigned short` variable. However, it is illegal to use the 32-bit address of `b` as an initializer for a 16-bit value.

```
__near int a;
__huge int b;

unsigned short ap = (unsigned short)&a; /* Correct */
unsigned short bp = (unsigned short)&b; /* Error */
```

Hexadecimal floating-point constants

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

Examples

```
0x1p0 is 1
0xA.8p2 is 10.5*2^2
```

Using the bool data type in C

To use the `bool` type in C source code, you must include the file `stdbool.h`. (The `bool` data type is supported by default in C++.)

Taking the address of a register variable

In ISO/ANSI C, it is illegal to take the address of a variable specified as a register variable.

The AVR IAR C/C++ Compiler allows this, but a warning is issued.

Duplicated size and sign specifiers

Should the size or sign specifiers be duplicated (for example, `short short` or `unsigned unsigned`), an error is issued.

"long float" means "double"

`long float` is accepted as a synonym for `double`.

Repeated typedefs

Redeclarations of `typedef` that occur in the same scope are allowed, but a warning is issued.

Mixing pointer types

Assignment and pointer difference is allowed between pointers to types that are interchangeable but not identical; for example, `unsigned char *` and `char *`. This includes pointers to integral types of the same size. A warning is issued.

Assignment of a string constant to a pointer to any kind of character is allowed, and no warning will be issued.

Non-top level const

Assignment of pointers is allowed in cases where the destination type has added type qualifiers that are not at the top level (for example, `int **` to `int const **`). It is also allowed to compare and take the difference of such pointers.

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;
}
```

Non-lvalue arrays

A non-lvalue array expression is converted to a pointer to the first element of the array when it is used.

Diagnostics

This chapter 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
```

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 diagnostic, *tag* is a unique tag that identifies the diagnostic message, and *message* is a self-explanatory message, 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 in a named file.

Severity levels

The diagnostics 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 194.

Warning

A diagnostic that is produced when the compiler finds a programming error or omission which is of concern, but not so severe as to prevent the completion of compilation. Warnings can be disabled by use of the command-line option `--no_warnings`, see page 191.

Error

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

SETTING THE SEVERITY LEVEL

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

See *Options summary*, page 169, for a description of the compiler options that are available for setting severity levels.

See the chapter *Pragma directives*, for a description of the pragma directives that are available for setting severity levels.

INTERNAL ERROR

An internal error is a diagnostic message that signals that there has been a serious and unexpected failure due to a fault in the compiler. It is produced using the following form:

Internal error: *message*

where *message* is an explanatory message. If internal errors occur, they should be reported to your software distributor or IAR Technical Support. Include information enough 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 program that generated the internal error
- A list of the options that were used when the internal error occurred.

A

absolute location	47
#pragma location	220
absolute placement	270
aggregate initializers, placing in flash memory	184
algorithm (STL header file)	247
alignment	137
forcing stricter	217
__ALIGNOF__() (predefined symbol)	229
anonymous structures	128
anonymous symbols, creating	271
applications	
building	4
initializing	65, 89
terminating	65, 90
architecture, AVR	xix
ARGFRAME (compiler function directive)	32
arrays	
hints	126
implementation-defined behavior	259
__asm (extended keyword)	205
asm (extended keyword)	205
assembler directives	
CFI	108
ENDMOD	81
EQU	194
MODULE	81
PUBLIC	194
REQUIRE	81
RSEG	81
RTMODEL	78
assembler instructions	
CLI	238
NOP	240
SEI	238
assembler labels	
?C_EXIT	91
?C_GETCHAR	91

?C_PUTCHAR	91
assembler language interface	93
creating skeleton code	96
assembler list file	32
assembler routines, calling from C	96
assembler, inline	95, 130, 271
assert	75, 232
assert.h (library header file)	245, 249
assumptions (programming experience)	xix
atomic operations, performing	210
attributes	144
auto variables	23
AVR	
architecture	xix
instruction set	xix
AVR derivatives	
mapping of processor options	6
specifying	173
supported	6
AVR, memory access	9

B

__BASE_FILE__ (predefined symbol)	229
basic_template_matching (pragma directive)	216
using	117
bitfields	
data representation	139
hints	126
implementation-defined behavior	259
bitfields (pragma directive)	139, 217
bool (data type)	138
adding support for in CLIB	249
adding support for in DLIB	245
making available in C	83
supported in C code	138
bubble sort algorithm, adding support for	83
byte order (of value), reversing	240

C

C and C++ linkage	100
C calling convention	99
C header files	245
call chains	131
call stack	108
callee-save registers, stored on stack	23
calling convention	
C	99
C++	98
--version1_calls (compiler option)	199
calloc (standard library function)	25
cassert (library header file)	247
cast operators	
in Extended EC++	110
missing from Embedded C++	110
casting, of pointers and integers	142
cctype (library header file)	247
cerrno (library header file)	247
CFI (assembler directive)	108
cfloat (library header file)	247
char (data type), signed and unsigned	139, 173, 187
characters, implementation-defined behavior	256
--char_is_signed (compiler option)	173
class memory (extended EC++)	112
class template partial specialization matching (extended EC++)	116
classes	111
CLI (assembler instruction)	238
CLIB	11, 248
documentation	243
header files	243
climits (library header file)	247
locale (library header file)	248
__close (library function)	70
Clustering of variables (compiler option)	125
clustering, disabling	189
cmath (library header file)	248

code	
excluding when linking	81
placement of	149
Code motion (compiler option)	124
code motion, disabling	189
CODE (segment)	151
Common subexpr elimination (compiler option)	123
common sub-expression elimination, disabling	190
compiler environment variables	169
compiler error return codes	169
compiler listing, generating	185
compiler object file	
excluding UBROF messages	191
including debug information	175, 194
specifying filename	192
compiler options	
Clustering of variables	125
Code motion	124
Common subexpr elimination	123
Cross call	125
Function inlining	123
setting	167
specifying parameters	168
summary	169
Type-based alias analysis	124
typographic convention	xxii
-D	174
-e	179
-f	182
-I	183
-l	97, 185
-m	186
-o	192
-r	175, 194
-s	195
-v	197
mapping of AVR derivatives	6
-y	200
locating __far variables	155

- locating `__tiny` variables 165
- z 200
- char_is_signed 173
- cpu 173
 - mapping of AVR derivatives 6
- cross_call_passes 174
- debug 175, 194
- dependencies 175
- diagnostics_tables 178
- diag_error 176
- diag_remark 177
- diag_suppress 177
- diag_warning 177
- disable_direct_mode 178
- dlib_config 178
- do_cross_call 179
- ec++ 179
- eegr_address 180
- ecc++ 180
- eeprom_size 180
- enable_external_bus 181
- enable_multibytes 180
- enhanced_core 181
- error_limit 181
- force_switch_type 182
- header_context 183
- initializers_in_flash 184, 197–198
- library_module 186
- lock_reg 186
- memory_model 186
- migration_preprocessor_extensions 187
- misrac 187
- misrac_verbose 188
- module_name 188
- no_clustering 189
- no_code_motion 189
- no_cross_call 189
- no_cse 190
- no_inline 190–191
- no_rampd 190
- no_tbaa 191
- no_ubrof_messages 191
- no_warnings 191
- no_wrap_diagnostics 192
- omit_types 192
- only_stdout 193
- preinclude 193
- preprocess 193
- public_equ 194
- remarks 194
- require_prototypes 194
- root_variables 195
- segment 195
- separate_cluster_for_initialized_variables 196
- silent 196
- spmc_r_address 196
- strict_ansi 196
- string_literals_in_flash 196
- version1_calls 199
- warnings_affect_exit_code 169, 199
- warnings_are_errors 200
- zero_register 201
- 64bit_doubles 201
- 64k_flash 201
- compiler version number 234
- compiling, from the command line 4
- complex numbers, supported in Embedded C++ 110
- complex (library header file) 246
- compound literals 271
- computer style, typographic convention xxii
- configuration
 - basic project settings 5
 - `__low_level_init` 66
- configuration symbols, in library configuration files 63
- consistency, module 77
- constants, placing in initialized data segments 200
- constructor blocks, pointers to 152
- constseg (pragma directive) 217

const_cast (cast operator)	110
conventions, typographic	xxii
copyright notice	ii
__CORE__ (predefined symbol).	229
cores, mapping of processor options	6
__CPU__ (predefined symbol)	229
--cpu (compiler option), mapping of AVR cores	6
__cpu (runtime model attribute).	79
__cpu_name (runtime model attribute)	79
Cross call (compiler option)	125
cross-call optimizations	174
csetjmp (library header file)	248
csignal (library header file).	248
CSTACK (segment)	
example	41
<i>See also</i> stack	
cstartup	
customizing	67
implementation	90
cstdarg (library header file).	248
cstddef (library header file).	248
cstdio (library header file).	248
cstdlib (library header file)	248
cstring (library header file)	248
ctime (library header file)	248
ctype.h (library header file).	245, 249
C++	
calling convention	98
features excluded from EC++	109
<i>See also</i> Embedded C++ <i>and</i> Extended Embedded C++	
terminology	xxii
C++ header files	246
C-SPY, low-level interface	76, 91
C-SPY, STL container support	118
?C_EXIT (assembler label).	91
?C_GETCHAR (assembler label).	91
C_INCLUDE (environment variable)	169, 183
?C_PUTCHAR (assembler label).	91
C99 standard, added functionality from	82

D

data	
alignment of	137
excluding when linking	81
placement of	149
data bus, enabling external	181
data memory attributes, using	19
data memory, default keywords.	10
data models	17
data pointers	141
default	10
data representation	137
data storage	15
extended keywords.	16
data types	138
floating point	139
integers	138
dataseg (pragma directive)	217
data, initialized	39
data_alignment (pragma directive)	217
__DATE__ (predefined symbol)	230
date (library function), configuring support for.	74
--debug (compiler option)	175, 194
debug information, including in object file	175, 194
declarations and statements, mixing	270
declarations in for loops	270
declaration, of functions	100
declarators, implementation-defined behavior	260
delete operator (extended EC++)	114
delete (keyword)	25
--dependencies (compiler option)	175
deque (STL header file)	247
__derivative__ (predefined symbol)	230
derivatives	
mapping of processor options	6
supported	6
destructors and interrupts, using	119
diagnostic messages	279

- classifying as errors 176
- classifying as remarks 177
- classifying as warnings 177
- disabling warnings 191
- disabling wrapping of 192
- enabling remarks 194
- listing all used 178
- suppressing 177
- diagnostics_tables (compiler option) 178
- diag_default (pragma directive) 218
- diag_error (compiler option) 176
- diag_error (pragma directive) 218
- diag_remark (compiler option) 177
- diag_remark (pragma directive) 218
- diag_suppress (compiler option) 177
- diag_suppress (pragma directive) 218
- diag_warning (compiler option) 177
- diag_warning (pragma directive) 219
- DIFUNCT (segment) 47, 152
- directives
 - function 32
 - pragma 13, 215
- __disable_interrupt (intrinsic function) 238
- disclaimer ii
- DLIB. 11, 244
 - documentation 243
- dlib_config (compiler option) 178
- document conventions xxii
- documentation, library 243
- double (data type) 139
 - specifying 64 bits 201
- double, configuring size of floating-point type 10
- __double_size (runtime model attribute) 79
- DW (directive) 239
- dynamic initialization 64, 88–89
- dynamic initialization in C++ 47
- dynamic memory 25

E

- ec++ (compiler option) 179
- EC++ header files 246
- eec++ (compiler option) 180
- __eeprom (extended keyword) 205–206
- EEPROM, specifying size of inbuilt 180
- EEPROM_AN (segment) 152
- EEPROM_I (segment) 153
- EEPROM_N (segment) 153
- ELPM (instruction) 201
- Embedded C++ 109
 - absolute location 48
 - differences from C++ 109
 - enabling 179
 - function linkage 100
 - language extensions 109
 - overview 109
 - special function types 32
 - static member variables 48
- Embedded C++ objects, placing in memory type 22
- __enable_interrupt (intrinsic function) 238
- enable_multibytes (compiler option) 180
- ENDMOD (assembler directive) 81
- __enhanced_core (runtime model attribute) 79
- enumerations, implementation-defined behavior 259
- enum, data representation 138
- environment
 - implementation-defined behavior 256
 - runtime 85
- environment variables 169
 - C_INCLUDE 169, 183
 - QCCAVR 169
- EQU (assembler directive) 194
- errno.h (library header file) 245, 249
- error messages 280
 - classifying 176
- error return codes 169
- exception handling, missing from Embedded C++ 109

exception vectors	46
exception (library header file)	246
experience, programming	xix
export keyword, missing from Extended EC++	115
Extended Embedded C++	110
enabling	180
standard template library (STL)	247
extended keywords	203
asm	205
data storage	16
default, for memory models	10
enabling	179
functions	27
overview	12
summary	203
syntax	16
@	204
__asm	205
__eeprom	142, 205–206
__far	141, 206
__farflash	142, 206
__farfunc	141, 207
__flash	141, 207
__generic	142, 208
__huge	141, 208
__hugeflash	142, 209
__interrupt	29, 209
<i>See also</i> INTVEC (segment)	
using in pragma directives	225
__intrinsic	210
__io	210
__monitor	210
using in pragma directives	220
__near	141, 210
__nearfunc	141, 211
__noreturn	211
using in pragma directives	221
__no_init	134, 211
using in pragma directives	220

using with __near variables	163
using with __tiny variables	166
__regvar	212
__root	195, 212
using in pragma directives	221
__task	213
__tiny	141, 213
__tinyflash	141, 213
__version_1	214
external data bus, enabling	181
external memory	197–199

F

-f (compiler option)	182
__far (extended keyword)	206
FARCODE (segment)	153
__farflash (extended keyword)	206
__farfunc (extended keyword)	207
FAR_C (segment)	154
FAR_F (segment)	154
FAR_HEAP (segment)	155
FAR_I (segment)	155
FAR_ID (segment)	156
FAR_N (segment)	156
FAR_Z (segment)	156
fatal error messages	280
__FILE__ (predefined symbol)	230
file dependencies, tracking	175
file paths, specifying for #include files	183
filename, of object file	192
__flash (extended keyword)	207
flash memory	197–198
library routines for accessing	249
placing aggregate initializers	184
float (data type)	139
floating point type, configuring size of double	10
floating-point constants	
hexadecimal notation	277

- hints 127
- floating-point format 139
 - hints 126–127
 - implementation-defined behavior 258
 - special cases 140
 - 32-bits 140
 - 64-bits 140
- float.h (library header file) 245, 249
- for loops, declarations in 270
- formats
 - floating-point values 139
 - standard IEEE (floating point) 139
- __formatted_write (library function) 59, 87
- __fractional_multiply_signed (intrinsic function) 239
- __fractional_multiply_signed_with_unsigned 239
- __fractional_multiply_unsigned (intrinsic function) 239
- fragmentation, of heap memory 25
- free (standard library function) 25
- fstream (library header file) 246
- fstream.h (library header file) 246
- __func__ (predefined symbol) 230
- FUNCALL (compiler function directive) 32
- __FUNCTION__ (predefined symbol) 230
- function directives 32
- Function inlining (compiler option) 123
- function inlining, disabling 190–191
- function pointers 141
- function prototypes 131
- function template parameter deduction (extended EC++) 116
- function type information, omitting in object output 192
- FUNCTION (compiler function directive) 32
- functional (STL header file) 247
- functions
 - declaring 100
 - Embedded C++ and special function types 32
 - executing 15
 - extended keywords 27
 - interrupt 29–30
 - intrinsic 93, 130

- monitor 30
- omitting type info 192
- overview 27
- parameters 102
- placing in segments 49
- recursive 131
 - storing data on stack 24
- reentrancy (DLIB) 244
- return values from 105
- special function types 29
- functions, inline 270

G

- __generic (extended keyword) 208
- getchar (library function) 87
- getenv (library function), configuring support for 73
- getzone (library function), configuring support for 74
- glossary xix
- guidelines, reading xix

H

- hash_map (STL header file) 247
- hash_set (STL header file) 247
- __HAS_EEPROM__ (predefined symbol) 230
- __HAS_EIND__ (predefined symbol) 230
- __HAS_ELPM__ (predefined symbol) 231
- __HAS_ENHANCED_CORE__ (predefined symbol) 231
- __HAS_FISCR__ (predefined symbol) 231
- __HAS_MUL__ (predefined symbol) 231
- __HAS_RAMPD__ (predefined symbol) 231
- __HAS_RAMPX__ (predefined symbol) 231
- __HAS_RAMPZ__ (predefined symbol) 231
- header files
 - assert.h 249
 - C 245
 - CLIB 243
 - ctype.h 249

C++	246	IAR Postlink (utility)	52
EC++	246	IAR Technical Support	280
errno.h	249	<code>__IAR_SYSTEMS_ICC__</code> (predefined symbol)	232
float.h	249	<code>__ICCAVR__</code> (predefined symbol)	232
iccbutl.h	249	ICCA90, specifying calling convention	199
limits.h	249	iccbutl.h (library header file)	249
math.h	249	identifiers, implementation-defined behavior	256
setjmp.h	249	IEEE format, floating-point values	139
special function registers	133	implementation, system startup	81
stdarg.h	249	implementation-defined behavior	255
stdbool.h	138, 245, 249	<code>include_alias</code> (pragma directive)	219
stddef.h	139, 249	<code>__indirect_jump_to</code> (intrinsic function)	239
stdio.h	249	inheritance, in Embedded C++	109
stdlib.h	249	initialization	
STL	247	dynamic	64, 88–89
string.h	249	initialization, description of segments	160
using as templates	133	initialized data	39
--header_context (compiler option)	183	initializers, placing in flash memory	184
heap	25	INITTAB (segment)	47, 160
changing default size (command line)	45	inline assembler	95, 130, 271
changing default size (IDE)	45	<i>See also</i> assembler language interface	
size	42–45	inline functions	270
storing data	16	inline (pragma directive)	219
HEAP (segment)	44, 157	inlining of functions, in compiler	123
hidden parameters	105	instruction set, AVR	xix
hints, optimization	130	integer characteristics, adding	83
<code>__huge</code> (extended keyword)	208	integers	138
<code>__hugeflash</code> (extended keyword)	209	casting	142
HUGE_C (segment)	157	implementation-defined behavior	258
HUGE_F (segment)	158	intptr_t	143
HUGE_HEAP (segment)	158	ptrdiff_t	143
HUGE_I (segment)	158	size_t	142
HUGE_ID (segment)	159	uintptr_t	143
HUGE_N (segment)	159	internal error	280
HUGE_Z (segment)	160	<code>__interrupt</code> (extended keyword)	29, 209
		using in pragma directives	225
		interrupt functions	29
		placement in memory	46
		interrupt vector table	29
I			
-I (compiler option)	183		

- interrupt vectors, specifying with pragma directive. 225
 - interrupts
 - disabling 210
 - disabling during function execution 30
 - INTVEC segment 160
 - processor state 23
 - interrupts and EC++ destructors, using 119
 - intptr_t (integer type) 143
 - __intrinsic (extended keyword) 210
 - intrinsic functions 130
 - overview 93
 - summary 237
 - __delay_cycles 238
 - __disable_interrupt 238
 - __enable_interrupt 238
 - __extended_load_program_memory 238
 - __fractional_multiply_signed 239
 - __fractional_multiply_signed_with_unsigned 239
 - __fractional_multiply_unsigned 239
 - __indirect_jump_to 239
 - __insert_opcode 239
 - __load_program_memory 239
 - __multiply_signed 239
 - __multiply_signed_with_unsigned 239
 - __multiply_unsigned 240
 - __no_operation 240
 - __require 240
 - __restore_interrupt 240
 - __reverse 240
 - __save_interrupt 241
 - __segment_begin 241
 - __segment_end 242
 - __sleep 242
 - __swap_nibbles 242
 - __watchdog_reset 242
 - intrinsics.h (header file) 238
 - INTVEC (segment) 46, 160
 - __io (extended keyword) 210
 - iomanip (library header file) 246
 - iomanip.h (library header file) 246
 - ios (library header file) 246
 - iosfwd (library header file) 246
 - iostream (library header file) 246
 - iostream.h (library header file) 246
 - ISO/ANSI C 11, 243
 - C++ features excluded from EC++ 109
 - language extensions 269
 - specifying strict usage 196
 - iso646.h (library header file) 245
 - istream (library header file) 246
 - iterator (STL header file) 247
- ## K
- keywords, extended. 12, 203
- ## L
- l (compiler option). 97, 185
 - language extensions
 - descriptions 269
 - Embedded C++ 109
 - enabling 179
 - language (pragma directive) 220
 - libraries. 4
 - runtime. 56, 85
 - standard template library 247
 - library configuration file
 - modifying 63
 - option for specifying 178
 - library documentation 243
 - library features, missing from Embedded C++ 110
 - library functions 243
 - for accessing flash 249
 - choosing printf formatter 59
 - choosing sprintf formatter 59
 - getchar 87
 - memcmp_G 250

memcpy_G	250
memcpy_P	250
printf	87
printf_P	250
putchar	87
puts_G	250
puts_P	251
reference information	xxi
remove	70
rename	70
scanf_P	251
sprintf	87
sprintf_P	251
sscanf_P	251
strcat_G	251
strcmp_G	251
strcmp_P	251
strcpy_G	251
strcpy_P	252
strerror_P	252
strlen_G	252
strlen_P	252
strncat_G	252
strncmp_G	252
strncpy_P	252
strncpy_G	252
strncpy_P	253
summary	245, 249
__close	70
__lseek	70
__open	70
__read	70
__write	70
library functions (CLIB)	
choosing scanf formatter	89
choosing sscanf formatter	89
library functions (DLIB)	
choosing scanf formatter	60
choosing sscanf formatter	60
library modules, creating	186
library object files	244
--library_module (compiler option)	186
limits.h (library header file)	245, 249
__LINE__ (predefined symbol)	232
linkage, C and C++	100
linker command files	
contents	34
customizing	35
customizing code segments	46
customizing data segments	37
customizing for RAM	41
customizing for static data	37
customizing initialized data	39
customizing located data	46
customizing the data stack	41
customizing the heap	44
customizing the RSTACK	43
introduction	34
template	35
using the -Z command	36
linking, from the command line	4
list (STL header file)	247
listing, generating	185
literals, compound	271
literals, placing in initialized data segments	200
literature, recommended	xxi
locale.h (library header file)	245
location (pragma directive)	28, 48–49, 220
LOCFRAME (compiler function directive)	32
loop-invariant expressions	124
low-level processor operations	237
__low_level_init, customizing	66
__lseek (library function)	70
M	
__MEMORY_MODEL__ (predefined symbol)	232
macros, variadic	235, 270

- malloc (standard library function) 25
 - map (STL header file) 247
 - math.h (library header file) 83, 245, 249
 - _medium_write (library function) 88
 - memcmp_G (library function) 250
 - memcpy_G (library function) 250
 - memcpy_P (library function) 250
 - memory
 - accessing 9
 - allocating in Embedded C++ 25
 - dynamic 25
 - external 197–199
 - flash 197–198
 - 64 Kbytes large 201
 - heap 25
 - non-initialized 134
 - non-volatile 156
 - RAM, saving 131
 - releasing in Embedded C++ 25
 - stack 23
 - saving 131
 - static 15
 - used by executing functions 15
 - used by global or static variables 15
 - memory management, type-safe 109
 - memory models
 - configuration 9
 - specifying 186
 - memory types 18
 - Embedded C++ 22
 - hints 128
 - placing variables in 22
 - pointers 20
 - specifying 19
 - structures 21
 - memory (STL header file) 247
 - __memory_model (runtime model attribute) 79
 - message (pragma directive) 220
 - migration_preprocessor_extensions (compiler option) . . . 187
 - misrac (compiler option) 187
 - misrac_verbose (compiler option) 188
 - module consistency 77
 - rtmodel 223
 - module name, specifying 188
 - module size, maximum 8
 - MODULE (assembler directive) 81
 - modules, assembler 81
 - module_name (compiler option) 188
 - __monitor (extended keyword) 210
 - using in pragma directives 220
 - monitor functions 30, 210
 - multibyte character support 180
 - multiple inheritance
 - in Extended EC++ 110
 - missing from Embedded C++ 109
 - multiple output files, from XLINK 51
 - __multiply_signed (intrinsic function) 239
 - __multiply_signed_with_unsigned (intrinsic function) . . 239
 - __multiply_unsigned (intrinsic function) 240
 - mutable attribute, in Extended EC++ 118
- ## N
- namespace support
 - in Extended EC++ 110, 119
 - missing from Embedded C++ 110
 - name, specifying for object file 192
 - NDEBUG (preprocessor symbol) 232
 - __near (extended keyword) 210
 - __nearfunc (extended keyword) 211
 - NEAR_C (segment) 161
 - NEAR_F (segment) 161
 - NEAR_HEAP (segment) 161
 - NEAR_I (segment) 162
 - NEAR_ID (segment) 162
 - NEAR_N (segment) 163
 - NEAR_Z (segment) 163
 - new operator (extended EC++) 114

new (keyword)	25
new (library header file)	246
new.h (library header file)	246
non-initialized variables	134
non-scalar parameters	131
non-volatile memory	156
NOP (assembler instruction)	240
__noreturn (extended keyword)	211
using in pragma directives	221
--no_code_motion (compiler option)	189
--no_cse (compiler option)	190
__no_init (extended keyword)	134, 211
using in pragma directives	220
--no_inline (compiler option)	190–191
__no_operation (intrinsic function)	240
__no_rampd (runtime model attribute)	79
compiler options	
--no_typedefs_in_diagnostics (compiler option)	191
--no_warnings (compiler option)	191
--no_wrap_diagnostics (compiler option)	192
NULL	249
numeric (STL header file)	247

O

-o (compiler option)	192
object filename, specifying	192
object module name, specifying	188
object_attribute (pragma directive)	134, 220
offsetof	249
--omit_types (compiler option)	192
--only_stdout (compiler option)	193
__open (library function)	70
operators	
@	28, 48–49
__memory_of(class)	113
optimization	
clustering, disabling	189
code motion, disabling	189

common sub-expression elimination, disabling	190
configuration	10
cross-call	174
function inlining, disabling	190–191
hints	130
size, specifying	200
speed, specifying	195
types and levels	122
type-based alias analysis	124
optimization techniques	123
optimize (pragma directive)	221
options summary, compiler	169
ostream (library header file)	246
output	
specifying	5
specifying file name	5
output files, from XLINK	5
multiple	51
output, preprocessor	193

P

pack (pragma directive)	222
parameters	
function	102
hidden	105
non-scalar	131
register	102
register vs stack	102
specifying	168
stack	102, 104
typographic convention	xxii
permanent registers	101
placement of code and data	149
pointer types, differences between	21
pointers	
casting	142
data	10, 141
function	141

- implementation-defined behavior 259
- to constructor blocks 152
- using instead of large non-scalar parameters 131
- polymorphism, in Embedded C++ 109
- porting, of code
 - containing pragma directives 216
- Postlink (utility) 52
- postlink.htm 52
- _Pragma (preprocessor operator) 232–233, 235
- pragma directives 13
 - basic_template_matching 216
 - basic_template_matching.using 117
 - bitfields 139, 217
 - constseg 217
 - dataseg 217
 - data_alignment 217
 - diag_default 218
 - diag_error 218
 - diag_remark 218
 - diag_suppress 218
 - diag_warning 219
 - include_alias 219
 - inline 219
 - language 220
 - location 28, 48–49, 220
 - message 220
 - object_attribute 134, 220
 - optimize 221
 - pack 222
 - required 223
 - rtmodel 223
 - segment 224
 - summary 215
 - syntax 216
 - type_attribute 224
 - vector 29, 225
- _Pragma() (predefined symbol) 233
- predefined symbols
 - backward compatibility 229
 - overview 13
 - summary 228
 - __ALIGNOF__ 229
 - __BASE_FILE__ 229
 - __CORE__ 229
 - __cplusplus 229
 - __CPU__ 229
 - __DATE__ 230
 - __derivative__ 230
 - __embedded_cplusplus 230
 - __FILE__ 230
 - __FUNCTION__ 230
 - __func__ 230
 - __HAS_EEPROM__ 230
 - __HAS_EIND__ 230
 - __HAS_ELPM__ 231
 - __HAS_ENHANCED_CORE__ 231
 - __HAS_FISCR__ 231
 - __HAS_MUL__ 231
 - __HAS_RAMPD__ 231
 - __HAS_RAMPX__ 231
 - __HAS_RAMPZ__ 231
 - __IAR_SYSTEMS_ICC__ 232
 - __ICCAVR__ 232
 - __LINE__ 232
 - __MEMORY_MODEL__ 232
 - __PRETTY_FUNCTION__ 233
 - __STDC__ 233
 - __STDC_VERSION__ 233
 - __TID__ 233
 - __TIME__ 234
 - __VERSION_1_CALLS__ 234
 - __VER__ 234
 - _Pragma() 233
 - preinclude (compiler option) 193
 - preprocess (compiler option) 193
 - preprocessing directives, implementation-defined behavior 260
 - preprocessor extension
 - #warning message 235
 - __VA_ARGS__ 235

preprocessor output	193
preprocessor symbols	228
defining	174
NDEBUG	232
preprocessor, extending	187
prerequisites (programming experience)	xix
__PRETTY_FUNCTION__ (predefined symbol)	233
print formatter, selecting	60
printf (library function)	59, 87
choosing formatter	59
printf_P (library function)	250
processor operations, low-level	237
processor options, mapping of derivatives	6
processor variant	
configuration	5
specifying on command line	173, 197
program size, maximum	8
programming experience, required	xix
programming hints	130
project options, setting	6
ptrdiff_t (integer type)	143, 249
PUBLIC (assembler directive)	194
--public_equ (compiler option)	194
putchar (library function)	87
puts_G (library function)	250
puts_P (library function)	251

Q

QCAVR (environment variable)	169
qualifiers, implementation-defined behavior	260
queue (STL header file)	247

R

-r (compiler option)	175, 194
raise (library function), configuring support for	73
RAM memory, saving	131
RAMP (register)	190

RAMPZ (register)	201
__read (library function)	70
read formatter, selecting	61, 89
reading guidelines	xix
reading, recommended	xxi
realloc (standard library function)	25
recursive functions	131
storing data on stack	24
reentrancy (DLIB)	244
reference information, typographic convention	xxii
register parameters	102
register vs stack parameters	102
registered trademarks	ii
registers	
callee-save, stored on stack	23
implementation-defined behavior	259
locking	186
permanent	101
scratch	101
using RAMPZ in direct access mode	190
__regvar (extended keyword)	212
reinterpret_cast (cast operator)	110
remark (diagnostic message)	
classifying	177
enabling	194
--remarks (compiler option)	194
remarks (diagnostic message)	279
remove (library function)	70
rename (library function)	70
__require (intrinsic function)	240
REQUIRED (assembler directive)	81
required (pragma directive)	223
--require_prototypes (compiler option)	194
return data stack	
reducing usage of	174
using cross-call optimizations	189
return values, from functions	105
__root (extended keyword)	212
using in pragma directives	221

routines, time-critical 93, 237
 RSEG (assembler directive) 81
 RSTACK (segment) 163
 RSTACK, size of stack 43
 RTMODEL (assembler directive) 78
 rtmodel (pragma directive) 223
 rtti support, missing from STL 110
 __rt_version (runtime model attribute) 79
 runtime environment 85
 runtime libraries 56, 85
 introduction 243
 naming convention (CLIB) 86
 naming convention (DLIB) 57
 runtime model attributes 77
 __cpu 79
 __cpu_name 79
 __double_size 79
 __enhanced_core 79
 __memory_model 79
 __no_rampd 79
 __rt_version 79
 runtime type information, missing from Embedded C++ . 109

S

-s (compiler option) 195
 scanf (library function)
 choosing formatter (CLIB) 89
 choosing formatter (DLIB) 60
 scanf_P (library function) 251
 scratch registers 101
 segment memory types, in XLINK 34
 segment name, specifying 195
 segment parts, unused 81
 segment (pragma directive) 224
 segments 149
 CODE 151
 CSTACK, example 41
 DIFUNCT 47, 152

EEPROM_AN 152
 EEPROM_I 153
 EEPROM_N 153
 FARCODE 153
 FAR_C 154
 FAR_F 154
 FAR_HEAP 155
 FAR_I 155
 FAR_ID 156
 FAR_N 156
 FAR_Z 156
 HEAP 44, 157
 HUGE_C 157
 HUGE_F 158
 HUGE_HEAP 158
 HUGE_I 158
 HUGE_ID 159
 HUGE_N 159
 HUGE_Z 160
 INITTAB 47, 160
 introduction 33
 INTVEC 46, 160
 NEAR_C 161
 NEAR_F 161
 NEAR_HEAP 161
 NEAR_I 162
 NEAR_ID 162
 NEAR_N 163
 NEAR_Z 163
 RSTACK 163
 reducing usage of 174
 using cross-call optimizations 189
 summary 149
 SWITCH 47, 164
 TINY_F 164
 TINY_HEAP 164
 TINY_I 165
 TINY_ID 165
 TINY_N 166

TINY_Z	166	sscanf_P (library function)	251
__segment_begin (intrinsic function)	241	sstream (library header file)	246
__segment_end (intrinsic function)	242	stack	23
SEI (assembler instruction)	238	advantages and problems using	24
semaphores, operations on	210	changing default size (from command line)	42–43
set (STL header file)	247	changing default size (in Embedded Workbench)	42–43
setjmp.h (library header file)	245, 249	contents of	23
settings, basic for project configuration	5	function usage	15
severity level, of diagnostic messages	279	internal data	151
specifying	280	return data	163
SFR (special function registers)	133	saving space	131
declaring extern	49	size	42
signal (library function), configuring support for	73	maximum	10
signal.h (library header file)	245	stack layout, at function entrance	104
signed char (data type)	138–139	stack parameters	102, 104
specifying	173	stack pointer	24
signed int (data type)	138	stack (STL header file)	247
signed long (data type)	138	stack, return address stack	43
signed short (data type)	138	standard error	193
--silent (compiler option)	196	standard output, specifying	193
silent operation, specifying	196	standard template library (STL)	
64-bits (floating-point format)	140	in Extended EC++	110, 117, 247
size of EEPROM, specifying	180	missing from Embedded C++	110
size optimization, specifying	200	startup in CLIB, system	89
size_t (integer type)	142, 249	startup, system	65
skeleton code, creating for assembler language interface	96	statements, implementation-defined behavior	260
SLEEP (instruction)	242	static memory	15
slist (STL header file)	247	static overlay	32
_small_write (library function)	88	static_cast (cast operator)	110
source files, list all referred	183	std namespace, missing from EC++	
special function registers (SFR)	133	and Extended EC++	119
special function types	29	stdarg.h (library header file)	245, 249
speed optimization, specifying	195	stdbool.h (library header file)	83, 138, 245, 249
printf (library function)	59, 87	__STDC__ (predefined symbol)	233
choosing formatter	59	__STDC_VERSION__ (predefined symbol)	233
printf_P (library function)	251	stddef.h (library header file)	139, 245, 249
scanf (library function)		stderr	70, 193
choosing formatter (CLIB)	89	stdexcept (library header file)	246
choosing formatter (DLIB)	60	stdin	70
		stdint.h (library header file)	83

stdio.h (library header file) 83, 245, 249
 stdlib.h (library header file) 83, 245, 249
 stdout 70, 193
 STL 117
 strcat_G (library function) 251
 strcmp_G (library function) 251
 strcmp_P (library function) 251
 strcpy_G (library function) 251
 strcpy_P (library function) 252
 streambuf (library header file) 246
 streams, supported in Embedded C++ 110
 strerror_P (library function) 252
 --strict_ansi (compiler option) 196
 string (library header file) 246
 strings, supported in Embedded C++ 110
 string.h (library header file) 245, 249
 strlen_G (library function) 252
 strlen_P (library function) 252
 strncat_G (library function) 252
 strncmp_G (library function) 252
 strncmp_P (library function) 252
 strncpy_G (library function) 252
 strncpy_P (library function) 253
 strstream (library header file) 246
 strtod (library function), configuring support for 74
 structure types
 alignment 143
 layout 143
 structures
 anonymous 128
 implementation-defined behavior 259
 placing in memory type 21
 support, technical 280
 __swap_nibbles (intrinsic function) 242
 SWITCH (segment) 47, 164
 symbols
 anonymous, creating 271
 overview of predefined 13
 preprocessor, defining 174

syntax, extended keywords 16
 system startup 65
 system startup in CLIB 89
 system startup, implementation 81
 system termination 65, 90
 system (library function), configuring support for 73

T

__task (extended keyword) 213
 technical support, IAR 280
 template support
 in Extended EC++ 110, 115
 missing from Embedded C++ 109
 Terminal I/O window, in C-SPY 91
 termination, system 65, 90
 terminology xix, xxii
 32-bits (floating-point format) 140
 this pointer, referring to a class object (extended EC++) . . 111
 this (pointer) 98
 __TID__ (predefined symbol) 233
 __TIME__ (predefined symbol) 234
 time (library function), configuring support for 74
 time-critical routines 93, 237
 time.h (library header file) 245
 __tiny (extended keyword) 213
 __tinyflash (extended keyword) 213
 TINY_F (segment) 164
 TINY_HEAP (segment) 164
 TINY_I (segment) 165
 TINY_ID (segment) 165
 TINY_N (segment) 166
 TINY_Z (segment) 166
 tips, programming 130
 trademarks ii
 translation, implementation-defined behavior 255
 trap vectors, specifying with pragma directive 225
 type information, omitting 192
 Type-based alias analysis (compiler option) 124

type-safe memory management	109
type_attribute (pragma directive)	224
typographic conventions	xxii

U

UBROF messages, excluding from object file	191
uintptr_t (integer type)	143
unions	
anonymous	128
implementation-defined behavior	259
unsigned char (data type)	138–139
changing to signed char	173
unsigned int (data type)	138
unsigned short (data type)	138
utility (STL header file)	247

V

-v (compiler option), mapping of AVR cores	6
variable type information, omitting in object output	192
variables	
auto	23
defined inside a function	23
global, placement in memory	15
local. <i>See</i> auto variables	
non-initialized	134
omitting type info	192
placing at absolute addresses	47
placing in named segments	47
placing in segments	48
specifying as <code>__root</code>	195
static, placement in memory	15
variadic macros	270
vector (pragma directive)	29, 225
vector (STL header file)	247
<code>__VER__</code> (predefined symbol)	234
version, of compiler	234
<code>__version_1</code> (extended keyword)	214

<code>__VERSION_1_CALLS__</code> (predefined symbol)	234
volatile, declaring objects	146

W

#warning message (preprocessor extension)	235
warnings	279
classifying	177
disabling	191
exit code	199
--warnings_affect_exit_code (compiler option)	169
--warnings_are_errors (compiler option)	200
watchdog reset instruction	242
wchar.h (library header file)	245
wchar_t (data type), adding support for in C	139
wctype.h (library header file)	245
<code>__write</code> (library function)	70
write formatter, selecting	88–89

X

XLINK options, managing multiple output files	51
XLINK output files	5
XLINK segment memory types	34

Z

-z (compiler option)	200
----------------------	-----

Symbols

#include files, specifying	183
#warning message (preprocessor extension)	235
-D (compiler option)	174
-e (compiler option)	179
-f (compiler option)	182
-I (compiler option)	183
-l (compiler option)	97, 185
-m (compiler option)	186

- o (compiler option) 192
- r (compiler option) 175, 194
- s (compiler option) 195
- v (compiler option) 197
 - mapping of AVR derivatives 6
- y (compiler option) 155, 165, 200
- z (compiler option) 200
- char_is_signed (compiler option) 173
- cpu (compiler option) 173
 - mapping of AVR derivatives 6
- cross_call_passes (compiler option) 174
- debug (compiler option) 175, 194
- dependencies (compiler option) 175
- diagnostics_tables (compiler option) 178
- diag_error (compiler option) 176
- diag_remark (compiler option) 177
- diag_suppress (compiler option) 177
- diag_warning (compiler option) 177
- disable_direct_mode (compiler option) 178
- dlib_config (compiler option) 178
- do_cross_call (compiler option) 179
- ec++ (compiler option) 179
- eocr_address (compiler option) 180
- eec++ (compiler option) 180
- eeprom_size (compiler option) 180
- enable_external_bus (compiler option) 181
- enable_multibytes (compiler option) 180
- enhanced_core (compiler option) 181
- error_limit (compiler option) 181
- force_switch_type (compiler option) 182
- header_context (compiler option) 183
- initializers_in_flash (compiler option) 184, 197–198
- library_module (compiler option) 186
- lock_reg (compiler option) 186
- memory_model (compiler option) 186
- migration_preprocessor_extensions (compiler option) 187
- misrac (compiler option) 187
- misrac_verbosity (compiler option) 188
- module_name (compiler option) 188
- no_clustering (compiler option) 189
- no_code_motion (compiler option) 189
- no_cross_call (compiler option) 189
- no_cse (compiler option) 190
- no_inline (compiler option) 190–191
- no_rampd (compiler option) 190
- no_tbaa (compiler option) 191
- no_typedefs_in_diagnostics (compiler option) 191
- no_ubrof_messages (compiler option) 191
- no_warnings (compiler option) 191
- no_wrap_diagnostics (compiler option) 192
- omit_types (compiler option) 192
- only_stdout (compiler option) 193
- preinclude (compiler option) 193
- preprocess (compiler option) 193
- remarks (compiler option) 194
- require_prototypes (compiler option) 194
- root_variables (compiler option) 195
- segment (compiler option) 195
 - separate_cluster_for_initialized_variables
(compiler option) 196
 - silent (compiler option) 196
 - spmc_r_address (compiler option) 196
 - strict_ansi (compiler option) 196
 - string_literals_in_flash (compiler option) 196
 - version1_calls (compiler option) 199
 - warnings_affect_exit_code (compiler option) 169, 199
 - warnings_are_errors (compiler option) 200
 - zero_register (compiler option) 201
 - 64bit_doubles (compiler option) 201
 - 64k_flash (compiler option) 201
 - ?C_EXIT (assembler label) 91
 - ?C_GETCHAR (assembler label) 91
 - ?C_PUTCHAR (assembler label) 91
 - @ (extended keyword) 204
 - @ (operator) 28, 48–49
 - _ _ALIGNOF_ _() (predefined symbol) 229
 - _ _asm (extended keyword) 205
 - _ _BASE_FILE_ _ (predefined symbol) 229
 - _ _close (library function) 70

__CORE__ (predefined symbol)	229	__indirect_jump_to (intrinsic function)	239
__cplusplus (predefined symbol)	229	__insert_opcode (intrinsic function)	239
__cpu (runtime model attribute)	79	__interrupt (extended keyword)	29, 209
__CPU__ (predefined symbol)	229	using in pragma directives	225
__cpu_name (runtime model attribute)	79	__intrinsic (extended keyword)	210
__DATE__ (predefined symbol)	230	__io (extended keyword)	210
__delay_cycles (intrinsic function)	238	__LINE__ (predefined symbol)	232
__derivative__ (predefined symbol)	230	__load_program_memory (intrinsic function)	239
__disable_interrupt (intrinsic function)	238	__low_level_init, customizing	66
__double_size (runtime model attribute)	79	__lseek (library function)	70
__eeprom (extended keyword)	142, 205–206	__memory_model (runtime model attribute)	79
__embedded_cplusplus (predefined symbol)	230	__MEMORY_MODEL__ (predefined symbol)	232
__enable_interrupt (intrinsic function)	238	__monitor (extended keyword)	210
__enhanced_core (runtime model attribute)	79	using in pragma directives	220
__extended_load_program_memory (intrinsic function)	238	__multiply_signed (intrinsic function)	239
__far (extended keyword)	141, 206	__multiply_signed_with_unsigned (intrinsic function)	239
__farflash (extended keyword)	142, 206	__multiply_unsigned (intrinsic function)	240
__farfunc (extended keyword)	141, 207	__near (extended keyword)	141, 210
__FILE__ (predefined symbol)	230	__nearfunc (extended keyword)	141, 211
__flash (extended keyword)	141, 207	__noreturn (extended keyword)	211
__fractional_multiply_signed (intrinsic function)	239	using in pragma directives	221
__fractional_multiply_signed_with_unsigned	239	__no_init (extended keyword)	134, 163, 166, 211
__fractional_multiply_unsigned (intrinsic function)	239	using in pragma directives	220
__FUNCTION__ (predefined symbol)	230	__no_operation (intrinsic function)	240
__func__ (predefined symbol)	230	__no_rampd (runtime model attribute)	79
__generic (extended keyword)	142, 208	__open (library function)	70
__HAS_EEPROM__ (predefined symbol)	230	__PRETTY_FUNCTION__ (predefined symbol)	233
__HAS_EIND__ (predefined symbol)	230	__read (library function)	70
__HAS_ELPM__ (predefined symbol)	231	__regvar (extended keyword)	212
__HAS_ENHANCED_CORE__ (predefined symbol)	231	__require (intrinsic function)	240
__HAS_FISCR__ (predefined symbol)	231	require, adding	181
__HAS_MUL__ (predefined symbol)	231	__restore_interrupt (intrinsic function)	240
__HAS_RAMPD__ (predefined symbol)	231	__reverse (intrinsic function)	240
__HAS_RAMPX__ (predefined symbol)	231	__root (extended keyword)	195, 212
__HAS_RAMPZ__ (predefined symbol)	231	using in pragma directives	221
__huge (extended keyword)	141, 208	__rt_version (runtime model attribute)	79
__hugeflash (extended keyword)	142, 209	__save_interrupt (intrinsic function)	241
__IAR_SYSTEMS_ICC__ (predefined symbol)	232	__segment_begin (intrinsic function)	241
__ICCAVR__ (predefined symbol)	232	__segment_end (intrinsic function)	242

__sleep (intrinsic function)	242
__STDC__ (predefined symbol)	233
__STDC_VERSION__ (predefined symbol)	233
__swap_nibbles (intrinsic function)	242
__task (extended keyword)	213
__TID__ (predefined symbol)	233
__TIME__ (predefined symbol)	234
__tiny (extended keyword)	141, 213
__tinyflash (extended keyword)	141, 213
__version_1 (extended keyword)	214
__VERSION_1_CALLS__ (predefined symbol)	234
__VER__ (predefined symbol)	234
__watchdog_reset (intrinsic function)	242
__write (library function)	70
_formatted_write (library function)	59, 87
_medium_write (library function)	88
_Pragma (preprocessor operator)	232–233, 235
_Pragma() (predefined symbol)	233
_small_write (library function)	88
__memory_of(class), operator	113
__VA_ARGS__ (preprocessor extension)	235

Numerics

32-bits (floating-point format)	140
64-bit doubles, specifying	201
64-bits (floating-point format)	140