

H8 IAR C/C++ Compiler

Reference Guide

for Renesas

H8/300H and H8S Microcomputer Families

COPYRIGHT NOTICE

© Copyright 1996–2006 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.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

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

EDITION NOTICE

First edition: June 2006

Part number: CH8-1

This guide applies to version 2.x of H8 IAR Embedded Workbench®.

Brief contents

| | |
|--------------------------------------------------|------------|
| Tables | xv |
| Preface | xvii |
| Part 1. Using the compiler | 1 |
| Getting started | 3 |
| Data storage | 13 |
| Functions | 25 |
| Placing code and data | 33 |
| The DLIB runtime environment | 51 |
| Assembler language interface | 83 |
| Using C++ | 103 |
| Efficient coding for embedded applications | 117 |
| Part 2. Compiler reference | 133 |
| Compiler usage | 135 |
| Compiler options | 143 |
| Data representation | 175 |
| Compiler extensions | 187 |
| Extended keywords | 199 |
| Pragma directives | 215 |
| Intrinsic functions | 227 |
| The preprocessor | 245 |
| Library functions | 255 |

| | |
|---------------------------------------|-----|
| Segment reference | 263 |
| Implementation-defined behavior | 283 |
| Index | 295 |

Contents

| | |
|-------------------------------------------------------|-------|
| Tables | xv |
| Preface | xvii |
| Who should read this guide | xvii |
| How to use this guide | xvii |
| What this guide contains | xviii |
| Other documentation | xix |
| Further reading | xix |
| Document conventions | xx |
| Typographic conventions | xx |
| | |
| Part I. Using the compiler | 1 |
| Getting started | 3 |
| IAR language overview | 3 |
| Supported H8/300H and H8S derivatives | 4 |
| Building applications—an overview | 4 |
| Compiling | 4 |
| Linking | 4 |
| Basic settings for project configuration | 5 |
| Core | 6 |
| Operating mode | 6 |
| Data model | 6 |
| Code model | 7 |
| Hardware configuration | 7 |
| Configuration dependencies | 7 |
| Size of double floating-point type | 8 |
| Optimization for speed and size | 8 |
| Runtime environment | 8 |
| Special support for embedded systems | 10 |
| Extended keywords | 10 |
| Pragma directives | 10 |

| | |
|---------------------------------------------------------------------------------|----|
| Predefined symbols | 10 |
| Special function types | 10 |
| Header files for I/O | 11 |
| Accessing low-level features | 11 |
| Data storage | 13 |
| Introduction | 13 |
| Data models | 14 |
| Specifying a data model | 15 |
| Memory types | 15 |
| Bitvar | 16 |
| Data8 memory | 16 |
| Data16 memory | 17 |
| Data32 memory | 17 |
| Using data memory attributes | 18 |
| Pointers and memory types | 19 |
| Structures and memory types | 20 |
| More examples | 20 |
| C++ and memory types | 21 |
| The stack and auto variables | 22 |
| Dynamic memory on the heap | 23 |
| Functions | 25 |
| Function-related extensions | 25 |
| Code models and memory attributes for function storage | 25 |
| Primitives for interrupts, concurrency, and OS-related programming | 26 |
| Interrupt functions | 26 |
| Trap functions | 27 |
| Monitor functions | 28 |
| C++ and special function types | 31 |
| 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 |
| The stack | 40 |
| The heap | 41 |
| Located data | 43 |
| Code segments | 43 |
| Normal code | 43 |
| Interrupt vectors | 44 |
| Function vectors for non-interrupt functions | 44 |
| C++ dynamic initialization | 45 |
| Controlling data and function placement in memory | 45 |
| Data placement at an absolute location | 46 |
| Data and function placement in segments | 48 |
| Verifying the linked result of code and data placement | 49 |
| Segment too long errors and range errors | 49 |
| Linker map file | 49 |
| The DLIB runtime environment | 51 |
| Introduction to the runtime environment | 51 |
| Runtime environment functionality | 51 |
| Library selection | 52 |
| Situations that require library building | 52 |
| Library configurations | 53 |
| Debug support in the runtime library | 53 |
| Using a prebuilt library | 54 |
| Customizing a prebuilt library without rebuilding | 56 |
| Choosing formatters for printf and scanf | 57 |
| Choosing printf formatter | 57 |
| Choosing scanf formatter | 58 |
| Overriding library modules | 59 |
| Building and using a customized library | 61 |
| Setting up a library project | 61 |

| | |
|---------------------------------------------------------|-----------|
| Modifying the library functionality | 61 |
| Using a customized library | 62 |
| System startup and termination | 63 |
| System startup | 64 |
| System termination | 64 |
| Customizing system initialization | 65 |
| __low_level_init | 65 |
| Modifying the file cstartup.s37 | 66 |
| Standard streams for input and output | 66 |
| Implementing low-level character input and output | 67 |
| Configuration symbols for printf and scanf | 68 |
| Customizing formatting capabilities | 69 |
| File input and output | 69 |
| Locale | 70 |
| Locale support in prebuilt libraries | 71 |
| Customizing the locale support | 71 |
| Changing locales at runtime | 72 |
| Environment interaction | 72 |
| Signal and raise | 73 |
| Time | 74 |
| Strtod | 74 |
| Assert | 74 |
| Heaps | 75 |
| C-SPY Debugger runtime interface | 75 |
| Low-level debugger runtime interface | 76 |
| The debugger terminal I/O window | 76 |
| Checking module consistency | 77 |
| Runtime model attributes | 77 |
| Using runtime model attributes | 78 |
| Predefined runtime attributes | 78 |
| Using a weak runtime model check | 80 |
| User-defined runtime model attributes | 81 |

| | |
|------------------------------------------------------------|-----|
| Assembler language interface | 83 |
| Mixing C and assembler | 83 |
| Intrinsic functions | 83 |
| Mixing C and assembler modules | 84 |
| Inline assembler | 85 |
| Calling assembler routines from C | 86 |
| Creating skeleton code | 86 |
| Compiling the code | 87 |
| Calling assembler routines from C++ | 88 |
| Calling convention | 89 |
| Choosing a calling convention | 90 |
| Function declarations | 91 |
| C and C++ linkage | 91 |
| Preserved versus scratch registers | 92 |
| Function call | 93 |
| Function exit | 95 |
| Restrictions for special function types | 96 |
| Examples | 96 |
| Function directives | 97 |
| Memory access methods | 98 |
| Example code for showing differences in memory types | 99 |
| The data16 memory access method | 99 |
| The data32 memory access method | 100 |
| The data8 memory access method | 100 |
| Call frame information | 101 |
| Using C++ | 103 |
| Overview | 103 |
| Standard Embedded C++ | 103 |
| Extended Embedded C++ | 104 |
| Enabling C++ support | 104 |
| Feature descriptions | 105 |
| Classes | 105 |
| Functions | 108 |

| | |
|--------------------------------------------------------------|-----|
| New and Delete operators | 109 |
| Templates | 110 |
| Variants of casts | 113 |
| Mutable | 113 |
| Namespace | 113 |
| The STD namespace | 113 |
| Pointer to member functions | 113 |
| Using interrupts and C++ destructors | 114 |
| C++ language extensions | 114 |
| Efficient coding for embedded applications | 117 |
| Taking advantage of the compilation system | 117 |
| Controlling compiler optimizations | 118 |
| Fine-tuning enabled transformations | 119 |
| Selecting data types and placing data in memory | 121 |
| Using efficient data types | 121 |
| Data model and data memory attributes | 122 |
| Rearranging elements in a structure | 123 |
| Anonymous structs and unions | 124 |
| Writing efficient code | 126 |
| Saving stack space and RAM memory | 126 |
| Stack pointer arithmetics | 127 |
| Function prototypes | 127 |
| Function calls | 128 |
| Integer types and bit negation | 129 |
| Protecting simultaneously accessed variables | 129 |
| Accessing special function registers | 130 |
| Non-initialized variables | 130 |
| | |
| Part 2. Compiler reference | 133 |
| Compiler usage | 135 |
| Compiler invocation | 135 |
| Invocation syntax | 135 |

| | |
|----------------------------------------------|-----|
| Passing options to the compiler | 136 |
| Environment variables | 136 |
| Include file search procedure | 136 |
| Compiler output | 138 |
| Diagnostics | 139 |
| Message format | 139 |
| Severity levels | 140 |
| Setting the severity level | 140 |
| Internal error | 140 |
| Compiler options | 143 |
| Compiler options syntax | 143 |
| Types of options | 143 |
| Rules for specifying parameters | 143 |
| Options summary | 146 |
| Descriptions of options | 148 |
| Data representation | 175 |
| Alignment | 175 |
| Alignment in the H8 IAR C/C++ Compiler | 176 |
| Basic data types | 176 |
| Integer types | 176 |
| Floating-point types | 178 |
| Pointer types | 179 |
| Function pointers | 179 |
| Data pointers | 180 |
| Casting | 180 |
| Structure types | 182 |
| Alignment | 182 |
| General layout | 182 |
| Packed structure types | 183 |
| Type qualifiers | 183 |
| Declaring objects volatile | 183 |
| Declaring objects const | 184 |

| | |
|-------------------------------------------------------------------|-----|
| Data types in C++ | 185 |
| Compiler extensions | 187 |
| Compiler extensions overview | 187 |
| Enabling language extensions | 188 |
| C language extensions | 188 |
| Important language extensions | 189 |
| Useful language extensions | 190 |
| Minor language extensions | 194 |
| Extended keywords | 199 |
| General syntax rules for extended keywords | 199 |
| Type attributes | 199 |
| Object attributes | 202 |
| Summary of extended keywords | 203 |
| Descriptions of extended keywords | 204 |
| Pragma directives | 215 |
| Summary of pragma directives | 215 |
| Descriptions of pragma directives | 216 |
| Intrinsic functions | 227 |
| Summary of intrinsic functions | 227 |
| Descriptions of intrinsic functions | 229 |
| The preprocessor | 245 |
| Overview of the preprocessor | 245 |
| Predefined preprocessor symbols | 246 |
| Summary of predefined symbols | 246 |
| Descriptions of predefined symbols | 247 |
| Description of miscellaneous preprocessor extensions | 252 |
| Library functions | 255 |
| Introduction | 255 |
| Header files | 255 |
| Library object files | 255 |

| | |
|--------------------------------------------------------------|------------|
| Reentrancy | 256 |
| IAR DLIB Library | 256 |
| C header files | 257 |
| C++ header files | 257 |
| Added C functionality | 260 |
| ctype.h | 260 |
| inttypes.h | 260 |
| math.h | 260 |
| stdbool.h | 261 |
| stdint.h | 261 |
| stdio.h | 261 |
| stdlib.h | 261 |
| wchar.h | 261 |
| wctype.h | 262 |
| Segment reference | 263 |
| Summary of segments | 263 |
| Descriptions of segments | 264 |
| Implementation-defined behavior | 283 |
| Descriptions of implementation-defined behavior | 283 |
| Translation | 283 |
| Environment | 284 |
| Identifiers | 284 |
| Characters | 284 |
| Integers | 286 |
| Floating point | 286 |
| Arrays and pointers | 287 |
| Registers | 287 |
| Structures, unions, enumerations, and bitfields | 287 |
| Qualifiers | 288 |
| Declarators | 288 |
| Statements | 288 |
| Preprocessing directives | 288 |
| IAR DLIB Library functions | 290 |

| | |
|-------------|-----|
| Index | 295 |
|-------------|-----|

Tables

| | |
|---------------------------------------------------------------------------|-----|
| 1: Typographic conventions used in this guide | xx |
| 2: Configuration dependencies | 7 |
| 3: Command line options for specifying library and dependency files | 9 |
| 4: Data model characteristics | 15 |
| 5: Memory types and their corresponding memory attributes | 18 |
| 6: Code models | 26 |
| 7: XLINK segment memory types | 34 |
| 8: Memory layout of a target system (example) | 35 |
| 9: Segment name suffixes | 38 |
| 10: Heaps, memory types, and segments | 42 |
| 11: Library configurations | 53 |
| 12: Levels of debugging support in runtime libraries | 54 |
| 13: Prebuilt libraries | 55 |
| 14: Customizable items | 56 |
| 15: Formatters for printf | 58 |
| 16: Formatters for scanf | 59 |
| 17: Descriptions of printf configuration symbols | 68 |
| 18: Descriptions of scanf configuration symbols | 69 |
| 19: Low-level I/O files | 70 |
| 20: Heaps and memory types | 75 |
| 21: Functions with special meanings when linked with debug info | 75 |
| 22: Example of runtime model attributes | 77 |
| 23: Predefined runtime model attributes | 78 |
| 24: Registers used for passing parameters | 94 |
| 25: Registers used for returning values | 96 |
| 26: Specifying the size of an assembler memory instruction | 98 |
| 27: Call frame information resources defined in a names block | 101 |
| 28: Compiler optimization levels | 118 |
| 29: Environment variables | 136 |
| 30: Error return codes | 139 |
| 31: Compiler options summary | 146 |

| | |
|-------------------------------------------------------------------------|-----|
| 32: Available code models | 149 |
| 33: Available data models | 151 |
| 34: Integer types | 176 |
| 35: Floating-point types | 178 |
| 36: Function pointers | 179 |
| 37: Data pointers | 180 |
| 38: Extended keywords summary | 203 |
| 39: <code>__code16</code> function memory attribute | 206 |
| 40: <code>__code24</code> function memory attribute | 207 |
| 41: <code>__data8</code> data memory attribute | 207 |
| 42: <code>__data16</code> data memory attribute | 208 |
| 43: <code>__data32</code> data memory attribute | 209 |
| 44: Pragma directives summary | 215 |
| 45: Intrinsic functions summary | 227 |
| 46: Predefined symbols summary | 246 |
| 47: Traditional standard C header files—DLIB | 257 |
| 48: Embedded C++ header files | 257 |
| 49: Additional Embedded C++ header files—DLIB | 258 |
| 50: Standard template library header files | 258 |
| 51: New standard C header files—DLIB | 259 |
| 52: Segment summary | 263 |
| 53: Message returned by <code>strerror()</code> —IAR DLIB library | 292 |

Preface

Welcome to the H8 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 H8 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 H8/300H and H8S microcomputer families and need to get detailed reference information on how to use the H8 IAR C/C++ Compiler. In addition, you should have a working knowledge of the following:

- The architecture and instruction set of the H8/300H and H8S microcomputer families. Refer to the documentation from Renesas for information about the H8/300H and H8S microcomputer families
- 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 H8 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 *H8 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 *H8 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 H8 IAR C/C++ Compiler for efficiently developing your application.
- *Data storage* describes how data can be stored in memory, with emphasis on the different data models and data memory type attributes.
- *Functions* gives a brief overview of function-related extensions—mechanisms for controlling functions—and describes some of these mechanisms in more detail.
- *Placing code and data* describes the concept of segments, introduces the linker 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.
- *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

- *Compiler usage* provides reference information about how the compiler interacts with its environment—the invocation syntax, methods for passing options to the compiler, environment variables, the include file search procedure, and the different types of compiler output. The chapter also describes how the compiler’s diagnostic system works.
- *Compiler options* explains how to set the compiler options, gives a summary of the options, and contains detailed reference information for each compiler option.
- *Data representation* describes the available data types, pointers, and structure types.
- *Compiler extensions* gives a brief overview of the compiler extensions to the ISO/ANSI C standard. More specifically the chapter describes the available C language extensions.
- *Extended keywords* gives reference information about each of the H8-specific keywords that are extensions to the standard C/C++ language.
- *Pragma directives* gives reference information about the pragma directives.
- *Intrinsic functions* gives reference information about the functions that can be used for accessing H8-specific low-level features.

- *The preprocessor* gives a brief overview of the preprocessor, including reference information about the different preprocessor directives, symbols, and other related information.
- *Library functions* gives an introduction to the C or C++ library functions, and summarizes the header files.
- *Segment reference* gives reference information about the compiler's use of segments.
- *Implementation-defined behavior* describes how the H8 IAR C/C++ Compiler handles the implementation-defined areas of the C language standard.

Other documentation

The complete set of IAR Systems development tools for the H8/300H and H8S microcomputer families 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 *H8 IAR Embedded Workbench® IDE User Guide*
- Programming for the H8 IAR Assembler, refer to the *H8 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
- Porting application code and projects created with a previous H8 IAR Embedded Workbench IDE, refer to *H8 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 Renesas website, www.Renesas.com, contains information and news about the H8/300H and H8S microcomputer families.
- 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:




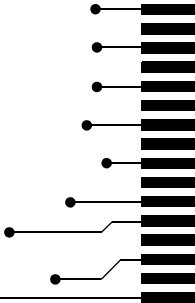
| Style | Used for |
|-------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|
| computer | Text that you enter or that appears on the screen. |
| <i>parameter</i> | A label representing the actual value you should enter as part of a command. |
| [option] | An optional part of a command. |
| {option} | A mandatory part of a command. |
| a b c | Alternatives in a command. |
| bold | Names of menus, menu commands, buttons, and dialog boxes that appear on the screen. |
| <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 H8 IAR C/C++ Compiler Reference Guide includes the following chapters:

- Getting started
- Data storage
- Functions
- Placing code and data
- The DLIB 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 H8 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 H8/300H and H8S microcomputer families. 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 H8 IAR C/C++ Compiler:

- C, the most widely used high-level programming language used in the embedded systems industry. Using the H8 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 *H8 IAR Assembler Reference Guide*.

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

Supported H8/300H and H8S derivatives

The H8 IAR C/C++ Compiler supports all derivatives based on the standard Renesas H8/300H and H8S microcomputer families.

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 H8 IAR C/C++ Compiler or the H8 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 *H8 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.r37` using the default settings:

```
icch8 myfile.c
```

In addition, you need to specify some critical options, see *Basic settings for project configuration*, page 5. For more information about the compiler invocation syntax and about how the compiler relates to its environment, see *Compiler usage*, page 135.

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.r37 myfile2.r37 -s __program_start -f lnkh8hss.xcl
dlh8hssfn.r37 -o aout.a37 -r
```

In this example, `myfile.r37` and `myfile2.r37` are object files, `lnkh8hss.xcl` is the linker command file, and `dlh8hssfn.r37` is the runtime library. The option `-s` specifies the label where the application starts. The option `-o` specifies the name of the output file, and the option `-r` is used for specifying the output format UBROF, which can be used for debugging in C-SPY.

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

Basic 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 H8/300H and H8S 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 *Compiler options syntax*, page 143, and the *H8 IAR Embedded Workbench® IDE User Guide*, respectively.

The basic settings are:

- Core—H8/300H or H8S
- Operating mode—Normal or Advanced
- Data model—Small or Huge
- Code model—Small or Large
- Hardware configuration—use of MAC register, interrupt mode, and bus width
- Size of `double` floating-point type—32-bit or 64-bit doubles
- Optimization settings
- Runtime environment.



If you are using the IAR Embedded Workbench IDE, you also have the option to select a derivative. A linker command file (`xc1`) and a C-SPY device description file (`ddf`) will automatically be selected based on your choice of derivative. In addition, settings for hardware configuration may be provided automatically.

In addition to all 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.

CORE

The H8 IAR C/C++ Compiler supports both the H8/300H and H8S microcomputer families. Use the `--core` option to select the processor core for which the code is to be generated.

This option has implications for the available operating modes, and the data and code model, see *Configuration dependencies*, page 7.

OPERATING MODE

The H8 IAR C/C++ Compiler supports the different operating modes—Normal and Advanced—available in the H8/300H and H8S microcomputer families. Use the `--operating_mode` option to specify which operating mode you are using.

This option has implications for the used core, and the data and code model, see *Configuration dependencies*, page 7.

DATA MODEL

One of the characteristics of the H8/300H and H8S microcomputer families is that there is a trade-off regarding the way memory is accessed, ranging from cheap access to small memory areas, up to more expensive methods that can access any location.

In the H8 IAR C/C++ Compiler, you can set a default memory access method by selecting a data model. The following data models are supported:

- The Small data model, which has a default data pointer size of 2 bytes and which can access the low 32 Kbytes or top 32 Kbytes of memory. In the Normal operating mode, this is equivalent to the entire 64 Kbytes of memory
- The Huge data model, which has a default data pointer size of 4 bytes and which can access the entire memory area. This model is only available in the Advanced operating mode.

However, it is possible to override the default access method for each individual variable. The chapter *Data storage* covers data models in greater detail. The chapter also covers how to fine-tune the access method for individual data objects.

This option has implications for the used core, operating mode, and the code model, see *Configuration dependencies*, page 7.

CODE MODEL

The H8 IAR C/C++ Compiler supports *code models* to control the size of function addresses, which determines the possible memory range for storing the function. The following code models are available:

- The Small code model, which has a default function pointer size of 2 bytes; functions can be placed in the memory range $0x2-0xFFFF$; this code model is available when using the Normal operating mode.
- The Large code model, which has a default function pointer size of 4 bytes; functions can be placed in the memory range $0x2-0xFFFFFFFF$; this code model is available when using the Advanced operating mode.

This option has implications for the used core, operating mode, and the data model, see *Configuration dependencies*, page 7.

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

HARDWARE CONFIGURATION

Some options specify the hardware properties of the selected microcomputer, or how you set up your microcomputer for your application. These are:

- The availability of the MAC register and the MAC instructions; see *--enable_mac*, page 157
- The interrupt mode; many devices support several different interrupt modes. The interrupt mode is selected by programming special function registers. When you generate code related to interrupt handling, you must specify the interrupt mode you will use; see *--interrupt_mode*, page 159
- The used address bus width of the device you are using; see *--bus_width*, page 148.

CONFIGURATION DEPENDENCIES

The following table lists the available combinations of the settings of core, operating mode, data model, and code model:

| Core | Operating mode | Data model | Code model |
|---------|----------------------------|-------------|------------|
| H8S | Normal: 16-bit addresses | Small | Small |
| H8S | Advanced: 32-bit addresses | Small, Huge | Large |
| H8/300H | Normal: 16-bit addresses | Small | Small |
| H8/300H | Advanced: 24-bit addresses | Small, Huge | Large |

Table 2: *Configuration dependencies*

SIZE OF DOUBLE FLOATING-POINT TYPE

Floating-point values are represented by 32- and 64-bit numbers in standard IEEE754 format. By using the compiler option `--double={32|64}`, 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 H8 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 unrolling and 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 118. For more information about efficient coding techniques, see the chapter *Efficient coding for embedded applications*.

RUNTIME ENVIRONMENT

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

The runtime library provided is 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 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 that you can use for building your own library version. This gives you full control of the runtime environment. If your project only contains assembler source code, there is no need to choose a runtime library.

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

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



Choosing a runtime library in the IAR Embedded Workbench IDE

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

Note that for the DLIB library there are two different configurations—Normal and Full—which include different levels of support for locale, file descriptors, multibyte characters, et cetera. See *Library configurations*, page 53, 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\h8\inc</code> | Specifies the include paths |
| <code>libraryfile.r37</code> | Specifies the library object file |
| <code>--dlib_config</code> <code>C:\..\configfile.h</code> | Specifies the library configuration file |

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

For a list of all prebuilt library object files for the IAR DLIB Library, see Table 13, *Prebuilt libraries*, page 55. The table also shows how the object files correspond to the dependent project options, and the corresponding configuration files. Make sure to use the object file that matches your other project options.

Setting library and runtime environment options

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

- The formatters used by the functions `printf`, `scanf`, and their variants, see *Choosing formatters for printf and scanf*, page 57.
- The size of the stack and the heap, see *The stack*, page 40, and *The heap*, page 41, respectively.

Special support for embedded systems

This section briefly describes the extensions provided by the H8 IAR C/C++ Compiler to support specific features of the H8/300H and H8S microcomputer families.

EXTENDED KEYWORDS

The H8 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 data objects as well as for declaring special function types.



By default, language extensions are enabled in the IAR Embedded Workbench IDE.

The command line option `-e` makes the extended keywords available, and reserves them so that they cannot be used as variable names. See, `-e`, page 156 for additional information.

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

PRAGMA DIRECTIVES

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

The pragma directives are always enabled in the H8 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, and the code and data models.

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

SPECIAL FUNCTION TYPES

The special hardware features of the H8/300H and H8S microcomputer families are supported by the compiler's special function types: interrupt, monitor, task, and trap. You can write a complete application without having to write any of these functions in assembler language.

For detailed information, see *Primitives for interrupts, concurrency, and OS-related programming*, page 26.

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 `h8\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 130.

To read about using I/O files for version 1 of the compiler and for files delivered by Renesas, see the *H8 IAR Embedded Workbench® Migration Guide*.

ACCESSING LOW-LEVEL FEATURES

For hardware-related parts of your application, accessing low-level features is essential. The H8 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 83.

Data storage

This chapter gives a brief introduction to the memory layout of the H8/300H and H8S microcomputer families 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, H8 IAR C/C++ Compiler provides a set of data models and data memory attributes, allowing you to fine-tune the access methods, resulting in smaller code size. The concepts of data models and memory types are described in relation to pointers, structures, Embedded C++ class objects, and non-initialized memory. Finally, detailed information about data storage on the stack and the heap is provided.

Introduction

The H8 IAR C/C++ Compiler supports using the H8/300H and H8S microcomputers in both Normal and Advanced operating mode.

In the Normal operating mode, all data must be placed within 64 Kbytes of memory, in the address range `0x0000` to `0xFFFF`.

In the Advanced operating mode, the microcomputer can address more memory, and data can be placed in the address range `0x000000` to `0xFFFFF` (H8/300H) or `0x00000000` to `0xFFFFFFFF` (H8S). Note that a specific device may impose additional limits. The specified address ranges are valid for a device without a bus width limitation, that is, a 24-bit bus width for an H8/300H device, or a 32-bit bus width for an H8S device.

In practice, many devices have a more narrow bus width. For example, an H8S device typically does not implement more than 24 address bits. In this case, the largest possible address is `0xFFFFF` even for an H8S device.

By specifying the bus width when compiling in the Advanced operating mode, you provide the compiler with information about the address range of the device. For example, by specifying a bus width of 24 address bits, the compiler recognizes an address in the address range `0xFF8000–0xFFFFFFFF` to be accessible with the `@aa:16` address mode.

Different types of physical memory can be placed in the memory range. A typical application will have both read-only memory (ROM) and read/write memory (RAM). Typically, there is no ROM at the highest 256 bytes in memory. Some parts of the memory range contain processor control registers and peripheral units.

The compiler can access memory in different ways. The access methods range from generic but expensive methods that can access the full memory space, to cheap methods that can access limited memory areas. The compiler supports this by means of data models and data memory attributes.

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.

Data models

The H8 IAR C/C++ Compiler supports two data models that can be used for applications with different data requirements.

Technically, the data model specifies the default memory type. This means that the data 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 default addressing mode used by the compiler to access variables
- The placement of the runtime stack.

The data model 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 18.

SPECIFYING A DATA MODEL

Two data models are implemented: Small, and Huge. These models are controlled by the `--data_model` option. Each model has a default memory type and a default pointer size. If you do not specify a data model option, the compiler will use the Small data model.

Your project can only use one data model at a time, and the same model must be used by all user modules and all library modules. However, you can override the default memory type by explicitly specifying a memory attribute, using either keywords or the `#pragma type_attribute` directive.

The following table summarizes the different data models:

| Data model name | Default memory attribute | Default pointer attribute | Placement of data |
|-----------------|--------------------------|---------------------------|-------------------------------------------------------------------------------------|
| Small | <code>__data16</code> | <code>__data16</code> | Low 32 Kbytes or top 32 Kbytes, which means all memory in the Normal operating mode |
| Huge | <code>__data32</code> | <code>__data32</code> | All 16 Mbytes (H8/300H) or 4 Gbytes (H8S) of memory |

Table 4: Data model characteristics

For detailed information about the memory ranges for each memory attribute, see *Descriptions of extended keywords*, page 204.



See the *H8 IAR Embedded Workbench® IDE User Guide* for information about setting options in the IAR Embedded Workbench IDE.



Use the `--data_model` option to specify the data model for your project; see *--data_model*, page 151.

Memory types

This section describes the concept of *memory types* used for accessing data by the H8 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 H8 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 space, to cheap methods that can access limited memory areas. Each memory type corresponds to one memory access method. By mapping different memories—or part of memories—to memory types, the compiler can generate code that can access data efficiently.

For example, the memory accessible using the `data16` memory access method is called memory of `data16` type, or simply `data16` memory.

By selecting a *data 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.

For more information about memory access methods, see *Memory access methods*, page 98.

BITVAR

The `bitvar` memory holds single bit variables and it can only be located in the highest 256 bytes of data memory, where the start address depends on the processor family and the operating mode you are using. This part of the memory is accessible via the addressing mode `@aa:8`.

For more details about the valid address ranges, see *__bitvar*, page 204.

DATA8 MEMORY

The `data8` memory consists of the highest 256 bytes of data memory, where the start address depends on the processor family and the operating mode you are using.

Because it is unlikely that the `data8` memory is large enough for an application to hold all default data (including the runtime stack), there is no data model that uses `data8` memory as the default memory. However, you can declare individual data objects to be placed in `data8` memory by using the `__data8` memory attribute.

The theoretical size of a data object placed in `data8` memory is limited to 254 bytes (256-2), although the practical size is normally much smaller, because special function registers are normally placed here as well.

The code generated by the compiler differs somewhat if you use the Normal operating mode or the Advanced operating mode. The most significant difference is that, although a `data8` pointer is only 8 bits in size, to dereference it, it must be expanded to 16 bits in the Normal operating mode, and expanded to 32 bits in the Advanced operating mode. This dereference problem also applies when indexing an array with a non-constant index.

Data8 memory is best used for single variables which are mainly accessed directly and not via a pointer. For such variables, the code generated by the compiler can be very efficient. This means a smaller footprint for the application, and faster execution at run-time.

For more details, see `__data8`, page 207 and *The data8 memory access method*, page 100.

DATA16 MEMORY

The data16 memory consists of the low 32 Kbytes and the high 32 Kbytes of data memory. In the Normal operating mode, this includes all available memory. The start address of the high memory area, as well as the maximum object size depend on the processor family and the operating mode you are using. This memory type is default in the Small data model.

The code generated by the compiler will differ somewhat between the Normal operating mode and the Advanced operating mode. The most significant difference is that, although a data16 pointer is only 16 bits in size, it must be expanded to 32 bits when dereferenced in the Advanced operating mode.

By using objects of this type, the code generated by the compiler to access them is fairly small. This means a small footprint for the application, and fast execution at run-time.

For more details, see `__data16`, page 208 and *The data16 memory access method*, page 99.

DATA32 MEMORY

Data32 objects can be placed anywhere in the data memory, which means that all memory up to 4 Gbytes can be accessed. Also, unlike the other memory types, there is no limitation on the size of the objects that can be placed in this memory type. Data32 memory is default in the Large data model.

The drawback of the data32 memory type is that the code generated to access the memory is larger and slower than that of any of the other memory types. In addition, the code uses more processor registers, which may force local variables to be stored on the stack rather than being allocated in registers. Note that data32 objects can only be used in the Advanced operating mode.

Note that if you use the Advanced operating mode, you may want to use the Small data model, and place the majority of data in the default low and high 32 Kbytes of memory. You can still declare individual variables with the `__data32` data memory attribute, and thus place them in any part of the memory.

For more details, see `__data32`, page 209 and *The data32 memory access method*, page 100.

USING DATA MEMORY ATTRIBUTES

The H8 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.

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

| Memory type | Memory attribute | Pointer size | Default in data model |
|-------------|-----------------------|--------------|-----------------------|
| Bitvar | <code>__bitvar</code> | N/A* | -- |
| Data8 | <code>__data8</code> | 8 bits | -- |
| Data16 | <code>__data16</code> | 16 bits | Small |
| Data32 | <code>__data32</code> | 32 bits | Large |

Table 5: Memory types and their corresponding memory attributes

* **You cannot take the address of a `__bitvar` variable, nor create a pointer to such a variable.**

The address ranges for each memory attribute depends on the core, the operating mode, and the bus width you are using. The keywords are only available if language extensions are enabled in the H8 IAR C/C++ Compiler.



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



Use the `-e` compiler option to enable language extensions. See *-e*, page 156 for additional information.

For reference information about each keyword, see *Descriptions of extended keywords*, page 204.

Syntax

The keywords follow the same syntax as the type qualifiers `const` and `volatile`. The memory attributes are *type attributes* and therefore they must be specified both when they are defined and in the declaration, see *Type qualifiers*, page 183.

The following declarations place the variable `i` and `j` in data16 memory. The variables `k` and `l` behave in the same way:

```
__data16 int i, j;
int __data16 k, l;
```

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

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

Type definitions

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

```
typedef char __data16 Byte;
typedef Byte *BytePtr;
Byte b;
BytePtr bp;
```

and

```
__data16 char b;
char __data16 *bp;
```

POINTERS AND MEMORY TYPES

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

In the H8 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 data16 memory is declared by:

```
int __data16 * p;
```

Note that the location of the pointer variable `p` is not affected by the keyword. In the following example, however, the pointer variable `p2` is placed in data32 memory. Like `p`, `p2` points to an integer in data16 memory.

```
int __data16 * __data32 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 the H8 IAR C/C++ Compiler, the size of the `data8`, `data16`, and `data32` pointers are 8, 16, and 32 bits, respectively.

In the H8 IAR C/C++ Compiler, it is illegal, with one exception, to convert pointers between different types without explicit casts. The exception is when casting from a small pointer type to a larger pointer type. Because the pointer size is 8 bits for data8 pointers, 16 bits for data16 pointers, and 32 bits for data32 pointers, and data8 resides in data16 and data32, it is legal to cast a data8 pointer to a data16 or data32 pointer without an explicit cast. In the same way it is legal to cast a data16 pointer to a data32 pointer without an explicit cast.

STRUCTURES AND MEMORY TYPES

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

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

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

The following declaration is incorrect:

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

MORE EXAMPLES

The following is a series of examples with descriptions. First, some integer variables are defined and then pointer variables are introduced. Finally, a function accepting a pointer to an integer in data16 memory is declared. The function returns a pointer to an integer in data32 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 __data16 b;</code> | A variable in data16 memory. |
| <code>__data32 int c;</code> | A variable in data32 memory. |
| <code>int * d;</code> | A pointer stored in default memory. The pointer points to an integer in default memory. |

| | |
|--------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>int __data32 * e;</code> | A pointer stored in default memory. The pointer points to an integer in data32 memory. |
| <code>int __data16 * __data32 f;</code> | A pointer stored in data32 memory pointing to an integer stored in data16 memory. |
| <code>int __data32 * myFunction(int __data16 *);</code> | A declaration of a function that takes a parameter which is a pointer to an integer stored in data16 memory. The function returns a pointer to an integer stored in data32 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 105—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.

In the Small data model, this means that objects of classes with a member function can only be placed in the default memory type (`__data16`).

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 data32 memory.

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

    __data32 static int gamma;
};

// Definitions needed (should be placed in a source file):
__data32 int MyClass::gamma;
```

```
// A variable definition:
__data16 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 current function, the function that called the current function, and the function that called it, etc. The second part contains free memory that can be allocated. The borderline between the two areas is called the *top of stack* and is represented by the stack pointer, which is a dedicated processor register. Memory is allocated on the stack by moving the stack pointer.

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

Advantages

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

It is possible for a function to call itself—a so-called *recursive function*—and each invocation can store its own data on the stack.

Potential problems

The way the stack works makes it impossible to store data that is supposed to live after the function has returned. The following function demonstrates a common programming mistake. It returns a pointer to the variable `x`, a variable that ceases to exist when the function returns.

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

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

Dynamic memory on the heap

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

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

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 H8 IAR C/C++ Compiler supports heaps in both `data16` memory and `data32` memory. For more information about this, see *The heap*, page 41.

Potential problems

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

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

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

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

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

Functions

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

Function-related extensions

In addition to the ISO/ANSI C standard, the H8 IAR C/C++ Compiler provides several extensions for writing functions in C. Using these, you can:

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

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

For more information about optimizations, see *Writing efficient code*, page 126. For information about the available intrinsic functions for accessing hardware operations, see the chapter *Intrinsic functions*, page 227.

Code models and memory attributes for function storage

The H8 IAR C/C++ Compiler supports two *code models* to control the size of function addresses—Small and Large. Technically, the code model specifies the default *memory type*. This means that the code model controls the following:

- The possible memory range for storing the function
- The maximum module size
- The maximum program size.

The code models are controlled by the `--code_model` option. If you do not specify a code model option, the compiler will use the Small code model in the Normal operating mode and the Large code model in the Advanced operating mode.

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

The following table summarizes the different code models:

| Code model name | Default memory attribute | Max module size | Max program size |
|-----------------|--------------------------|-----------------|------------------|
| Small | <code>--code16</code> | 64 Kbytes | 64 Kbytes |
| Large | <code>--code24</code> | 16 Mbytes | 16 Mbytes |

Table 6: Code models

The code model must follow the operating mode strictly. In the Normal operating mode, only the Small code model is available. In the Advanced operating mode, only the Large code model is available. This means that you have no real choice when it comes to code models. The code model option has primarily been added for future compatibility.

For detailed information about the memory ranges for each memory attribute, see *Descriptions of extended keywords*, page 204.



See the *H8 IAR Embedded Workbench® IDE User Guide* for information about specifying a code model in the IAR Embedded Workbench IDE.



Use the `--code_model` option to specify the code model for your project; see *--code_model*, page 149.

Primitives for interrupts, concurrency, and OS-related programming

The H8 IAR C/C++ Compiler provides the following primitives related to writing interrupt functions, concurrent functions, and OS-related functions:

- The extended keywords `__interrupt`, `__task`, `__trap`, `__raw`, and `__monitor`
- The pragma directive `#pragma vector`
- The intrinsic functions `__enable_interrupt`, `__disable_interrupt`, and many other related intrinsic functions.

INTERRUPT FUNCTIONS

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

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

The H8/300H and H8S microcomputer families support many interrupt sources. For each interrupt source, an interrupt routine can be written. Each interrupt routine is associated with a vector number, which is specified in the H8/300H and H8S microcomputer documentation from the chip manufacturer. The interrupt vector corresponds to the vector number in the device documentation from Renesas. If you want to handle several different interrupts using the same interrupt function, you can specify several interrupt vectors.

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=_INT_WOVI //Symbol defined in I/O header file
__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 H8/300H and H8S microcomputer documentation for more information about the interrupt vector table.

Note that interrupt vectors are 16 bits in the Normal operating mode and 32 bits in the Advanced operating mode.

TRAP FUNCTIONS

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

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

Each trap function is associated with a vector. The vector value ranges from 0 to 3.

The `__trap` keyword and the `#pragma vector` directive can be used to define trap functions.

For example, the following piece of code defines a function doubling its argument:

```
#pragma vector=2
__trap int twice(int x)
{
    return x + x;
}
```

When a trap function is defined with a vector, the trap vector entry in the processor interrupt vector table is populated. It is also possible to define a trap function without a vector. This is useful if an application is capable of populating or changing the trap vectors at runtime. For more information about the interrupt vector table, see the chip manufacturer's documentation for the H8/300H and H8S microcomputer families.

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

MONITOR FUNCTIONS

A monitor function causes interrupts to be disabled during execution of the function. At function entry, the interrupt state is saved and interrupts are disabled. At function exit, the original interrupt state is restored.

To define a monitor function, you can use the `__monitor` keyword. For reference information, see *__monitor*, page 211.

Note: If you specify one interrupt mode when compiling your code, but set up your device to use another interrupt mode, interrupts will not be correctly disabled for monitor functions. This means that it is critical to compile your application code specifying the interrupt mode that is used in your application; see *--interrupt_mode*, page 159.

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 H8 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 <intrinsics.h>

volatile long tick_count = 0;

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

    ~Mutex ()
    {
        __set_interrupt_state(_state);
    }

private:
    __istate_t _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();
    }
}
```

Note: You can also implement a semaphore by using the intrinsic function `__TAS`.

C++ AND SPECIAL FUNCTION TYPES

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

- Interrupt member functions must be static. When 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.
- Trap member functions cannot be declared virtual. The reason for this is that trap functions cannot be called via function pointers.

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 H8 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 H8 IAR C/C++ Compiler uses only the following XLINK segment memory types:

| Segment memory type | Description |
|---------------------|--------------------------------|
| BIT | For bit variables; in RAM only |
| CODE | For executable code |
| CONST | For data placed in ROM |
| DATA | For data placed in RAM |

Table 7: XLINK segment memory types

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

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.

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 `config` directory contains ready-made linker command files for all supported devices. The files contain the information required by the linker, and is ready to be used. The only change you will normally have to make to the supplied linker command file is to customize it so it fits the target system memory map. If, for example, your application uses additional external RAM, you need to add details about the external RAM memory area. In the `src\template` directory, you can find more examples of generic linker command files.

As an example, we can assume that the target system has a 24-bit bus width and the following memory layout:

| Range | Type of memory | Holding segments of segment memory type |
|---------------------|----------------|-----------------------------------------|
| 0x000000–0x03FFFF | ROM | CODE and CONST |
| 0xFF0000–0xFFFFFFFF | RAM | DATA |

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

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

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

The contents of the linker command file

The linker command file is an extended command line, which means you can use the file to specify any linker option. Typically, the linker command file contains three different types of linker command line options:

- The CPU used:
`-ch8`
 This specifies your target microcomputer.
- 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.

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

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

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

Using the `-Z` command for sequential placement

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

The following illustrates how to use the `-Z` command to place the segment `MYSEGMENTA` followed by the segment `MYSEGMENTB` in `CONST` memory (that is, ROM) in the memory range `0x2000-0xCFFF`.

```
-Z (CONST) MYSEGMENTA, MYSEGMENTB=0-3FFFF
```

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 (CONST) MYSEGMENTA=0-3FFFF
-Z (CODE) MYCODE
```

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

```
-Z (CONST) MYSMALLSEGMENT=0-FFFF
-Z (CONST) MYLARGESEGMENT=0-3FFFF
```



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.

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 a fictitious memory range:

```
-P (DATA) MYDATA=FF0000-FFFFFFF
```


If your application has an additional RAM area in the memory range 0xFC0000-0xFCFFFF, you just add that to the original definition:

```
-P (DATA) MYDATA=FF0000-FFFFFF, FC0000-FCFFFF
```

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

Data segments

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

To get a clear understanding about how the data segments work, you must be familiar with the different memory types and the different data models available in the H8 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 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, `DATA16_Z`. The segment base names are derived from the memory type attributes, for example:

```
__data16
```

would yield the segment base name `DATA16`.

Some of the declared data is placed in non-volatile memory, for example ROM, and some of the data is placed in RAM. For this reason, it is also important to know the XLINK segment memory type of each segment. For more details about segment memory types, see *Segment memory type*, page 34.

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

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

Table 9: Segment name suffixes

For a summary of all supported segments, see *Summary of segments*, page 263.

Examples

Assume the following examples:

```
__data16 int j;           The data16 variables that are to be initialized to zero
__data16 int i = 0;      when the system starts will be placed in the segment
                          DATA16_Z.

__no_init __data16 int j; The data16 non-initialized variables will be placed in
                          the segment DATA16_N.

__data16 int j = 4;      The data16 non-zero initialized variables will be
                          placed in the segment DATA16_I, and initializer data
                          in segment DATA16_ID.
```

Initialized data

In ISO/ANSI C all static variables—variables that are allocated to a fixed memory address—have to be initialized by the runtime 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 addition, the H8 IAR C/C++ Compiler provides an extension to this rule—variables declared with the keyword `__no_init` are not initialized at all—for more information, see `__no_init`, page 211.

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:

```
DATAI6_I           0x1000-0x10FF and 0x1200-0x12FF
DATAI6_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:

```
DATAI6_I           0x1000-0x10FF and 0x1200-0x12FF
DATAI6_ID          0x4000-0x40FF and 0x4200-0x42FF
```

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

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

Data segments for static memory in the default linker command file

The default linker command file contains directives similar to the following directives to place the static data segments. For details, study the appropriate linker command file for the selected combination of core, code model, and data model.

First, the segments to be placed in ROM are defined:

```
-Z (CONST) DATA16_C=2-7FFF
-Z (CONST) DATA32_C=2-7FFFFFFF
-Z (CONST) DATA8_ID, DATA16_ID, DATA32_ID=2-7FFFFFFF
```

Note: Typically, there is no ROM memory at the highest 256 bytes of memory, which means there is no `DATA8_C` segment. Constant data declared `__data8` is placed in the `DATA8_I` segment instead.

Then, the RAM data segments are placed in memory:

```
-Z (DATA) DATA8_I, DATA8_Z, DATA8_N=FFFFFFF01-FFFFFFFE
-Z (DATA) DATA16_I, DATA16_Z, DATA16_N=FFFF8000-FFFFFFFE
-Z (DATA) DATA32_I, DATA32_Z, DATA32_N=80000000-FFFFFFFE
```

THE STACK

The stack is used by functions to store variables and other information that is used locally by functions, as described in the chapter *Data storage*. It is a continuous block of memory pointed to by the processor stack pointer register `ER7`.

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

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



Stack size allocation in the IAR Embedded Workbench IDE

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

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



Stack size allocation from the command line

The size of the `CSTACK` segment is defined in the linker command file (alternatively, from the command line).

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

```
-D_CSTACK_SIZE=size
```

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

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



Placement of stack segment

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

```
-Z (DATA) CSTACK+_CSTACK_SIZE#FF0000-FFFFFF
```

Note:

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



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.

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:

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

Heap segments in DLIB

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

```
__data16_malloc
```

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

Each heap will reside in a segment with the name `_HEAP` prefixed by a segment base name derived from the memory attribute.

Heaps can be placed in the following memory types:

| Memory type | Segment name | Memory attribute |
|-------------|--------------|------------------|
| Data16 | DATA16_HEAP | __data16 |
| Data32 | DATA32_HEAP | __data32 |

Table 10: Heaps, memory types, and segments

Note that the data32 heap is only available in the Advanced operating mode.



Heap size allocation in the IAR Embedded Workbench IDE

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

Add the required heap sizes in the **Heap size** text boxes.



Heap size allocation from the command line

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

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

```
-D_DATA16_HEAP_SIZE=size
-D_DATA32_HEAP_SIZE=size
```

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

Specify the appropriate size for your application.



Placement of heap segment

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

```
-Z (DATA) DATA16_HEAP+_DATA16_HEAP_SIZE=FF8000-FFFFFF
-Z (DATA) DATA32_HEAP+_DATA32_HEAP_SIZE=FF0000-FFFFFF
```

Note: The ranges do not specify the size of the heaps; they specifies the range of the available memory.



Heap size and standard I/O

If you have excluded `FILE` descriptors from the `DLIB` runtime environment, as in the normal configuration, there are no input and output buffers at all. Otherwise, as in the full configuration, be aware that the size of the input and output buffers is set to 512 bytes in the `stdio` library header file. If the heap is too small, I/O will not be buffered, which is considerably slower than when I/O is buffered. If you execute the application using the 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 H8/300H and H8S device. 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 `SEGMENTBASENAME_AC` or the `SEGMENTBASENAME_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 does not have to be specified in the linker command file.

Code segments

This section contains descriptions of the segments used for storing code, and the interrupt vector table. For a complete list of all segments, see *Summary of segments*, page 263.

NORMAL CODE

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

If you use the Small code model, the code is placed in the `CODE16` segment. If you use the Large code model, the code is placed in the `CODE24` segment.

In the default linker command file it can look like this:

```
-P (CODE) CODE24=2-7FFFFFFF
```

For the code segments, the `-P` linker directive is used for allowing `XLINK` to split up the segments and pack their contents more efficiently. This is useful if the memory range is non-consecutive, or if located data exists in the memory range, like interrupt vectors.

INTERRUPT VECTORS

The interrupt vector table contains pointers to interrupt routines, including the reset routine. The table is placed in the segment `INTVEC`. For the H8/300H and H8S microcomputer families, you must place this segment on the address `0x0`. The linker directive would then look like this:

```
-Z (CONST) INTVEC=0-3FF
```

The last address for the interrupt vector segment depends on the peripheral devices used in your hardware configuration. The value specified in this example is probably high enough, but may need to be changed in rare situations.

FUNCTION VECTORS FOR NON-INTERRUPT FUNCTIONS

The H8/300H and H8S microcomputers allow functions to be called with a short memory indirect call (addressing mode `@aa:8`). By using this feature of the architecture, the code can be made more compact when calling commonly used functions. This implies that these functions need a function vector; this vector is stored in the memory range `0-0xFF`.

By default, the compiler uses this addressing mode for many of the DLIB runtime library functions. In addition, you can declare functions with the extended keyword `__vector_call`. The compiler calls this type of functions with the `@aa:8` addressing mode. The compiler also automatically creates the required function vector.

Both the vectors for the library functions and your own `__vector_call` declared functions are automatically inserted in the segment `FLIST`. This segment must be defined in the linker command file and the linker directive can look like this:

```
-P (CONST) FLIST=0-FF
```

Note that the linker command file defines the `INTVEC` segment before the `FLIST` segment. This means that interrupt routines are assigned vector entries first, and then the functions with entries in the `FLIST` segment uses any free entries in the vector area.

If the vector area is too small to hold all interrupt and function vectors, you can reduce the number of vectors in three ways:

- Reduce the number of `__vector_call` declared functions.
- Use the compiler option `--direct_library_calls`. In this case, the compiler will use normal function calls for all library functions. Note that, for full effect, you must use this option for all your code that uses library functions.
- By default, the DLIB runtime library is not compiled with the `--direct_library_calls` option. To remove all vectors for the library functions, you can rebuild the library using this option.

Note: In general, the `@aa:8` addressing mode cannot be used for all functions in your application, as the number of vectors is limited.

C++ dynamic initialization

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

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

For example:

```
-Z (CONST) DIFUNCT=2-7FFFFFFF
```

For additional information, see *DIFUNCT*, page 281.

Controlling data and function placement in memory

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

- Code and data models

Use the different compiler options for code and data models, respectively, to take advantage of the different addressing modes available for the microcomputer and thereby also place functions and data objects in different parts of memory. To read more about data and code models, see *Data models*, page 14, and *Code models and memory attributes for function storage*, page 25, respectively.

- Memory attributes

Use memory attributes to override the default addressing mode and placement of individual data objects. To read more about memory attributes for data, see *Using data memory attributes*, page 18.

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

Use the `@` operator or the `#pragma location` directive to place individual global and static variables at absolute addresses. The variables must be declared either `__no_init` or `const`. This is useful for individual data objects that must be located at a fixed address, for example variables with external requirements, or for populating any hardware tables similar to interrupt vector tables. Note that it is not possible to use this notation for absolute placement of individual functions.

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

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

At compile time, data and functions are placed in different segments as described in *Data segments*, page 37, and *Code segments*, page 43, respectively. At link time, one of the most important functions of the linker is to assign load addresses to the various segments used by the application. All segments, except for the segments holding absolute located data, are automatically allocated to memory according to the specifications of memory ranges in the linker command file, as described in *Placing segments in memory*, page 34.

DATA PLACEMENT AT AN ABSOLUTE LOCATION

The @ operator, alternatively the #pragma location directive, can be used for placing global and static variables at absolute addresses. The variables must be declared either `__no_init` or `const`. If declared `const`, they can have initializers or an initializer can be omitted. If you omit the initializer, the runtime system will not provide a value at that address. To place a variable at an absolute address, the argument to the @ operator and the #pragma location directive should be a literal number, representing the actual address. The absolute location must fulfill the alignment requirement for the variable that should be located.

C++ static member variables can be placed at an absolute address just like any other static variable.

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.

Declaring located variables extern and volatile

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, for example, address `0x100`.

To avoid this problem and make the process the same in C and C++, you should declare these SFRs `extern`, for example:

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

For information about `volatile` declared objects, see *Protecting simultaneously accessed variables*, page 136.

Examples

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

```
__no_init char alpha @ 0xFF2000; /* OK */
```

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

```
#pragma location=0xFF2002
volatile const int beta; /* OK */

volatile const int gamma @ 0xFF2004 = 3; /* OK */
```

In the following example, the value is not initialized by the compiler; the value must be set by other means. The typical use is for configurations where the values are loaded to ROM separately, or for special function registers that are read-only. To force the compiler to read the value, declare it `volatile`:

```
volatile __no_init const char c @ 0xE004;
void foo(void)
{
    ...
    a = b + c + d;
    ...
}
```

The following examples show incorrect usage:

```
int delta @ 0xFF2006; /* Error, neither */
/* "__no_init" nor "const".*/

const int epsilon @ 0xFF2007; /* Error, misaligned. */
```

DATA AND FUNCTION PLACEMENT IN SEGMENTS

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

C++ static member variables can be placed in named segments just like any other static variable.

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

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

Also note that user-defined `const` segments placed in RAM are not initialized by the compiler. If initialization is required, you must do that yourself.

For more information about segments, see the chapter *Placing code and data*.

Examples of placing variables in named segments

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

```
__no_init int alpha @ "MYSEGMENT"; /* Placed in default memory */

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

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

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

```
__data32 __no_init int alpha @ "MYSEGMENT"; /* Placed in data32 */
```

The following example shows incorrect usage:

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

Examples of placing functions in named segments

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

void g(void) @ "MYSEGMENT"
{
}

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

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 the Embedded Workbench IDE, or the option `-x` on the command line, and one of their suboptions to generate a linker listing.

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

For further information about the listing options and the linker listing, see the *IAR Linker and Library Tools Reference Guide*, and the *H8 IAR Embedded Workbench® IDE User Guide*.

The DLIB runtime environment

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

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

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

Introduction to the runtime environment

The runtime environment is the environment in which your application executes. The runtime environment depends on the target hardware, the software environment, and the application code. The 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 `h8\lib` and `h8\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
- Runtime environment support, that is, startup and exit code and low-level interface to some library functions.
- Special compiler support for some functions, for instance functions for floating-point arithmetics

Some parts, like the startup and exit code and the size of the heaps and stack 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.s37`, 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
- The `FLIST` segment is too small to hold all the function vector entries, and you want to rebuild the library and use the `--direct_library_calls` compiler option to make sure function vectors are not used by library functions; see *Function vectors for non-interrupt functions*, page 44.

For information about how to build a customized library, see *Building and using a customized library*, page 61.

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 11: Library configurations

In addition to these configurations, you can define your own configurations, which means that you must modify the configuration file. Note that the library configuration file describes how a library was built and thus cannot be changed unless you rebuild the library. For further information, see *Building and using a customized library*, page 61.

The prebuilt libraries are based on the default configurations, see Table 13, *Prebuilt libraries*, page 55. 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 12: Levels of debugging support in runtime libraries

If you build your application project with the XLINK options **With runtime control modules** or **With I/O emulation modules**, certain functions in the library will be replaced by functions that communicate with the IAR C-SPY Debugger. For further information, see *C-SPY Debugger runtime interface*, page 75.



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

Using a prebuilt library

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

- Processor core
- Operating mode
- Code model
- Data model
- Size of `double`
- Library configuration—Normal or Full.

For the H8 IAR C/C++ Compiler, this means there is a prebuilt runtime library for each combination of these options. The libraries are built with size optimization 9 (High). The following table shows the mapping of the library file name, processor core, code model, data model, size of `double`, and library configuration:

| Library file name | Processor core | Code model | Data model | Size of double | Library configuration |
|-------------------|----------------|------------|------------|----------------|-----------------------|
| dlh8hssf.n.r37 | H8300H | Small | Small | 32 bits | Normal |
| dlh8hlsf.n.r37 | H8300H | Large | Small | 32 bits | Normal |
| dlh8hlhf.n.r37 | H8300H | Large | Huge | 32 bits | Normal |
| dlh8hssd.n.r37 | H8300H | Small | Small | 64 bits | Normal |
| dlh8hlsd.n.r37 | H8300H | Large | Small | 64 bits | Normal |
| dlh8hlhd.n.r37 | H8300H | Large | Huge | 64 bits | Normal |
| dlh8sssf.n.r37 | H8S | Small | Small | 32 bits | Normal |
| dlh8slsf.n.r37 | H8S | Large | Small | 32 bits | Normal |
| dlh8slhf.n.r37 | H8S | Large | Huge | 32 bits | Normal |
| dlh8sssd.n.r37 | H8S | Small | Small | 64 bits | Normal |
| dlh8slsd.n.r37 | H8S | Large | Small | 64 bits | Normal |
| dlh8slhd.n.r37 | H8S | Large | Huge | 64 bits | Normal |
| dlh8hssff.r37 | H8300H | Small | Small | 32 bits | Full |
| dlh8hlsff.r37 | H8300H | Large | Small | 32 bits | Full |
| dlh8hlhff.r37 | H8300H | Large | Huge | 32 bits | Full |
| dlh8hssdf.r37 | H8300H | Small | Small | 64 bits | Full |
| dlh8hlsdf.r37 | H8300H | Large | Small | 64 bits | Full |
| dlh8hlhdf.r37 | H8300H | Large | Huge | 64 bits | Full |
| dlh8ssdff.r37 | H8S | Small | Small | 32 bits | Full |
| dlh8slsff.r37 | H8S | Large | Small | 32 bits | Full |
| dlh8slhff.r37 | H8S | Large | Huge | 32 bits | Full |
| dlh8ssddf.r37 | H8S | Small | Small | 64 bits | Full |
| dlh8slsdf.r37 | H8S | Large | Small | 64 bits | Full |
| dlh8slhdf.r37 | H8S | Large | Huge | 64 bits | Full |

Table 13: Prebuilt libraries

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

```
<type><core><code_model><data_model><size_of_double><lib_con>.r37
```

where

- `<type>` is `d1` for the IAR DLIB runtime environment
- `<core>` is one of `h8h` and `h8s` for H8300H and H8S processor core, respectively
- `<code_model>` is one of `s` or `l` for Small and Large code model, respectively
- `<data_model>` is one of `s` or `h` for Small and Huge data model, respectively
- `<size_of_double>` is one of `f` or `d`, for 32-bit doubles or 64-bit doubles, respectively
- `<lib_con>` is one of `n` or `f` for Normal and Full library configuration, respectively.

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



The IAR Embedded Workbench IDE will include the correct library object file and library configuration file based on the options you select. See the *H8 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:
`dlh8hssfn.r37`
- Specify the include paths for the compiler and assembler:
`-I h8\inc`
- Specify the library configuration file for the compiler:
`--dlib_config C:\...\dlh8hssfn.h`

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

You can find the library object files and the library configuration files in the subdirectory `h8\lib`.

CUSTOMIZING A PREBUILT LIBRARY WITHOUT REBUILDING

The prebuilt libraries delivered with the H8 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>, page 57</i> |

Table 14: Customizable items

| Items that can be customized | Described on page |
|-------------------------------------|--------------------------------------------------------|
| Startup and termination code | <i>System startup and termination</i> , page 63 |
| Low-level input and output | <i>Standard streams for input and output</i> , page 66 |
| File input and output | <i>File input and output</i> , page 69 |
| Low-level environment functions | <i>Environment interaction</i> , page 72 |
| 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 14: Customizable items (Continued)

For a description about how to override library modules, see *Overriding library modules*, page 59.

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 68.

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> (default) | <code>_PrintfLarge</code> | <code>_PrintfSmall</code> | <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 <code>space</code> , <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 15: 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 68.



Specifying the print formatter in the IAR Embedded Workbench IDE

To specify the `printf` formatter in the IAR Embedded Workbench IDE, 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 (`_PrintfFull`), add one of the following lines in the linker command file you are using:

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

CHOOSING SCANF FORMATTER

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

The following table summarizes the capabilities of the different formatters:

| Formatting capabilities | <code>_ScanfFull</code> (default) | <code>_ScanfLarge</code> | <code>_ScanfSmall</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 |
| 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 <code>[</code> and <code>]</code> | Yes | Yes | No |
| Assignment suppressing <code>*</code> | Yes | Yes | No |
| <code>long long</code> support | Yes | No | No |

Table 16: 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 68.



Specifying `scanf` formatter in the IAR Embedded Workbench IDE

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



Specifying `scanf` formatter from the command line

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

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

Overriding library modules

The library contains modules which you probably need to override with your own customized modules, for example functions for character-based I/O and `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 `h8\src\lib` directory.

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



Overriding library modules using the IAR Embedded Workbench IDE

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

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



Overriding library modules from the command line

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

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

```
icch8 library_module
```

This creates a replacement object module file named `library_module.r37`.

- 4 Add `library_module.r37` to the XLINK command line, either directly or by using an extended linker command file, for example:

```
xlink library_module dlh8hssf.n.r37
```

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

Run XLINK to rebuild your application.

This will use your version of `library_module.r37`, 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 52, 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 *H8 IAR Embedded Workbench® IDE User Guide*.

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

SETTING UP A LIBRARY PROJECT

The IAR Embedded Workbench IDE provides a library project template which can be used for customizing the runtime environment configuration. This library template has full library configuration, see Table 11, *Library configurations*, page 53.



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

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

MODIFYING THE LIBRARY FUNCTIONALITY

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

The library functionality is determined by a set of *configuration symbols*. The default values of these symbols are defined in the file `Dlib_defaults.h`. This read-only file describes the configuration possibilities. In addition, your library has its own library configuration file `dlh8Custom.h`, which sets up that specific library with full library configuration. For more information, see Table 14, *Customizable items*, page 56.

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 `dlh8Custom.h` and customize it by setting the values of the configuration symbols according to the application requirements.

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

USING A CUSTOMIZED LIBRARY

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



In the IAR Embedded Workbench IDE you must perform the following steps:

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

System startup and termination

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

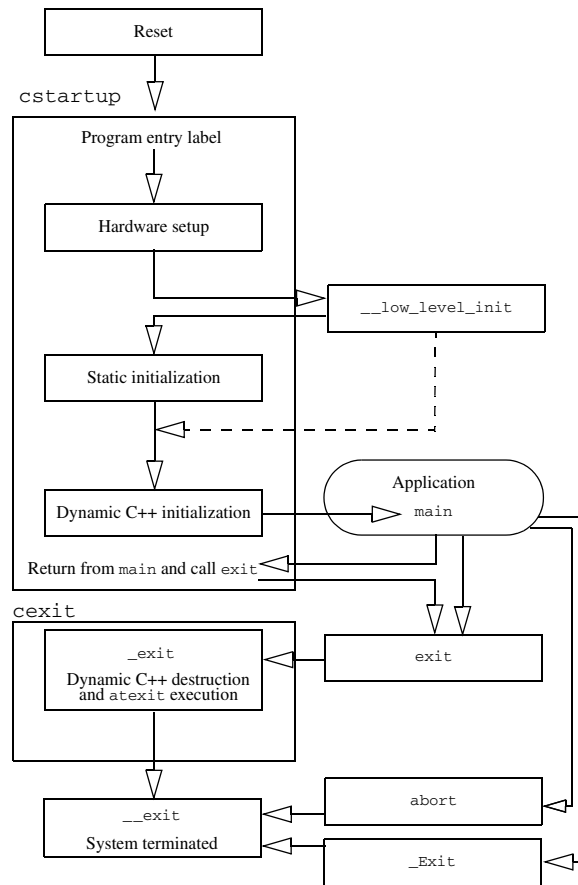


Figure 1: Startup and exit sequences

The code for handling startup and termination is located in the source files `cstartup.s37`, `cexit.s37`, and `low_level_init.c` or `low_level_init.s37` located in the `h8\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.
- The stack pointer to be used by the optional `__low_level_init` is set up to the end of the `CSTACK` segment. Note that if the required memory is not yet available in your hardware configuration, you need to modify the startup code.
- Optionally, the function `__low_level_init` is called, giving the application a chance to perform early initializations. Note that this requires a stack and thereby the previous step.
- The stack pointer to be used by your application is set up to the end of the `CSTACK` segment. This requires the memory to be available.
- Static variables are initialized; 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 assembler function `_exit` 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 75.

Customizing system initialization

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

You can do this by providing a customized version of the routine `__low_level_init`, which is called from `cstartup.s37` 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.s37`, and `low_level_init.c` or `low_level_init.s37`, located in the `h8\src\lib` directory.



If you intend to rebuild the library, the source files are available in the template library project, see *Building and using a customized library*, page 61.

Note: Regardless of whether you modify the routine `__low_level_init` or the file `cstartup.s37`, you do not have to rebuild the library.

__LOW_LEVEL_INIT

Two skeleton low-level initialization files are supplied with the product: a C source file, `low_level_init.c` and an alternative assembler source file, `low_level_init.s37`. The only limitation using the C source version is that static initialized variables cannot be used within the file, as variable initialization has not been performed at this point.

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

MODIFYING THE FILE `CSTARTUP.S37`

As noted earlier, you should not modify the file `cstartup.s37` if a customized version of `__low_level_init` is enough for your needs. However, if you do need to modify the file `cstartup.s37`, 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 59.

Situations that require a modified `cstartup.s37` file

One specific situation that requires you to modify the `cstartup.s37` file is when the memory where the stack is located is not available when the microcomputer is reset.

The `__low_level_init` function must set up the hardware to make the stack memory available. However, the call from `cstartup` to `__low_level_init` requires a small amount of stack, and that memory must be available before the call to `__low_level_init`.

In this case, you must modify `cstartup` and set up the stack pointer to some RAM area, which can temporarily be used for the call to `__low_level_init`. For example, you can use memory reserved for static variables, as this memory is not initialized until after the call to `__low_level_init`. For more details, see the comments available in the `cstartup.s37` file.

Note that parts of the code in `cstartup.s37` depend on the used data model. Normally, the assembler does not need to know what data model you use, and there are no assembler options to specify the data model. To assemble the file `cstartup.s37`, you have to define the preprocessor symbol `__DATA_MODEL__` to the value 1 (for the Small data model) or 3 (for the Huge data model).



To define the symbol from the command line, use:

```
-D__DATA_MODEL__=1 (or 3)
```



In the IAR Embedded Workbench IDE, the assembler will automatically get the correct value for the preprocessor symbol `__DATA_MODEL__`.

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 `h8\src\lib` directory.

If you intend to rebuild the library, the source files are available in the template library project, see *Building and using a customized library*, page 61. 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 75.

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 *Data and function placement in segments*, page 48.

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 57.

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) |

Table 17: Descriptions of printf configuration symbols

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

Table 17: Descriptions of `printf` configuration symbols (Continued)

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 (<code>ll</code> qualifier) |
| <code>_DLIB_SCANF_SPECIFIER_FLOAT</code> | Floating-point numbers |
| <code>_DLIB_SCANF_SPECIFIER_N</code> | Output count (<code>%n</code>) |
| <code>_DLIB_SCANF_QUALIFIERS</code> | Qualifiers <code>h</code> , <code>j</code> , <code>l</code> , <code>t</code> , <code>z</code> , and <code>L</code> |
| <code>_DLIB_SCANF_SCANSET</code> | Scanset (<code>[*]</code>) |
| <code>_DLIB_SCANF_WIDTH</code> | Width |
| <code>_DLIB_SCANF_ASSIGNMENT_SUPPRESSING</code> | Assignment suppressing (<code>[*]</code>) |

Table 18: Descriptions of `scanf` configuration symbols

CUSTOMIZING FORMATTING CAPABILITIES

To customize the formatting capabilities, you need to set up a library project, see *Building and using a customized library*, page 61. 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 53. 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 19: Low-level I/O files

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

Note: If you link your library with I/O debugging support, C-SPY variants of the low-level I/O functions will be linked for interaction with C-SPY. For more information, see *Debug support in the runtime library*, page 53.

Locale

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

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

The DLIB library can be used in two 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 support the C locale only
- All libraries with *full library configuration* have support for the locale interface. For prebuilt libraries with locale interface, it is by default only supported to switch multibyte character encoding during runtime.
- Libraries with *normal library configuration* do not have support for the locale interface.

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

CUSTOMIZING THE LOCALE SUPPORT

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

- The standard C locale
- The POSIX locale
- A wide range of 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 61.

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 `h8\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 59.

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 61.

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 53.

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 `h8\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 59.

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 61.

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 `h8\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 59.

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 61.

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 75.

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 61. 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 `h8\src\lib` directory. For further information, see *Building and using a customized library*, page 61. To turn off assertions, you must define the symbol `NDEBUG`.



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

Heaps

The runtime environment supports heaps in the following memory types:

| Memory type | Segment name | Extended keyword | Used by default in data model |
|-------------|--------------|------------------|-------------------------------|
| Data16 | DATA16_HEAP | __data16 | Small |
| Data32 | DATA32_HEAP | __data32 | Huge |

Table 20: Heaps and memory types

See *The heap*, page 41 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 data model. For information about how to use a specific heap in C++, see *New and Delete operators*, page 109.

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

| Function | Description |
|----------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>__close</code> | Closes the associated host file on the host computer |
| <code>__exit</code> | C-SPY notifies that the end of the application has been reached * |
| <code>__open</code> | Opens a file on the host computer |
| <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>__seek</code> | Seeks in the associated host file on the host computer |
| <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>_ReportAssert</code> | Handles failed asserts * |
| <code>abort</code> | C-SPY notifies that the application has called <code>abort</code> * |
| <code>clock</code> | Returns the clock 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>system</code> | Writes a message to the Debug Log window and returns -1 |

Table 21: Functions with special meanings when linked with debug info

| Function | Description |
|-------------------|---------------------------------------|
| <code>time</code> | Returns the time on the host computer |

Table 21: Functions with special meanings when linked with debug info (Continued)

* 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 53. 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 *H8 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 H8 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 22: 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 225.

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 *H8 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 H8 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>__bus_width¹⁾</code> | 20, 24, 28, 32, or * | Corresponds to the <code>--bus_width</code> option. The attribute is specified for files compiled in the Advanced operating mode, but not for files compiled in the Normal operating mode. |
| <code>__code_model</code> | small or large | Corresponds to the code model used in the project. |
| <code>__data_model</code> | small or huge | Corresponds to the data model used in the project. |
| <code>__double_size</code> | 32, or 64 | Size in bits of the <code>double</code> data type. |
| <code>__interrupt_mode¹⁾</code> | 0, 1, 2, 3, or * | Corresponds to the used interrupt mode. |
| <code>__int_size</code> | 16 | Size in bits of the <code>int</code> data type. Present for future compatibility. |

Table 23: Predefined runtime model attributes

| Runtime model attribute | Value | Description |
|--------------------------------------------|--------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>__operating_mode</code> | normal or advanced | Corresponds to the operating mode used in the project. |
| <code>__rt_version</code> | <i>n</i> | This runtime key is always present in all modules generated by the H8 IAR C/C++ Compiler. If a major change in the runtime characteristics occurs, the value of this key changes. |
| <code>__use_h8s_instr</code> ¹⁾ | yes, no, or * | H8S instructions may be used in the code generated for this module. |
| <code>__use_mac_instr</code> ¹⁾ | yes, no, or * | Corresponds to the <code>--enable_mac</code> option. |

Table 23: Predefined runtime model attributes (Continued)

¹⁾ If you use the `--weak_rtmodel_check` compiler option, this runtime model attribute may have the value `*` instead of any other possible value. To read more about this, see *Using a weak runtime model check*, page 80.

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 *H8 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 calling convention 3*, page 91.

The following assembler source code provides a function, `test_and_set`, that tests and sets the variable `semaphore`. Because the function uses an instruction that is only available for H8S, the function may only be included in a project which uses an H8S microcomputer. To ensure this, the runtime model attribute `__use_h8s_instr` has been defined with the value `yes`. This definition ensures that the module `test_and_set` can only be linked with either other modules that contain the same definition, or with modules that specify the special value `*` for the attribute. Note that the compiler sets this attribute to `no` when you build an H8/300H project unless the command line option `--weak_rtmodel_check` is used.

```
CASEON
RTMODEL "__use_h8s_instr", "yes"
MODULE test_and_set
PUBLIC test_and_set
EXTERN semaphore
RSEG CODE16:CODE:NOROOT(1)
```

```

test_and_set:
    ; ER5 and ER6 are scratch registers
    ; Return semaphore value in R6L
    MOV.B  #0, R6L
    MOV.L  #semaphore, ER5
    TAS   @ER5
    BPL   was_0
    MOV.B  #1, R6L
was_0:
    RTS
    ENDMOD
    END

```

If this module is used in an application built for the H8/300H core, the following error is issued by the linker:

```

Error[e117]: Incompatible runtime models. Module myMain specifies
that '__use_h8s_instr' must be 'no', but module test_and_set has
the value 'yes'

```

USING A WEAK RUNTIME MODEL CHECK

If you use the `--weak_rtmodel_check` compiler option, some of the runtime model attributes may have the value `*` instead of any other possible value. This means that the compiled module can be linked with other modules which have a specific value other than `*`. This allows for modules that are compatible to be linked together although they have not been compiled with the same compiler options.

If you use the `--weak_rtmodel_check` compiler option, the following runtime model attributes will have the `*` value in the following situations:

| | |
|------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>__bus_width</code> | If no code is generated that depends on the specified bus width, this symbol will have the value <code>*</code> , which means that bus width-independent code can be used in any project. |
| <code>__use_h8s_instr</code> | If compiling for H8/300H, this symbol will have the value <code>*</code> , which means that the generated code can be used in projects built for an H8S microcomputer. |
| <code>__use_mac_instr</code> | If no code is generated that depends on the presence of <code>MAC</code> registers or <code>MAC</code> instructions, this symbol will have the value <code>*</code> , which means that the generated code can be used in any project independently of the presence of <code>MAC</code> hardware. |

`__interrupt_mode` If no code is generated that depends on the specified interrupt mode, this symbol will have the value `*`, which means that the generated code can be used in any project independently of which interrupt mode that is being used.

If you use the IAR Embedded Workbench IDE, libraries will automatically be built with the option `--weak_rtmodel_check`, making it possible to use the libraries in more situations than would otherwise have been possible.

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 `#pragma rtmodel` directive or the `RTMODEL` assembler directive. For each property, select a key and a set of values that describe the states of the property that are incompatible. Note that key names that start with two underscores are reserved by the compiler.

Examples

If you write code for a specific device, your code will not work correctly if used in a project for another device. In this case it can be useful to specify a runtime model attribute that describes the device for which you have written your code. Using this attribute in all modules specific to this device, and corresponding attributes for code related to any other device provides you with an automatic control that code will not be incorrectly linked together. Most parts of your code is probably device-neutral and do not need to include a device attribute.

In a file `ad_converter_handler.c`:

```
#pragma rtmodel="device", "H8S_2148_Series"
.../* Code that handles the A/D converters
    for this specific device */
```

In the main file of your application, which is intended to be used for an application for a different device, for example the H8S/2655 Series:

```
#pragma rtmodel="device", "H8S_2655_Series"
```

If you try to build these two files together, the linker will notify you about the files being incompatible.

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

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


Assembler language interface

When you develop an application for an embedded system, there may be situations where you will find it necessary to write parts of the code in assembler, for example, when using mechanisms in the H8/300H and H8S microcomputer families that require precise timing and special instruction sequences.

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

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

Mixing C and assembler

The H8 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 89.

The section on memory access methods, page 98, covers how data in memory is accessed.

The answer to the final question is that the call stack can be displayed when you run assembler code in the debugger. However, the debugger requires information about the *call frame*, which must be supplied as annotations in the assembler source file. For more information, see *Call frame information*, page 101.

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. On the other hand, you will have a well-defined interface between what the compiler performs and what you write in assembler. When using inline assembler, you will not have any guarantees that your inline assembler lines do not interfere with the compiler generated code.

The recommended method for mixing C or C++ and assembler modules is described in *Calling assembler routines from C*, page 86, and *Calling assembler routines from C++*, page 88, 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 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 __data8 flag;

void foo()
{
    while (!flag)
    {
        asm("BSET #0,@flag:8");
    }
}
```

In this example, the change 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

- Alignment cannot be controlled
- 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 information about the `bool` data type, see *Integer types*, page 176.

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.

The compiler supports several different calling conventions, see *Calling convention*, page 89. Although any calling convention can be used, it is strongly recommended to use calling convention 1 or 2 when you interface between C and assembler.

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;

__cc_version1 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 the IAR Embedded Workbench IDE, specify list options on file level. Select the file in the workspace window. Then choose **Project>Options**. In the **C/C++ Compiler** category, select **Override inherited settings**. On the **List** page, deselect **Output list file**, and instead select the **Output assembler file** option and its suboption **Include source**. Also, be sure to specify a low level of optimization.



Use the following options to compile the skeleton code:

```
icch8 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 `s37`. Also remember to specify the code model, and data model you are using as well as a low level of optimization and `-e` for enabling language extensions.

The result is the assembler source output file `skeleton.s37`.



Note: The `-lA` option creates a list file containing call frame information (CFI) directives, which can be useful if you intend to study these directives and how they are used. If you only want to study the calling convention, you can exclude the CFI directives from the list file. In the IAR Embedded Workbench IDE, select **Project>Options>C/C++ Compiler>List** and deselect the suboption **Include compiler runtime information**. On the command line, use the option `-lB` instead of `-lA`. Note that CFI information must be included in the source code to make the C-SPY Call Stack window work.

The output file

The output file contains the following important information:

- The calling convention
- The return values
- The global variables
- The function parameters
- 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 H8 IAR C/C++ Compiler provides three calling conventions. This section describes the calling conventions used by the H8 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 three calling conventions to choose between:

- Calling convention 3 (`__cc_version3`)

A `__cc_version3` declared function, or a function without an explicitly specified calling convention, uses the calling convention of the H8 IAR C/C++ Compiler, version 2.x. Basically, the compiler selects appropriate parameters with a size of up to 64 bits and passes them in the registers `ER6`, `ER5`, and `ER4`. Remaining parameters are passed on the stack. The first stack parameter has the lowest address on the stack, and so on. This calling convention is used by the compiler by default. However, it is not recommended to use this convention when interfacing between C and assembler.

- Calling convention 2 (`__cc_version2`)

A `__cc_version2` declared function uses the calling convention of ICCH8, compiler versions 1.20A and up. Basically, the first two scalar parameters with a size of up to 32 bits are sent in the registers `ER6` and `ER5`. This calling convention is provided to support assembler functions written for older compilers being called by the new compiler. Old C functions using this calling convention can normally not be used, as they require an environment which is not provided by the new compiler. You can use this calling convention when interfacing between C and assembler, allowing you to pass up to two parameters in registers.

- Calling convention 1 (`__cc_version1`)

A `__cc_version1` declared function uses the calling convention of ICCH8, compiler versions up to 1.20A. Basically, the first scalar parameter with a size of up to 32 bits is sent in the `ER6` register. This calling convention is provided to support assembler functions written for older compilers being called by the new compiler. Old C functions using this calling convention can normally not be used, as they require an environment which is not provided by the new compiler. You can use this calling convention when interfacing between C and assembler, allowing you to pass one parameter in registers.

You can declare individual functions to use the calling convention 1 or 2 by using the `__cc_version1` or `__cc_version2` function attribute, for example:

```
extern __cc_version1 void doit(int arg);
```



Hints for using calling convention 3

The default calling convention, calling convention 3, is very complex and 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 86.

If you intend to use calling convention 3, you should also specify a value to the runtime model attribute `__rt_version` using the `RTMODEL` assembler directive:

```
RTMODEL "__rt_version"="value"
```

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

```
__cc_version1 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"
{
    __cc_version1 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++:

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

    __cc_version1 int f(int);

#ifdef __cplusplus
}
#endif
```

PRESERVED VERSUS SCRATCH REGISTERS

The general H8/300H and H8S 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 `__cc_version3`, the registers ER4, ER5, and ER6 are scratch registers
- For `__cc_version1` and `__cc_version2`, the registers ER5 and ER6 are scratch registers.

Preserved registers

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

- For `__cc_version3`, the registers ER0–ER3 must be preserved
- For `__cc_version1` and `__cc_version2`, the registers ER0–ER4 must be preserved.

Special registers

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

- The stack pointer register—`R7` or `ER7`—must at all times point to or below the last element on the stack. In the eventuality of an interrupt, everything below the point the stack pointer points to, will be destroyed.
- The status registers—`CCR` and `EXR`, as required—are saved in interrupt, monitor, and trap functions.
- When running in the Normal operating mode, the stack pointer only occupies the register `R7` of `ER7`. In this version of the compiler, the register `E7` of `ER7` is not used by the compiler, and is free for use as an extra scratch register or similar. However, note that future versions of the compiler may use register `E7`.

FUNCTION CALL

During a function call, the calling function passes the parameters, either in registers or on the stack. Control is then passed to the called function with the return address being automatically pushed on the stack.

The called function:

- stores any local registers required by the function on the 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 are passed on the stack.

Register parameters

The registers available for passing parameters are:

- For `__cc_version3`, parameters with a size up to 64 bits are sent in the registers `ER4`, `ER5`, and `ER6`
- For `__cc_version2`, the first two scalar parameters with a size up to 32 bits are sent in the registers `ER6` and `ER5`
- For `__cc_version1`, the first scalar parameter with a size up to 32 bits is sent in the registers `ER6`.

In addition to the parameters visible in a function declaration and definition, there can be hidden parameters, which are also passed via registers. If the function returns a `struct`, or `union`, and for calling convention 1 and 2 also `double`, the memory location where the structure is to be stored is passed using a pointer with the stack pointer attribute. This pointer is passed as a hidden parameter in a register.

Register assignment using calling convention 1 and 2

The assignment of registers to parameters is a straightforward process. Traversing the parameters in strict order from left to right, using a first-fit algorithm, the first, or first two, parameters are passed in registers if they are scalar and up to 32 bits in size.

Parameters are allocated to registers according to the following table:

| Parameters | Passed in registers (<code>__cc_version1</code>) | Passed in registers (<code>__cc_version2</code>) |
|---------------|-------------------------------------------------------|-------------------------------------------------------|
| 8-bit values | R6L | Parameter 1: R6L Parameter 2: R5L |
| 16-bit values | R6 | Parameter 1: R6 Parameter 2: R5 |
| 32-bit values | ER6 | Parameter 1: ER6 Parameter 2: ER5 |

Table 24: Registers used for passing parameters

Register assignment using calling convention 3

In this 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 to achieve the most efficient register usage.

The algorithm for assigning parameters to registers is quite complex in this 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 86.

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
- Unnamed parameters to variable length functions; in other words, functions declared as `foo(param1, ...)`, for example `printf`.

Stack layout

Stack parameters are stored in the main memory, starting at the location pointed to by the stack pointer. Below the stack pointer (towards low memory) there is free space that the called function can use. The parameters are passed in reverse order, which means the last parameter is transferred first and it is stored at the location pointed to by the stack pointer. The next one is stored at the next location on the stack that is divisible by two, etc.

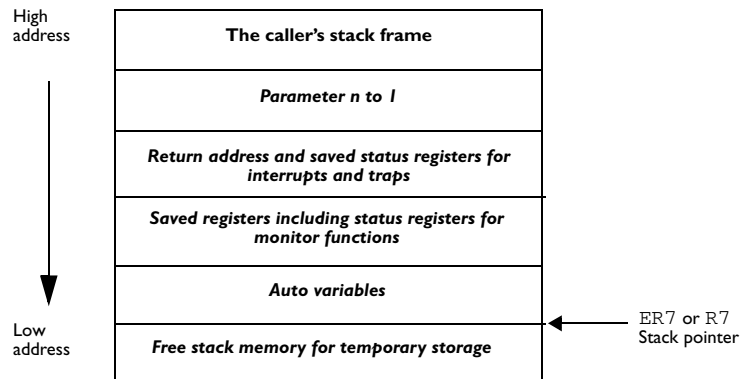


Figure 2: Storing stack parameters in memory

FUNCTION EXIT

The called function exits by deallocating auto variables, restoring registers, and finally popping the return address from the stack.

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 all three calling conventions, the following details are valid:

- Scalar return values are passed in registers
- For calling convention 1 and 2, return values larger than 32 bits are returned as `struct/union` return values
- `struct` and `union` return values are passed using a pointer with the stack data pointer attribute. The memory location where this type of return values are stored is pointed to by an implicit first parameter passed to the function.

Registers used for returning values

Scalar return values are passed in registers; the following registers are available for returning values R6L, R6, ER6, or ER5 : ER4 depending on the size of the return value.

| Return values | Passed in registers |
|---------------|---------------------------------------|
| 8-bit values | R6L |
| 16-bit values | R6 |
| 32-bit values | ER6 |
| 64-bit values | ER5 : ER4 (calling convention 3 only) |

Table 25: Registers used for returning values

Stack cleaning at function exit

Normally, it is the responsibility of the caller to clean the stack after the called function has returned.

Return address handling

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

The return address is restored directly from the stack with the RTS instruction; RTE for interrupt functions.

RESTRICTIONS FOR SPECIAL FUNCTION TYPES

The following deviations from the specified calling conventions exist:

- An `__interrupt` declared function saves all used registers (there are no scratch registers)
- A `__task` declared function does not save the caller's registers
- A `__raw` declared interrupt function does not save the caller's register.

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:

```
__cc_version1 short add1(short);
```

This function takes one parameter in the register R6, and the return value is passed back to its caller in the register R6.

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:

```
INC.W    #1,R6
RTS
```

Example 2

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

```
struct a_struct { long a; };
__cc_version1 short a_function(struct a_struct x, short y);
```

The calling function must reserve four 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 R6. The return value is passed back to its caller in the register R6.

Example 3

The function below will return a `struct`.

```
struct a_struct { long a; };
__cc_version1 struct a_struct a_function(short 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 R6 in the Small data model, and in ER6 in the other models. The parameter `x` is passed on the stack, as calling convention 1 only passes one parameter in registers.

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

```
__cc_version1 struct a_struct * a_function(short x);
```

In this case, the return value is a scalar, so there is no hidden parameter. The parameter `x` is passed in R6, and the return value is returned in R6 or ER6, depending on the data model.

FUNCTION DIRECTIVES

Note: This type of directives is primarily intended to support static overlay, a feature which is useful in some smaller microcontrollers. The H8 IAR C/C++ Compiler does not use static overlay, because it has no use for it.

The function directives `FUNCTION`, `ARGFRAME`, `LOCFRAME`, and `FUNCALL` are generated by the H8 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** (`-1A`) to create an assembler list file.

For reference information about the function directives, see the *H8 IAR Assembler Reference Guide*.

Memory access methods

This section describes the different memory types presented in the chapter *Data storage*. In addition to just presenting the assembler code used for accessing data, it will be used for explaining the reason behind the different memory types.

You should be familiar with the H8/300H and H8S instruction set, in particular the different addressing modes used by the instructions that can access memory.

The H8 IAR Assembler supports all of the many addressing modes of the H8/300H and H8S architectures. For simplicity, the following examples assume an H8S device. The following addressing modes are used in the examples:

| Addressing mode | Symbol | Instruction example | Instruction size |
|-------------------------------------------------------|--------------|--------------------------------|------------------|
| Register indirect | @ERn | MOV.B @ER2, R6L | 2 bytes |
| Register indirect with displacement and 16-bit offset | @(d:16, ERn) | MOV.B @(4:16, ER2), R6L | 4 bytes |
| Register indirect with displacement and 32-bit offset | @(d:32, ERn) | MOV.B @ (0x40000:32, ER2), R6L | 8 bytes |
| Absolute 8-bit address | @aa:8 | MOV.B @x:8, R6L | 2 bytes |
| Absolute 16-bit address | @aa:16 | MOV.B @x:16, R6L | 4 bytes |
| Absolute 32-bit address | @aa:32 | MOV.B @x:32, R6L | 6 bytes |

Table 26: Specifying the size of an assembler memory instruction

Note that the instruction size differs significantly between the different addressing modes, especially for the absolute addressing modes. The generated code will become significantly smaller if the most suitable addressing mode can be used for accessing your data. This is the reason for introducing data models and data memory attributes, which are important tools for you to control the size of the generated application code.

EXAMPLE CODE FOR SHOWING DIFFERENCES IN MEMORY TYPES

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

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

These three examples can be illustrated by the following C program:

```
char x;
char y[10];

struct s
{
    long a;
    char b;
};

char test(short i, struct s * p)
{
    return x + y[i] + p->b;
}
```

THE DATA16 MEMORY ACCESS METHOD

When it is known to the compiler that a data object is located in data16 memory, it can use the @aa:16 addressing mode for absolute addressing. The compiler may also be able to use the @(d:16, ERn) addressing mode for indirect addressing with displacement, instead of using the more expensive @(d:32, ERn) addressing mode.

For more details about data16 memory, see *Data16 memory*, page 17.

Examples

The following list shows how data16 memory is accessed for the code examples listed in *Example code for showing differences in memory types*, page 99; in this case the Normal operating mode is assumed:

| | |
|---------------------------|------------------------------------------------------|
| MOV.B @x:16, R6L | Access the global variable x |
| MOV.B @(y:16, ER5), R6L | Access an entry in the global array y, index i in R5 |
| MOV.B @(0x4:16, ER5), R6L | Access through a pointer, pointer p in R5 |

THE DATA32 MEMORY ACCESS METHOD

When it is known to the compiler that a data object is located in data32 memory, it must use the @aa:32 addressing mode for absolute addressing.

For more details about data32 memory, see *Data32 memory*, page 17.

Examples

The following list shows how data32 memory is accessed for the code examples listed in *Example code for showing differences in memory types*, page 99:

| | |
|---------------------------|---------------------------------------------------|
| MOV.B @x:32, R6L | Access the global variable <i>x</i> |
| MOV.B @(y:32, ER5), R6L | Access an array, index <i>i</i> in ER5 |
| MOV.B @(0x4:16, ER5), R6L | Access through a pointer, pointer <i>p</i> in ER5 |

THE DATA8 MEMORY ACCESS METHOD

When it is known to the compiler that a data object is located in data8 memory, it can use the @aa:8 addressing mode for absolute addressing.

For more details about data8 memory, see *Data8 memory*, page 16.

Examples

The following list shows how data8 memory is accessed for the examples listed in *Example code for showing differences in memory types*, page 99; in this case the Normal operating mode is assumed. The list also includes the code required to extend a data8 pointer to be used for accessing the variable.

| | |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| MOV.B @x:8, R6L | Access the global variable <i>x</i> |
| ADD.B #LOW(y), R5L | Access an entry in the global array <i>y</i> . R5 contains the index <i>i</i> when starting. Note that only the low part of <i>i</i> is used. The array index in data8 memory is 8 bits in size. |
| MOV.B #0xFF, R5H | |
| MOV.B @ER5, R6L | |
| ADD.B #0x4, R5L | Access through a pointer. R5L contains the pointer <i>p</i> when starting. |
| MOV.B #0xFF, R5H | |
| MOV.B @ER5, R6L | |

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 H8 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.

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

| Resource | Description |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------|
| R0L:8, R0H:8, R1L:8, R1H:8, R2L:8, R2H:8, R3L:8, R3H:8, R4L:8, R4H:8, R5L:8, R5H:8, R6L:8, R6H:8, E0:16, E1:16, E2:16, E3:16, E4:16, E5:16, E6:16, E7:16, R0:16, R1:16, R2:16, R3:16, R4:16, R5:16, R6:16, ER0:32, ER1:32, ER2:32, ER3:32, ER4:32, ER5:32, ER6:32 | Normal registers |
| R7:16 | The stack pointer in Normal operating mode |
| ER7:32 | The stack pointer in Advanced operating mode |

Table 27: Call frame information resources defined in a names block

| Resource | Description |
|----------------------------------------------|---------------------------------------------|
| CCR: 8, EXR: 8 | Special purpose registers* |
| ?RET32 | The return address |
| ?RETWORD0: 16, ?RETBYTE2: 8, ?RETBYTE3: 8 | These resources describe the return address |

Table 27: Call frame information resources defined in a names block (Continued)

* **Note that for compatibility, EXR is a defined resource also for H8/300H, although H8/300H does not have an EXR register.**

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 the 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
- The `mutable` attribute.

The exclusion of these language features makes the runtime library significantly more efficient. The Embedded C++ library furthermore differs from the full C++ library in that:

- The standard template library (STL) is excluded
- Streams, strings, and complex numbers are supported without the use of templates
- Library features which relate to exception handling and runtime type information (the headers `except`, `stdexcept`, and `typeinfo`) are excluded.

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

EXTENDED EMBEDDED C++

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

- Full template support
- Namespace support
- The `mutable` attribute
- The cast operators `static_cast`, `const_cast`, and `reinterpret_cast`.

All these added features conform to the C++ standard.

To support Extended EC++, this product includes a version of the standard template library (STL), in other words, the C++ standard chapters utilities, containers, iterators, algorithms, and some numerics. This STL has been tailored for use with the Extended EC++ language, which means that there are no exceptions, no multiple inheritance, and no support for runtime type information (`rtti`). Moreover, the library is not in the `std` namespace.

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

ENABLING C++ SUPPORT



In the H8 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 156. 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 157.



To set the equivalent option in the IAR Embedded Workbench IDE, 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 can be implicitly converted 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 qualifiers*, page 183.

Example

```
class A {
public:
    //Located in data16 at address 60
    static __data16 __no_init int i @ 60;
    static __vector_call void f(); //Call via vector area
    __vector_call void g();      //Call via vector area
};
```

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 converted to a default data pointer.

Example

```
class B {
public:
    void f();
    int i;
};
B __data8 b;
```

If you use the Small data model, the following declaration is illegal. A data32 pointer cannot implicitly be converted to a default (data16) pointer. The address of `b` cannot be converted to a `this` pointer of an object of class `B`:

```
B __data32 b;
```

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

In the following examples, the Advanced operating mode and the Huge data model are assumed:

```
class __data16 C {
public:
    void f();           // Has a this pointer of type C __data16 *
    void f() const;    // Has a this pointer of type
                        // C __data16 const *
    C();               // Has a this pointer pointing into data16
                        // memory
    C(C const &);      // Takes a parameter of type C __data16
                        // const & (also true of generated copy
                        // constructor)

    int i;
};
C Ca;                 // Resides in data16 memory instead of the
                        // default memory
C __data16 Cb;        // Resides in data16 memory, the 'this'
                        // pointer still points into data16 memory
```

```

C __data32 Cc;    // Not allowed, the __data32 pointer
                 // can't be implicitly converted into a
                 // __data16 pointer

void h()
{
    C Cd;        // Not allowed, auto variables are located in
                 // data32 memory
}
C * Cp1;        // Creates a pointer to data16 memory
C __data8 * Cp2; // Creates a pointer to data8 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 __data16 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 `__data16`.

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 __data16 D : public C { // OK, same class memory
public:
    void g();
    int j;
};

class __data8 E : public C { // OK, data8 memory is inside
                           // data16
public:
    void g() // Has a this pointer pointing into data8 memory
    {
        f(); // Gets a this pointer into data16 memory
    }
    int j;
};

class __data32 F : public C { // Not OK, data32 memory isn't
                           // inside data16 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++.

In member functions, the `this` parameter is a pointer to the memory specified by the class. To call a member function of an object, the `this` parameter must be able to point to the object. For the same reason, if the class has anything but a trivial constructor or a destructor, objects of that type cannot be created in a memory to which the `this` parameter could not point.

In particular, temporary objects and auto objects are created in stack memory, and cannot be used in situations where this would violate the restrictions on the `this` pointer.

For more information about memory types, see *Memory types*, page 15.

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, `data16` and `data32` memory.

These examples assume that there is a heap in both `data16` and `data32` memory.

```
void __data16 * operator new __data16(__data16_size_t);
void __data32 * operator new __data32(__data32_size_t);
void operator delete(void __data16 *);
void operator delete(void __data32 *);
```

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

```
void __data16 * operator new[] __data16(__data16_size_t);
void __data32 * operator new[] __data32(__data32_size_t);
void operator delete[](void __data16 *);
void operator delete[](void __data32 *);
```

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 __data16(__data16_size_t)
int __data16 *p = new __data16 int;

//Calls operator new __data16(__data16_size_t)
int __data16 *q = new int __data16;

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

//Calls operator new __data32(__data32_size_t)
class __data32 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 __data16 *)
delete s; //Calls operator delete(void __data32 *)
```

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 __data16 * t = new __data32 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

```
// Data16 is assumed to be the memory type of the default
// pointer.
template<typename> class Z;
template<typename T> class Z<T *>;

Z<int __data8 *> zn; // T = int __data8
Z<int __data16 *> zf; // T = int
Z<int *> zd; // T = int
Z<int __data32 *> zh; // T = int __data32
```

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 __data8 *) 0); // T = int. The result is different
                        // than the analogous situation with
                        // class template specializations.
fun((int *) 0); // T = int
fun((int __data16 *) 0); // T = int
fun((int __data32 *) 0); // T = int __data32
```

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 __data8 *) 0); // T = int __data8
```

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

```
typedef __bitvar struct {unsigned char n:1;} aBitVar;

template <aBitVar &y>
void foo()
{
    y.n = 1;
}

extern aBitVar x;
```

```
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 *Extended Embedded C++*, page 104.

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 __data16> __data16 x; // x placed in data16 memory,
                                 // heap allocation from data16,
                                 // uses pointers to data16
                                 // memory
vector<int __data32> __data16 y; // y placed in data16 memory,
                                 // heap allocation from data32,
                                 // uses pointers to data32
                                 // memory
vector<int __data16> __data32 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> mem` 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 __data16> x;
vector<int __data32> 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.

For the current version of the H8 IAR C/C++ Compiler, this is not very relevant as only one function memory attribute can be used at a time.

Example

```
class X{
public:
    __code24 void f();
};
void (__code24 X::*pmf)(void) = &X::f;
```

USING INTERRUPTS AND C++ 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`.

C++ language extensions

When you use the compiler in C++ mode and have enabled IAR language extensions, the following C++ language extensions are available in the compiler:

- In a friend declaration of a class, the `class` keyword may be omitted, for example:

```
class B;
class A
{
    friend B;           //Possible when using IAR language
                       //extensions
    friend class B;    //According to standard
};
```

- Constants of a scalar type may be defined within classes, for example:

```
class A {
    const int size = 10; //Possible when using IAR language
                        //extensions
    int a[size];
};
```

According to the standard, initialized static data members should be used instead.

- In the declaration of a class member, a qualified name may be used, for example:

```
struct A {
    int A::f(); //Possible when using IAR language extensions
    int f();   //According to standard
};
```

- It is permitted to use an implicit type conversion between a pointer to a function with C linkage (`extern "C"`) and a pointer to a function with C++ linkage (`extern "C++"`), for example:

```
extern "C" void f(); //Function with C linkage
void (*pf) ()       //pf points to a function with C++ linkage
                   = &f; //Implicit conversion of pointer.
```

According to the standard, the pointer must be explicitly converted.

- If the second or third operands in a construction that contains the `?` operator are string literals or wide string literals (which in C++ are constants), the operands may be implicitly converted to `char *` or `wchar_t *`, for example:

```
char *P = x ? "abc" : "def"; //Possible when using IAR
                             //language extensions
char const *P = x ? "abc" : "def"; //According to standard
```

- Default arguments may be specified for function parameters not only in the top-level function declaration, which is according to the standard, but also in typedef declarations, in pointer-to-function function declarations, and in pointer-to-member function declarations.
- In a function that contains a non-static local variable and a class that contains a non-evaluated expression (for example a `sizeof` expression), the expression may reference the non-static local variable. However, a warning is issued.

If you use any of these constructions without first enabling language extensions, errors are issued.

Efficient coding for embedded applications

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

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 H8 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 | Code hoisting Common subexpression elimination |
| High (Maximum optimization) | Cross jumping Cross call (when optimizing for size) Loop unrolling Function inlining Code motion Type-based alias analysis |

Table 28: 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 *H8 IAR Embedded Workbench® IDE User Guide* for information about the compiler options available in the IAR Embedded Workbench IDE. 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
- Loop unrolling
- Function inlining
- Code motion
- Type-based alias analysis.

Common subexpression elimination

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

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

To read more about the command line option, see `--no_cse`, page 163.

Loop unrolling

It is possible to duplicate the loop body of a small loop, whose number of iterations can be determined at compile time, to reduce the loop overhead.

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

The compiler heuristically decides which loops to unroll. Different heuristics are used when optimizing for speed 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_unroll`, page 165.

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 164.

Code motion

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

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

Type-based alias analysis

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

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

Type-based alias analysis is performed at optimization level **High**. For ISO/ANSI standard-conforming C 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 164.

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.

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 variables.

USING EFFICIENT DATA TYPES

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

- Use small and unsigned data types, (`unsigned char` and `unsigned short`) unless your application really requires signed values.
- 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 `data8` this is `signed char`, for `data16` this is `short`, and for `data32` it is `long`.
- 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 H8 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 H8 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 setting of the `--double` compiler option.

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 since 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;
}
```

DATA MODEL AND DATA MEMORY ATTRIBUTES

The H8 IAR C/C++ Compiler provides two different data models and a set of data memory attributes—`__bitvar`, `__data8`, `__data16`, and `__data32`—corresponding to the different memory types. For more details, see *Memory types*, page 15.

For many applications it is sufficient to use the data model feature to specify the default memory for the data objects. However, efficient usage of the data model in combination with the memory attributes can significantly reduce the application size.

Using Small or Huge data model in the Advanced operating mode

When running in the Advanced operating mode, it may feel natural to use the Huge data model. However, if most of your data, including the runtime stack, can fit in 64 Kbytes of memory—the low 32 Kbytes and the high 32 Kbytes—you can potentially gain much by using the Small data model; primarily when most data accesses are direct.

You can still place individual variables in other parts of the memory by using the data memory attribute `__data32`. Note two things:

- Pointers to variables declared `__data32` cannot be handled by the prebuilt runtime library for the Small data model because this model assumes that all pointers are pointers to data16 variables and have the size 16 bits. However, the compiler will notify you about any attempts of sending a pointer to a data32 variable to a library function.
- Even though pointers to data16 variables are 16 bits in size, they have to be extended to 32 bits before they can be used in the Advanced operating mode. For code that makes most data accesses through pointers, some of the advantages with using the Small data model are lost. In some cases, the code might even become larger and slower than when using the Huge data model.

Placing frequently accessed data in data8 memory

There is no data model where data8 is the default memory. It is not likely that all of your default data, including the runtime stack, fits into the topmost 256 bytes of memory, especially considering that the special function registers are normally also placed in this area. However, if you place frequently accessed variables here by using the `__data8` memory attribute, the code to access these variables will be both smaller and faster.

Note that this is mainly true for character-sized variables. Among the `MOV` instructions, it is only `MOV.B` that can take advantage of placement in data8 memory. The `MOV.W` and `MOV.L` instructions must access data8 variables using the `@aa:16` addressing mode.

As for data16, placing a variable in data8 memory is primarily useful for direct data accesses. In general, accessing data8 variables through pointers is not a good idea. The code will probably be both larger and slower compared to placing the variables in default memory. In the same way, it is normally no advantage to store arrays in data8 memory.

REARRANGING ELEMENTS IN A STRUCTURE

The H8/300H and H8S microcomputer families require that data in memory must be aligned. Each element in a structure needs to be aligned according to its specified type requirements. This means that the compiler must insert *pad bytes* if the alignment is not correct.

There are two reasons why this can be considered a problem:

- Network communication protocols are usually specified in terms of data types with no padding in between
- There is a need to save data memory.

For information about alignment requirements, see *Alignment*, page 175.

There are two ways to solve the problem:

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

For further details about the `#pragma pack` directive, see *#pragma pack*, page 222.

ANONYMOUS STRUCTS AND UNIONS

When a structure or union is declared without a name, it becomes anonymous. The effect is that its members will only be seen in the surrounding scope.

Anonymous structures are part of the C++ language; however, they are not part of the C standard. In the H8 IAR C/C++ Compiler they can be used in C if language extensions are enabled.



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



Use the `-e` compiler option to enable language extensions. See *-e*, page 156, 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 164.
- 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 83.

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.

STACK POINTER ARITHMETICS

The stack pointer has a size of 32 bits (`ER7`) in the Advanced operating mode and 16 bits (`R7`) in the Normal operating mode. This means that stack arithmetics in the Normal operating mode can be less expensive than in the Advanced operating mode.

For example, when the compiler reserves an area on the stack for local (auto) variables it subtracts a constant value from the stack pointer. In the Advanced operating mode, this requires using the `SUB.L` instruction. In the Normal operating mode, a `SUB.W` instruction can be used instead, which means faster and a more compact code can be generated.

In addition to the difference in hardware behavior when using different operating modes, a corresponding technique can be used if the maximum size of the runtime stack is known by the compiler.

For example, in the Advanced operating mode, only the lower 16 bits of the stack pointer need to be updated if the size of the stack is smaller than 64 Kbytes, and the stack is located entirely within one 64 Kbyte page in memory (`...0000-...FFFF`). Another example, but less common, is if the stack size is smaller than 256 bytes, and the stack is located entirely within one 256-byte page in memory (`...00-...FF`).

By using the compiler option `--stack_pointer_size`, you can make it possible for the compiler to generate more efficient code in these cases.

To read more about this option, see `--stack_pointer_size`, page 171.

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;
{
    .....
}
```

FUNCTION CALLS

For the H8/300H and H8S architecture, there are only a few possibilities to influence the function call and make them more efficient:

- By declaring a function with the function attribute `__vector_call`, the address to the function will automatically be inserted at a free place in the vector area of the microcomputer. Any calls to this function will then use the `@@aa:8` addressing mode. This type of call is 2 bytes in size, compared to 4 bytes for a normal call. By declaring frequently used functions as `__vector_call`, you reduce the size of your application.
- You may declare a function with the `__trap` keyword. In addition to the primary effect that the trap function will execute with interrupts disabled, the call is also shorter than a normal function call. However, in all other respects, trap functions have all the overhead of normal function calls.
- In contrast to functions declared `__trap`, there is an intrinsic function named `__TRAPA` that *only* inserts a `TRAPA` instruction. This is not handled as a normal function call. The caller does not save any registers, or add any other overhead. This makes the function call efficient. On the other hand, the called function must not destroy any registers except the status register. All things considered, this type of function call is primarily suitable for tiny routines written in assembler where you want minimal overhead. Note that you cannot send any parameters to the function when using the `__TRAPA` intrinsic function.

Note: The function memory attributes `__code16` and `__code24` do not affect the function call.

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, so it cannot be larger than 255. It also cannot be negative, which means that 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. To read more about the `volatile` type qualifier, see *Declaring objects volatile*, page 183.

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.

ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for a number of H8/300H and H8S derivatives are included in the H8 IAR C/C++ Compiler delivery. The header files are named `ioderivative.h` and define the processor-specific special function registers (SFRs).

Note: Each header file contains one section used by the compiler, and one section used by the assembler.

SFRs with bitfields are declared in the header file. The following example is from `io2130.h` when used in the Advanced operating mode:

```
union un_kbcomp {           /* union KBCOMP */
    unsigned char BYTE;     /* Byte access */
    struct {               /* Bit access */
        unsigned char KBCH :3; /* KBCH */
        unsigned char KBADE :1; /* KBADE */
        unsigned char IrCKS :3; /* IrCKS */
        unsigned char IrE :1; /* IrE */
    } BIT;
};
#define KBCOMP (*(volatile union un_kbcomp __data32 *)0xFFFFE4)
```

By including the appropriate include file in your code, it is possible to access either the whole register or any individual bit (or bitfields) from C code as follows:

```
/* Access to the whole register */
KBCOMP.BYTE = 0;

/* Bitfield accesses */
KBCOMP.BIT.KBADE = 1;
KBCOMP.BIT.KBCH = 0;
```

You can also use the header files as templates when you create new header files for other H8/300H and H8S devices.

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.

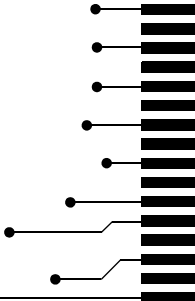
Variables declared using the `__no_init` keyword could, for example, be large input buffers or mapped to special RAM that keeps its content even when the application is turned off.

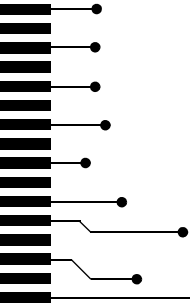
For information about the `__no_init` keyword, see page 211. Note that to use this keyword, language extensions must be enabled; see `-e`, page 156. For information about the `#pragma object_attribute`, see page 220.

Part 2. Compiler reference

This part of the H8 IAR C/C++ Compiler Reference Guide contains the following chapters:

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





Compiler usage

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

Compiler invocation

You can use the compiler either from the IAR Embedded Workbench IDE or from the command line. Refer to the *H8 IAR Embedded Workbench® IDE User Guide* for information about using the compiler from the IAR Embedded Workbench IDE.

INVOCATION SYNTAX

The invocation syntax for the compiler is:

```
icch8 [options][sourcefile][options]
```

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

```
icch8 prog --debug
```

The source file can either be a C or C++ file, typically with the filename extension `c` or `cpp`, respectively. If no filename extension is specified, the default extension `c` is assumed.

Generally, the order of options on the command line, both relative to each other and to the source filename, is *not* significant. There is, however, one exception: when you use the `-I` option, the directories are searched in the same order that they are specified on the command line.

If you run the compiler from the command prompt without any arguments, the compiler version number and all available options including brief descriptions are directed to `stdout` and displayed on the screen.

PASSING OPTIONS TO THE COMPILER

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

- Directly from the command line
Specify the options on the command line after the `icc8` command, either before or after the source filename; see *Invocation syntax*, page 135.
- Via environment variables
The compiler automatically appends the value of the environment variables to every command line; see *Environment variables*, page 136.
- Via a text file by using the `-f` option; see *-f*, page 158.

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

ENVIRONMENT VARIABLES

The following environment variables can be used with the H8 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\h8\inc;c:\headers</code> |
| <code>QCCH8</code> | Specifies command line options; for example: <code>QCCH8=-lA asm.lst -z9</code> |

Table 29: Environment variables

Include file search procedure

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

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

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

it searches the following directories for the file to include:

- 1 The directories specified with the `-I` option, in the order that they were specified, see *-I*, page 159.
- 2 The directories specified using the `C_INCLUDE` environment variable, if any, see *Environment variables*, page 136.

- 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:

```
icch8 ..\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.

Compiler output

The compiler can produce the following output:

- A linkable object file

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

- Optional list files

Different types of list files can be specified using the compiler option `-l`, see `-l`, page 159. By default, these files will have the filename extension `.lst`.

- Optional preprocessor output files

A preprocessor output file is produced when you use the `--preprocess` option; by default, the file will have the filename extension `.i`.

- Diagnostic messages

Diagnostic messages are directed to `stderr` and displayed on the screen, as well as printed in an optional list file. To read more about diagnostic messages, see *Diagnostics*, page 139.

- Error return codes

These codes provide status information to the operating system which can be tested in a batch file, see *Error return codes*, page 139.

- For each memory, the generated amount of bytes for functions and data

Information about the generated amount of bytes for functions and data for each memory is directed to `stdout` and displayed on the screen. Some of the bytes might be reported as *shared*.

Shared objects are functions or data objects that are shared between modules. If any of these occur in more than one module, only one copy will be retained. For example, in some cases inline functions are not inlined, which means that they are marked as shared, because only one instance of each function will be included in the final application. This mechanism is sometimes also used for compiler-generated code or data not directly associated with a particular function or variable, and when only one instance is required in the final application.

Error return codes

The H8 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 errors. |
| 3 | There were fatal errors making the compiler abort. |

Table 30: Error return codes

Diagnostics

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

MESSAGE FORMAT

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

```
filename,linenumber level[tag]: message
```

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 169.

Warning

A diagnostic message that is produced when the compiler finds a programming error or omission which is of concern, but not so severe as to prevent the completion of compilation. Warnings can be disabled by use of the command-line option *--no_warnings*, see page 166.

Error

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

SETTING THE SEVERITY LEVEL

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

See *Options summary*, page 146, for a description of the compiler options that are available for setting severity levels.

See the chapter *Pragma directives*, for a description of the pragma directives that are available for setting severity levels.

INTERNAL ERROR

An internal error is a diagnostic message that signals that there has been a serious and unexpected failure due to a fault in the compiler. It is produced using the following form:

```
Internal error: message
```


where *message* is an explanatory message. If internal errors occur, they should be reported to your software distributor or IAR Systems Technical Support. Include 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 application that generated the internal error

A list of the options that were used when the internal error occurred.

Compiler options

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

Compiler options syntax

Compiler options are parameters you can specify to change the default behavior of the compiler. You can specify compiler options from the command line—which is described in more detail in this section—and from within the IAR Embedded Workbench IDE. Refer to the *H8 IAR Embedded Workbench® IDE User Guide* for information about the compiler options available in the IAR Embedded Workbench IDE and how to set them.

TYPES OF OPTIONS

There are two *types of names* for command line options, *short* names and *long* names. Some options have both.

- A short option name consists of one character, and it may have parameters. You specify it with a single dash, for example `-e`
- A long option name consists of one or several words joined by underscores, and it may have parameters. You specify it with double dashes, for example `--char_is_signed`.

For information about the different methods for passing options, see *Passing options to the compiler*, page 136.

RULES FOR SPECIFYING PARAMETERS

There are some general syntax rules for specifying option parameters. First, the rules depending on whether the parameter is *optional* or *mandatory*, and whether the option has a short or a long name, are described. Then, the remaining rules are listed.

Optional parameters

For options with a short name and an optional parameter, the parameter should be specified without a preceding space, for example:

```
-z or -z3
```

For options with a long name and an optional parameter, the parameter should be specified with a preceding equal sign (=).

However, there are no long options with only an optional parameter. Optional parameters are always specified together with a mandatory parameter, see *Options with both optional and mandatory parameters*, page 144.

Mandatory parameters

For options with a short name and a mandatory parameter, the parameter can be specified either with or without a preceding space, for example:

```
-I..\src or -I ..\src\
```

For options with a long name and a mandatory parameter, the parameter can be specified either with a preceding equal sign (=) or with a preceding space, for example:

```
--diagnostics_tables=filename
```

or

```
-diagnostics_tables filename
```

Options with both optional and mandatory parameters

For options taking both optional and mandatory parameters, the rules for specifying the parameters are:

- For short options, optional parameters are specified without a preceding space
- For long options, optional parameters are specified with a preceding equal sign (=)
- For short and long options, mandatory parameters are specified with a preceding space.

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

```
-lA filename
```

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

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

Rules for specifying a filename or directory as parameters

The following rules apply for options taking a filename or directory as parameters:

- Options that take a filename as a parameter can optionally also take a path. The path can be relative or absolute. For example, to generate a listing to the file `list.lst` in the directory `..\listings\`:

```
icch8 prog -l ..\listings\list.lst
```

- For options that take a filename as the destination for output, the parameter can be specified as a path without a specified filename. The compiler stores the output in that directory, in a file with an extension according to the option. The filename will be the same as the name of the compiled source file, unless a different name has been specified with the option `-o`, in which case that name will be used. For example:

```
icch8 prog -l ..\listings\
```

The produced list file will have the default name `..\listings\prog.lst`

- The *current directory* is specified with a period (`.`). For example:

```
icch8 prog -l .
```

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

```
icch8 prog -l -
```

Additional rules

In addition, the following rules apply:

- When an option takes a parameter, the parameter cannot start with a dash (`-`) followed by another character. Instead, you can prefix the parameter with two dashes; the following example will create a list file called `-r`:

```
icch8 prog -l ---r
```

- For options that accept multiple arguments may be repeated, the arguments can be provided as a comma-separated list (without a space), for example:

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

Options summary

The following table summarizes the compiler command line options:

| Command line option | Description |
|------------------------|-----------------------------------------------------------------|
| --bus_width | Specifies width of address bus |
| --char_is_signed | Treats <code>char</code> as signed |
| --code_model | Specifies the code model |
| --core | Specifies a processor core |
| -D | Defines preprocessor symbols |
| --data_model | Specifies the data model |
| --debug | Generates debug information |
| --dependencies | Lists file dependencies |
| --diag_error | Treats these as errors |
| --diag_remark | Treats these as remarks |
| --diag_suppress | Suppresses these diagnostics |
| --diag_warning | Treats these as warnings |
| --diagnostics_tables | Lists all diagnostic messages |
| --direct_library_calls | Library functions are not called via the vector area |
| --dlib_config | Determines the library configuration file |
| --double | Forces the compiler to use 32-bit or 64-bit doubles |
| -e | Enables language extensions |
| --ec++ | Enables Embedded C++ syntax |
| --eec++ | Enables Extended Embedded C++ syntax |
| --enable_mac | Enables use of MAC register |
| --enable_multibytes | Enables support for multibyte characters |
| --error_limit | Specifies the allowed number of errors before compilation stops |
| -f | Extends the command line |
| --header_context | Lists all referred source files |
| -I | Specifies include file path |
| --interrupt_mode | Specifies interrupt mode |
| -l | Creates a list file |
| --library_module | Creates a library module |

Table 31: Compiler options summary

| Command line option | Description |
|--------------------------------------------------|------------------------------------------------------------------------------------------------------------|
| <code>--max_cycles_no_interrupt</code> | Specifies maximum number of cycles during which interrupts may be disabled. |
| <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</code> | Sets object module name |
| <code>--no_code_motion</code> | Disables code motion optimization |
| <code>--no_cse</code> | Disables common subexpression elimination |
| <code>--no_inline</code> | Disables function inlining |
| <code>--no_path_in_file_macros</code> | Removes the path from the return value of the symbols <code>__FILE__</code> and <code>__BASE_FILE__</code> |
| <code>--no_tbaa</code> | Disables type-based alias analysis |
| <code>--no_typedefs_in_diagnostics</code> | Disables the use of typedef names in diagnostics |
| <code>--no_unroll</code> | Disables loop unrolling |
| <code>--no_warnings</code> | Disables all warnings |
| <code>--no_wrap_diagnostics</code> | Disables wrapping of diagnostic messages |
| <code>-o</code> | Sets object filename |
| <code>--omit_types</code> | Excludes type information |
| <code>--only_stdout</code> | Uses standard output only |
| <code>--operating_mode</code> | Specifies the operating mode |
| <code>--preinclude</code> | Includes an include file before reading the source file |
| <code>--preprocess</code> | Generates preprocessor output |
| <code>--public_equ</code> | Defines a global named assembler label |
| <code>-r</code> | Generates debug information |
| <code>--remarks</code> | Enables remarks |
| <code>--require_prototypes</code> | Verifies that prototypes are proper |
| <code>-s</code> | Optimizes for speed |
| <code>--silent</code> | Sets silent operation |
| <code>--stack_pointer_size</code> | Specifies how many bits of the stack pointer that the compiler will use for stack pointer arithmetics |
| <code>--strict_ansi</code> | Checks for strict compliance with ISO/ANSI C |

Table 31: Compiler options summary (Continued)

| Command line option | Description |
|-----------------------------|----------------------------------------------------------------|
| --warnings_affect_exit_code | Warnings affects exit code |
| --warnings_are_errors | Warnings are treated as errors |
| --weak_rtmodel_check | Modules compiled with different options may be linked together |
| -z | Optimizes for size |

Table 31: Compiler options summary (Continued)

Descriptions of options

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



Note that if you use the options page **Extra Options** to specify specific command line options, there is no check for consistency problems like conflicting options, duplication of options, or use of irrelevant options.

--bus_width --bus_width={20|24|28|32}

Parameters

| | |
|----------------|------------------------------------------------|
| 20, 24 | Available for the H8/300H microcomputer family |
| 20, 24, 28, 32 | Available for the H8S microcomputer family |

Description

Use this option to specify the width of the address bus for the device you are using. You must specify this option if you use the Advanced operating mode and compile a file that contains located variables. In the Normal operating mode, the bus width is assumed to be 16 bits, and this option cannot be specified.

By specifying the bus width, the compiler can use the correct and optimal addressing mode when addressing variables that have an absolute location.



Project>Options>General Options>Target>Address bus width

```
--char_is_signed --char_is_signed
```

Description

By default, the compiler interprets the `char` type as unsigned. Use this option to make the compiler interpret the `char` type as signed instead. This can be useful when you, for example, want to maintain compatibility with another compiler.

Note: The runtime library is compiled without the `--char_is_signed` option. If you use this option, you may get type mismatch warnings from the IAR XLINK Linker, because the library uses unsigned chars.



Project>Options>C/C++ Compiler>Language>Plain ‘char’ is

```
--code_model --code_model={small|large}
```

Parameters

| Parameter | Default function memory attribute | Max. module and program size | Pointer size | Available in mode |
|-----------|-----------------------------------|------------------------------|--------------|-------------------------|
| small | __code16 | 64 Kbytes | 2 bytes | Normal operating mode |
| large | __code24 | 16 Mbytes | 4 bytes | Advanced operating mode |

Table 32: Available code models

Description

The H8 IAR C/C++ Compiler supports placing code in different parts of memory by means of code models: the Small code model and the Large code model. The Small code model is default in the Normal operating mode and the Large code model is default in the Advanced operating mode. Use the `--code_model` option to select the code model for which the code is to be generated.

Note that all modules of your application must use the same code model.

See also

Basic settings for project configuration, page 5 and *Code models and memory attributes for function storage*, page 25.



In the IAR Embedded Workbench IDE, the appropriate code model will automatically be used based on the selected operating mode.

```
--core --core={H8300H|H8S}
```

Parameters

| | |
|------------------|-----------------------------------------------------|
| H8300H (default) | Generates code for the H8/300H microcomputer family |
| H8S | Generates code for the H8S microcomputer family |

Description

The H8 IAR C/C++ Compiler supports both the H8/300H and H8S microcomputer families. Use this option to select the processor core for which the code is to be generated.

See also

Basic settings for project configuration, page 5



Project>Options>General Options>Target>Core

```
-D -D symbol[=value]
```

Parameters

| | |
|---------------|--------------------------------------|
| <i>symbol</i> | The name of the preprocessor symbol |
| <i>value</i> | The value of the preprocessor symbol |

Description

Use this option to define a preprocessor symbol. If no value is specified, 1 is used. This option can be used one or more times on the command line.

The option `-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=
```

**Project>Options>C/C++ Compiler>Preprocessor>Defined symbols**`--data_model --data_model={small|huge}`**Parameters**

| Parameter | Default data memory attribute | Default data pointer | Available in mode | Placement of data |
|-----------------|-------------------------------|----------------------|-------------------------------------|-----------------------------------------|
| small (default) | __data16 | __data16 | Normal and Advanced operating modes | Low 32 Kbytes or high 32 Kbytes |
| huge | __data32 | __data32 | Advanced operating mode | 16 Mbyte for H8/300H and 4Gbyte for H8S |

*Table 33: Available data models***Description**

The H8 IAR C/C++ Compiler supports placing data in different parts of memory by means of code models: the Small data model and the Huge data model. Use the `--data_model` option to select the data model for which the code is to be generated.

Note that all modules of your application must use the same data model.

See also

Basic settings for project configuration, page 5 and *Data models*, page 14.

**Project>Options>General Options>Target>Data model**`--debug, -r --debug -r`**Description**

Use the `--debug` or `-r` option to make the compiler include information in the object modules that is useful to the IAR C-SPY® Debugger and other symbolic debuggers.

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

**Project>Options>C/C++ Compiler>Output>Generate debug information**

```
--dependencies --dependencies=[i|m] {filename|directory}
```

Parameters

| | |
|-------------|-------------------------------|
| i (default) | Lists only the names of files |
| m | Lists in makefile style |

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 145.

Description

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

Example

If `--dependencies` or `--dependencies=i` is used, the name of each opened source file, including the full path, if available, is output on a separate line. For example:

```
c:\iar\product\include\stdio.h
d:\myproject\include\foo.h
```

If `--dependencies=m` is used, the output uses makefile style. For each source file, one line containing a makefile dependency rule is produced. Each line consists of the name of the object file, a colon, a space, and the name of a source file. For example:

```
foo.r37: c:\iar\product\include\stdio.h
foo.r37: d:\myproject\include\foo.h
```

An example of using `--dependencies` with a popular make utility, such as `gmake` (GNU make):

- 1 Set up the rule for compiling files to be something like:

```
%.r37 : %.c
        $(ICC) $(ICCFLLAGS) $< --dependencies=m $*.d
```

That is, in addition to producing an object file, the command also produces a dependency file in makefile style (in this example, using the extension `.d`).

- 2 Include all the dependency files in the makefile using, for example:

```
-include $(sources:.c=.d)
```

Because of the dash (-) it works the first time, when the `.d` files do not yet exist.



This option is not available in the IAR Embedded Workbench IDE.

```
--diag_error --diag_error=tag, tag, ...
```

Parameters

| | |
|------------|--------------------------------------------------------------------------|
| <i>tag</i> | The number of a diagnostic message, for example the message number Pe117 |
|------------|--------------------------------------------------------------------------|

Description

Use this option to reclassify certain diagnostic messages as errors. An error indicates a violation of the C or C++ language rules, of such severity that object code will not be generated, and the exit code will be non-zero. This option may be used more than once on the command line.



Project>Options>C/C++ Compiler>Diagnostics>Treat these as errors

```
--diag_remark --diag_remark=tag, tag, ...
```

Parameters

| | |
|------------|--------------------------------------------------------------------------|
| <i>tag</i> | The number of a diagnostic message, for example the message number Pe177 |
|------------|--------------------------------------------------------------------------|

Description

Use this option to reclassify certain diagnostic messages as remarks. A remark is the least severe type of diagnostic message and indicates a source code construct that may cause strange behavior in the generated code. This option may be used more than once on the command line.

Note: By default, remarks are not displayed; use the `--remarks` option to display them.



Project>Options>C/C++ Compiler>Diagnostics>Treat these as remarks

```
--diag_suppress --diag_suppress=tag, tag, ...
```

Parameters

| | |
|------------|--------------------------------------------------------------------------|
| <i>tag</i> | The number of a diagnostic message, for example the message number Pe177 |
|------------|--------------------------------------------------------------------------|

Description

Use this option to suppress certain diagnostic messages. These messages will not be displayed. This option may be used more than once on the command line.



Project>Options>C/C++ Compiler>Diagnostics>Suppress these diagnostics

```
--diag_warning --diag_warning=tag, tag, ...
```

Parameters

| | |
|------------|--------------------------------------------------------------------------|
| <i>tag</i> | The number of a diagnostic message, for example the message number Pe826 |
|------------|--------------------------------------------------------------------------|

Description

Use this option to reclassify certain diagnostic messages as warnings. A warning indicates an error or omission that is of concern, but which will not cause the compiler to stop before compilation is completed. This option may be used more than once on the command line.

Example

The following example classifies the remark Pe826 as a warning:

```
--diag_warning=Pe826
```



Project>Options>C/C++ Compiler>Diagnostics>Treat these as warnings

```
--diagnostics_tables --diagnostics_tables {filename|directory}
```

Parameters

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 145.

Description

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.



This option is not available in the IAR Embedded Workbench IDE.

```
--direct_library_calls --direct_library_calls
```

Description

By default, many library routines will be called via the vector area, which means the addressing mode `@@aa:8` is used. When this option is used, all library routines will be called as normal subroutines instead of via the vector area.



Project>Options>C/C++ Compiler>Optimizations>Library calls>Direct

```
--dlib_config --dlib_config filename
```

Parameters

For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 145.

Description

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 `h8\lib`. For examples and a list of prebuilt runtime libraries, see *Using a prebuilt library*, page 54.

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 61.



To set the related options, select:

Project>Options>General Options>Library Configuration

```
--double --double={32|64}
```

Parameters

| | |
|--------------|-------------------------|
| 32 (default) | 32-bit doubles are used |
| 64 | 64-bit doubles are used |

Description

Use this option to select the precision used by the compiler for representing the floating-point types `double` and `long double`. The compiler can use either 32-bit or 64-bit precision. By default, the compiler uses 32-bit precision.

See also

Floating-point types, page 178.



Project>Options>General Options>Target>Size of type 'double'

`-e -e`

Description

In the command line version of the H8 IAR C/C++ Compiler, language extensions are disabled by default. If you use language extensions such as extended keywords and anonymous structs and unions in your source code, you must enable them by using this option.

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

See also

The chapter *Compiler extensions*.



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

`--ec++ --ec++`

Description

In the H8 IAR C/C++ Compiler, the default language is C. If you use Embedded C++ syntax in your source code, you must use this option to set the language the compiler uses to Embedded C++.



Project>Options>C/C++ Compiler>Language>Embedded C++

```
--eec++ --eec++
```

Description

In the H8 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 also

Extended Embedded C++, page 104.



Project>Options>C/C++ Compiler>Language>Extended Embedded C++

```
--enable_mac --enable_mac
```

Description

The H8 IAR C/C++ Compiler can take advantage of the MAC instructions available in the H8/300H and H8S microcomputer families. Use the `--enable_mac` option to make the compiler enable the MAC register and thereby, when possible, use the MAC instructions.



Project>Options>General Options>Target>MAC

```
--enable_multibytes --enable_multibytes
```

Description

By default, multibyte characters cannot be used in C or C++ source code. Use this option to make multibyte characters in the source code be interpreted according to the host computer's default setting for multibyte support.

Multibyte characters are allowed in C and C++ style comments, in string literals, and in character constants. They are transferred untouched to the generated code.



Project>Options>C/C++ Compiler>Language>Enable multibyte support

```
--error_limit --error_limit=n
```

Parameters

n The number of errors before the compiler stops the compilation. *n* must be a positive integer; 0 indicates no limit.

Description

Use this option to specify the search path for `#include` files. This option may be used more than once on the command line.

See also

Include file search procedure, page 136.



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

```
--interrupt_mode --interrupt_mode={0|1|2|3}
```

Parameters

| | |
|------------|------------------------------------------------------------------------------------|
| 0, 1, 2, 3 | One of the interrupt modes available in the H8/300H and H8S microcomputer families |
|------------|------------------------------------------------------------------------------------|

Description

The H8 IAR C/C++ Compiler supports the different interrupts modes available in the H8/300H and H8S microcomputer families. Use the `--interrupt_mode` option to specify the interrupt mode in use.

See also

Interrupt functions, page 26, and *Monitor functions*, page 28.



Project>Options>General Options>Target>Interrupt mode

```
-1 -l[a|A|b|B|c|C|D][N][H] {filename|directory}
```

Parameters

| | |
|---|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| a | Assembler list file |
| A | Assembler list file with C or C++ source as comments |
| b | Basic assembler list file. This file has the same contents as a list file produced with <code>-la</code> , except that no extra compiler generated information (runtime model attributes, call frame information, frame size information) is included* |

| | |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| a | Assembler list file |
| B | Basic assembler list file. This file has the same contents as a list file produced with <code>-lA</code> , except that no extra compiler-generated information (runtime model attributes, call frame information, frame size information) is included * |
| c | C or C++ list file |
| C (default) | C or C++ list file with assembler source as comments |
| D | C or C++ list file with assembler source as comments, but without instruction offsets and hexadecimal byte values |
| N | No diagnostics in file |
| H | Include source lines from header files in output. Without this option, only source lines from the primary source file are included |

* This makes the list file less useful as input to the assembler, but more useful for reading by a human.

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 145.

Description

Use this option to generate an assembler or C/C++ listing to a file. Note that this option can be used one or more times on the command line.



To set related options, select **Project>Options>C/C++ Compiler>List**

`--library_module` `--library_module`

Description

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



Project>Options>C/C++ Compiler>Output>Output file>Library

```
--max_cycles_no_interrupt --max_cycles_no_interrupt=n
```

Parameters

n The maximum number of cycles during which interrupts may be disabled. The default value 0 prohibits the compiler from using the `EEPMOV.B` instruction.

Description

By default, the compiler does not use the `EEPMOV.B` instruction. Use this option to specify the maximum number of cycles during which interrupts may be disabled. If the compiler is allowed to have interrupts disabled for a period, it can generate code containing the `EEPMOV.B` instruction.



Project>Options>C/C++ Compiler>Optimizations>Generate EEPMOV

and

Project>Options>C/C++ Compiler>Optimizations>Max cycles

```
--migration_preprocessor_extensions --migration_preprocessor_extensions
```

Description

If you need to migrate code from an earlier IAR C or C/C++ compiler, you may want to use this option. Use this option to use the following in preprocessor expressions:

- Floating-point expressions
- Basic type names and `sizeof`
- All symbol names (including typedefs and variables).

Note: If you use this option, not only will the compiler accept code that is not standards 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.



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

```
--misrac --misrac[={tag1,tag2-tag3,...|all|required}]
```

Parameters

| | |
|-----------------------------------------------------------|-----------------------------------------------------------------------------------------------------|
| <code>--misrac=<i>n</i></code> | Enables checking for the MISRA C rule with number <i>n</i> |
| <code>--misrac=<i>o</i>,<i>n</i></code> | Enables checking for the MISRA C rules with numbers <i>o</i> and <i>n</i> |
| <code>--misrac=<i>o</i>-<i>p</i></code> | Enables checking for all MISRA C rules with numbers from <i>o</i> to <i>p</i> |
| <code>--misrac=<i>m</i>,<i>n</i>,<i>o</i>-<i>p</i></code> | Enables checking for MISRA C rules with numbers <i>m</i> , <i>n</i> , and from <i>o</i> to <i>p</i> |
| <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 |

Description

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 (1998)*. By using one or more arguments with the option, you can restrict the checking to a specific subset of the MISRA C rules. 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 related options, select:

Project>Options>General Options>MISRA C or **Project>Options>C/C++ Compiler>MISRA C**

```
--misrac_verbose --misrac_verbose
```

Description

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.



Project>Options>General Options>MISRA C>Log MISRA C Settings

```
--module_name --module_name=name
```

Parameters

name An explicit object module name

Description

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

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



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

```
--no_code_motion --no_code_motion
```

Description

Use this option to disable code motion optimizations. 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.



Project>Options>C/C++ Compiler>Optimization>Enable transformations>Code motion

```
--no_cse --no_cse
```

Description

Use this option 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.



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

`--no_inline` `--no_inline`

Description

Use this option to disable function inlining. Function inlining means that a simple function, whose definition is known at compile time, is integrated into the body of its caller to eliminate the overhead of the call.

This optimization, which is performed at optimization level 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 that for size.

Note: This option has no effect at optimization levels below 9.



Project>Options>C/C++ Compiler>Optimization>Enable transformations>Function inlining

`--no_path_in_file_macros` `--no_path_in_file_macros`

Description

Use this option to exclude the path from the return value of the predefined preprocessor symbols `__FILE__` and `__BASE_FILE__`.



This option is not available in the IAR Embedded Workbench IDE.

`--no_tbaa` `--no_tbaa`

Description

Use this option to disable type-based alias analysis. When this options is not used, the compiler is free to assume that objects are only accessed through the declared type or through `unsigned char`.

See also

Type-based alias analysis, page 120.



Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Type-based alias analysis

```
--no_typedefs_in_diagnostics --no_typedefs_in_diagnostics
```

Description

Use this option to disable the use of typedef names in diagnostics. Normally, when a type is mentioned in a message from the compiler, most commonly in a diagnostic message of some kind, the typedef names that were used in the original declaration are used whenever they make the resulting text shorter.

Example

```
typedef int (*MyPtr)(char const *);
MyPtr p = "foo";
```

will give an error message like the following:

```
Error[Pe144]: a value of type "char *" cannot be used to
initialize an entity of type "MyPtr"
```

If the `--no_typedefs_in_diagnostics` option is used, the error message will be like this:

```
Error[Pe144]: a value of type "char *" cannot be used to
initialize an entity of type "int (*)(char const *)"
```



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

```
--no_unroll --no_unroll
```

Description

Use this option to disable loop unrolling. The code body of a small loop, whose number of iterations can be determined at compile time, is duplicated to reduce the loop overhead.

For small loops, the overhead required to perform the looping can be large compared with the work performed in the loop body.

The loop unrolling optimization duplicates the body several times, reducing the loop overhead. The unrolled body also opens up for other optimization opportunities.

This optimization, which is performed at optimization level 9, normally reduces execution time, but increases code size. The resulting code may also be difficult to debug.

The compiler heuristically decides which loops to unroll. Different heuristics are used when optimizing for speed and size.

Note: This option has no effect at optimization levels below 9.



Project>Options>C/C++ Compiler>Optimization>Enable transformations>Loop unrolling

--no_warnings --no_warnings

Description

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



This option is not available in the IAR Embedded Workbench IDE.

--no_wrap_diagnostics --no_wrap_diagnostics

Description

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.



This option is not available in the IAR Embedded Workbench IDE.

-o -o {filename|directory}

Parameters

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 145.

Description

By default, the object code output produced by the compiler is located in a file with the same name as the source file, but with the extension `.obj`. Use this option to explicitly specify a different output filename for the object code output.



Project>Options>General Options>Output>Output directories>Object files

```
--omit_types --omit_types
```

Description

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



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

```
--only_stdout --only_stdout
```

Description

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



This option is not available in the IAR Embedded Workbench IDE.

```
--operating_mode --operating_mode={normal|advanced}
```

Parameters

| | |
|-----------------------|--------------------------------------|
| <code>normal</code> | The Normal operating mode is used. |
| <code>advanced</code> | The Advanced operating mode is used. |

Description

The H8 IAR C/C++ Compiler supports the different operating modes—Normal and Advanced—available in the H8/300H and H8S microcomputer families. Use the `--operating_mode` option to specify the operating mode in use.

See also

Basic settings for project configuration, page 5.



Project>Options>General Options>Target>Operating mode

`--preinclude` `--preinclude includefile`

Parameters

For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 145.

Description

Use this option to make the compiler include the specified include file before it starts to read the source file. This is useful if you want to change something in the source code for the entire application, for instance if you want to define a new symbol.



Project>Options>C/C++ Compiler>Preprocessor>Preinclude file

`--preprocess` `--preprocess [= [c] [n] [l]] {filename|directory}`

Parameters

| | |
|----------------|---------------------------|
| <code>c</code> | Preserve comments |
| <code>n</code> | Preprocess only |
| <code>l</code> | Generate #line directives |

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 145.

Description

Use this option to direct preprocessor output to a named file.



Project>Options>C/C++ Compiler>Preprocessor>Preprocessor output to file

`--public_equ` `--public_equ symbol [= value]`

Parameters

| | |
|---------------|---------------------------------------------------|
| <i>symbol</i> | The name of the assembler symbol to be defined |
| <i>value</i> | An optional value of the defined assembler symbol |

Description

This option is equivalent to defining a label in assembler language by using the `EQU` directive and exporting it using the `PUBLIC` directive. This option may be used more than once on the command line.



This option is not available in the IAR Embedded Workbench IDE.

`-r, --debug` `-r`
`--debug`

Description

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

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



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

`--remarks` `--remarks`

Description

The least severe diagnostic messages are called remarks. A remark indicates a source code construct that may cause strange behavior in the generated code. By default, the compiler does not generate remarks. Use this option to make the compiler generate remarks.

See also

Severity levels, page 140.



Project>Options>C/C++ Compiler>Diagnostics>Enable remarks

`--require_prototypes` `--require_prototypes`

Description

Use this option to force the compiler to verify that all functions have proper prototypes. Using this option means that code containing any of the following will generate an error:

- A function call of a function with no declaration, or with a Kernighan & Ritchie C declaration
- A function definition of a public function with no previous prototype declaration

- An indirect function call through a function pointer with a type that does not include a prototype.



Project>Options>C/C++ Compiler>Language>Require prototypes

-s -s[2|3|6|9]

Parameters

| | |
|-------------|-----------------------------|
| 2 | None* (Best debug support) |
| 3 (default) | Low* |
| 6 | Medium |
| 9 | High (Maximum optimization) |

***The most important difference between -s2 and -s3 is that at level 2, all non-static variables will live during their entire scope.**

Description

Use this option to make the compiler optimize the code for maximum execution speed. If no optimization option is specified, the optimization level 3 is used by default.

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.



Project>Options>C/C++ Compiler>Optimization>Speed

--silent --silent

Description

By default, the compiler issues introductory messages and a final statistics report. Use this option to make the compiler operate without sending these messages to the standard output stream (normally the screen).

This option does not affect the display of error and warning messages.



This option is not available in the IAR Embedded Workbench IDE.

```
--stack_pointer_size --stack_pointer_size={8|16|32}
```

Parameters

8, 16, 32 The number of bits of the stack pointer that the compiler will use for stack pointer arithmetics.

Description

Use this option to inform the compiler of how many bits of the stack pointer the compiler must update when performing stack pointer arithmetics, or in other words, how many bits of the stack pointer that can change while the application executes. If you carefully consider the stack size, its placement in memory, and the use of this option, the compiler can produce more compact code.

When you use this option, you should be aware of the following issues:

- You must make sure that the stack does not cross a 64-Kbyte (for 16-bit arithmetics) or 256-byte (for 8-bit arithmetics) page boundary
- Specifying the value 32 in the Normal operating mode will silently be interpreted as the value 16
- Even if specifying the value 32 in the Advanced operating mode, only 16 bits of the stack pointer will be used when using the Small data model
- There is no runtime model attribute that protects you from mixing modules compiled with different settings on this option
- You are not recommended to use this option when compiling code that will be used in more than one application, for example when building libraries. If a small value is set when compiling the library code, and a too large stack is used in the application using the library, you might get errors that are hard to find.

See also

Stack pointer arithmetics, page 127.



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

```
--strict_ansi --strict_ansi
```

By default, the compiler accepts a relaxed superset of ISO/ANSI C, see *Minor language extensions*, page 194. Use this option to ensure that the program conforms to the ISO/ANSI C standard.

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



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

`--warnings_affect_exit_code` `--warnings_affect_exit_code`

Description

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.



This option is not available in the IAR Embedded Workbench IDE.

`--warnings_are_errors` `--warnings_are_errors`

Description

Use this option to make the compiler treat all warnings as errors. If the compiler encounters an error, no object code is generated. Warnings that have been changed into remarks are not treated as errors.

Note: Any diagnostic messages that have been reclassified as warnings by the compiler option `--diag_warning` or the `#pragma diag_warning` directive will also be treated as errors when `--warnings_are_errors` is used.

See also

`--diag_warning`, page 154 and `#pragma diag_warning`, page 219.



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

`--weak_rtmodel_check` `--weak_rtmodel_check`

Description

The H8 IAR C/C++ Compiler provides a robust system for ensuring module consistency. Use this option to use a weaker consistency check.

In some situations, you may want a weaker check. For example, if you build a library for the H8/300 core, you may want it to be used in H8S projects as well. This is possible, as the H8S architecture is backward compatible with the H8/300H architecture. By specifying the `--weak_rtmodel_check` option, code compiled for H8/300H will be accepted in H8S projects.

See also

Using a weak runtime model check, page 80.



This option will automatically be set when building a library,
Project>Options>General Options>Output>Output file>Library

`-z` `-z[2|3|6|9]`

Parameters

| | |
|-------------|-----------------------------|
| 2 | None* (Best debug support) |
| 3 (default) | Low* |
| 6 | Medium |
| 9 | High (Maximum optimization) |

***The most important difference between `-z2` and `-z3` is that at level 2, all non-static variables will live during their entire scope.**

Description

Use this option to make the compiler optimize the code for minimum size. If no optimization option is specified, the optimization level 3 is used by default.

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.



Project>Options>C/C++ Compiler>Optimizations>Size

Data representation

This chapter describes the data types, pointers, and structure types supported by the H8 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 H8 IAR C/C++ COMPILER

The H8/300H and H8S microcomputer can access memory using 8-, 16-, or 32-bit operations. However, when a 16- or 32-bit access is performed, the data must be located at an even address. The H8 IAR C/C++ Compiler ensures this by assigning an alignment to every data type, ensuring that the H8/300H and H8S microcomputer will be able to read the data. A 16- or 32-bit access to an odd address will not work.

Basic data types

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

INTEGER TYPES

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

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

Table 34: Integer types

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

Bool

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

The enum type

The compiler will use the smallest type required to hold enum constants, preferring signed rather than unsigned.

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

To make the compiler use a larger type than it would automatically use, define an enum constant with a large enough value. For example,

```
/* Disables usage of the char type for enum */
enum Cards{Spade1, Spade2,
DontUseChar=257};
```

The char type

The char type is by default unsigned in the compiler, but the `--char_is_signed` compiler option allows you to make it signed. Note, however, that the library is compiled with the char type as unsigned.

The wchar_t type

The wchar_t data type is an integer type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locals.

The wchar_t data type is supported by default in the C++ language. To use the wchar_t type also in C source code, you must include the file `stddef.h` from the runtime library.

Bitfields

In ISO/ANSI C, int and unsigned int can be used as the base type for integer bitfields. In the H8 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 H8 IAR C/C++ Compiler, floating-point values are represented in standard IEEE 754 format. The sizes for the different floating-point types are:

| Type | Size if <code>--double=32</code> | Size if <code>--double=64</code> |
|-------------|----------------------------------|----------------------------------|
| float | 32 bits | 32 bits |
| double | 32 bits (default) | 64 bits |
| long double | 32 bits (default) | 64 bits |

Table 35: Floating-point types

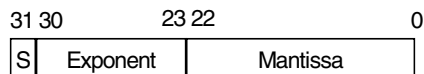
Note: The size of `double` and `long double` depends on the `--double={32|64}` option, see `--double`, page 155. The type `long double` uses the same precision as `double`.

The compiler does not support subnormal numbers. All operations that should produce subnormal numbers will instead generate zero.

Exception flags according to the IEEE 754 standard are not supported.

32-bit floating-point format

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



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

The value of the number is:

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

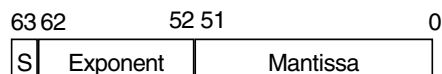
The range of the number is:

$$\pm 1.18\text{E}-38 \text{ to } \pm 3.39\text{E}+38$$

The precision of the float operators (+, -, *, and /) is approximately 7 decimal digits.

64-bit floating-point format

The representation of a 64-bit floating-point number as an integer is:



The exponent is 11 bits, and the mantissa is 52 bits.

The value of the number is:

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

The range of the number is:

$$\pm 2.23\text{E}-308 \text{ to } \pm 1.79\text{E}+308$$

The precision of the float operators (+, -, *, and /) is approximately 15 decimal digits.

Representation of special floating-point numbers

The following list describes the representation of special floating-point numbers:

- Zero is represented by zero mantissa and exponent. The sign bit signifies positive or negative zero.
- Infinity is represented by setting the exponent to the highest value and the mantissa to zero. The sign bit signifies positive or negative infinity.
- Not a number (NaN) is represented by setting the exponent to the highest positive value and the most significant bit in the mantissa to 1. The value of the sign bit is ignored.

Pointer types

The H8 IAR C/C++ Compiler has two basic types of pointers: functions pointers and data pointers.

FUNCTION POINTERS

The size of function pointers is always 16 or 32 bits, and they can address the entire memory. The internal representation of a function pointer is the actual address it refers to.

The following function pointers are available:

| Keyword | Address range* | Pointer size | Description |
|-----------------------|----------------|--------------|---------------------------------------------------------------------------------|
| <code>__code16</code> | 0-0xFFFF | 2 bytes | No restrictions on code placement. Can be used in Normal operating mode only. |
| <code>__code24</code> | 0-0xFFFFFFFF | 4 bytes | No restrictions on code placement. Can be used in Advanced operating mode only. |

Table 36: Function pointers

* The value 0 denotes the NULL pointer.

DATA POINTERS

Data pointers have three sizes: 8, 16, or 32 bits. The following data pointers are available:

| Keyword | Pointer size | Index type | Pointer value range ¹⁾ | Address range |
|-----------------------|--------------|-----------------|-----------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>__data8</code> | 1 byte | signed char | 0x0-0xFF | Normal operating mode: 0xFF00-0xFFFF Advanced operating mode H8/300H: 0xFFFF00-0xFFFFFFFF Advanced operating mode H8S: 0xFFFFFFFF00-0xFFFFFFFF |
| <code>__data16</code> | 2 bytes | signed short | 0x0000- 0xFFFF | Normal operating mode: 0x0-0xFFFF Advanced operating mode H8/300H: 0x0-0x7FFF, 0xFF8000-0xFFFFFFFF Advanced operating mode H8S: 0x0-0x7FFF, 0xFFFF8000-0xFFFFFFFF |
| <code>__data32</code> | 4 bytes | signed long | 0x00000000- 0xFFFFFFFF ²⁾ | Advanced operating mode H8/300H: 0x0-0xFFFFFFFF Advanced operating mode H8S: 0x0-0xFFFFFFFF |

Table 37: Data pointers

¹⁾ The value 0 denotes the NULL pointer.

²⁾ Note that, in runtime, the compiler may represent a physical address in more than one way. For example, when a `__data16` pointer is converted to a `__data32` pointer, the pointer value is sign-extended causing a `__data16` pointer value of 0x8000 to be extended to 0xFFFF8000 even though the actual memory location has the address 0xFF8000 because the device only implements 24 address bits. In most cases, the compiler will handle this automatically, but in some cases you may need to be aware of the internal representation and the details of how pointer values are converted at casts.

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 unsigned integer type to a pointer of a larger type is performed by zero extension
- Casting a *value* of a signed integer type to a pointer of a larger type is performed by sign extension
- Casting a *pointer type* to a smaller integer type is performed by truncation

- Casting a *pointer type* to a larger integer type is performed by first casting the pointer to an unsigned integer type of the same size as the pointer and then, if necessary, by zero extension
- Casting a *data pointer* to a function pointer and vice versa is illegal
- Casting a *function pointer* to an integer type gives an undefined result
- Casting a `__data8` pointer to a larger data pointer type is performed by extending the 8-bit value with bits that have the value 1
- Casting a `__data16` pointer to a `__data32` pointer is performed by sign extension
- Casting a data pointer type to a smaller data pointer type is performed by truncation.

Note: If a `__data32` pointer that points to the topmost 32 Kbyte of memory is converted to a `__data16` pointer and then back to a `__data32` pointer, the value will in most cases not be equal to the original `__data32` pointer value. After the cast to the `__data16` pointer and back, the `__data32` pointer has been sign-extended and has the value 1 in all bit positions above the bus width for the device. The new pointer value will work correctly when accessing the original variable. However, in a comparison between the original `__data32` pointer and the converted `__data32` pointer, the pointers will not be equal. The same problem is also present when casting from a `__data32` pointer to a `__data8` pointer and back.

size_t

`size_t` is the unsigned integer type required to hold the maximum size of an object. In the H8 IAR C/C++ Compiler, the size of `size_t` is 16 bits (`unsigned short`) in the Small data model, and 32 bits (`unsigned long`) in the Huge data model.

ptrdiff_t

`ptrdiff_t` is the type of the signed integer required to hold the difference between two pointers to elements of the same array. In the H8 IAR C/C++ Compiler, the size of `ptrdiff_t` is 16 bits (`signed short`) in the Small data model, and 32 bits (`signed long`) in the Huge data model.

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 H8 IAR C/C++ Compiler, the size of `intptr_t` is 16 bits (`signed short`) in the Small data model, and 32 bits (`signed long`) in the Huge data model.

uintptr_t

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

Structure types

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

ALIGNMENT

The `struct` and `union` types inherit the alignment requirements of their members, which means the `struct` and `union` have the same alignment as the member with the highest alignment requirement. 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).

Example

```
struct {
    short s; /* stored in byte 0 and 1 */
    char c; /* stored in byte 2 */
    long l; /* stored in byte 4, 5, 6, and 7 */
    char c2; /* stored in byte 8 */
} s;
```

The following diagram shows the layout in memory:

| | | | | | |
|----------------|---------------|---------------|----------------|----------------|---------------|
| s.s 2 bytes | s.c 1 byte | pad 1 byte | s.l 4 bytes | s.c2 1 byte | pad 1 byte |
|----------------|---------------|---------------|----------------|----------------|---------------|

The alignment of the structure is 2 bytes, and its size is 10 bytes.

PACKED STRUCTURE TYPES

The `#pragma pack` directive is used for changing the alignment requirements of the members of a structure. This will change the way the layout of the structure is performed. The members will be placed in the same order as when declared, but there might be less pad space between members.

Example

```
#pragma pack(1)
struct {
    short s;
    char c;
    long l;
    char c2;
} s;
```

will be placed:



For more information, see *Rearranging elements in a structure*, page 123.

Type qualifiers

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

DECLARING OBJECTS VOLATILE

There are three main reasons for declaring an object `volatile`:

- Shared access; the object is shared between several tasks in a multitasking environment
- Trigger access; as for a memory-mapped SFR where the fact that an access occurs has an effect
- Modified access; where the contents of the object can change in ways not known to the compiler.

Definition of access to volatile objects

The ISO/ANSI standard defines an abstract machine, which governs the behavior of accesses to `volatile` declared objects. In general and in accordance to the abstract machine, the compiler:

- Considers each read and write access to an object that has been declared `volatile` as an access
- The unit for the access is either the entire object or, for accesses to an element in a composite object—such as an array, struct, class, or union—the element. For example:

```
char volatile a;
a = 5; /* A write access */
a += 6; /* First a read then a write access */
```

- An access to a bitfield is treated as an access to the underlying type.

However, these rules are not detailed enough to handle the hardware-related requirements. The rules specific to the H8 IAR C/C++ Compiler are described below.

Rules for accesses

In the H8 IAR C/C++ Compiler, 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, they cannot be interrupted.

The H8 IAR C/C++ Compiler adheres to these rules for all data objects with the size 8, 16, or 32 bits. These data objects are accessed with a single `MOV.B/W/L` instruction.

For single bit bitfields, the compiler will attempt to access the bitfield using a bit instruction. However, complex operations on the bitfield might read or write the entire underlying type in which the bit is located.

For all other object types, only rule number one applies.

DECLARING OBJECTS CONST

The `const` type qualifier is used for indicating that a data object, accessed directly or via a pointer, is non-writable. A pointer to `const` declared data can point to both constant and non-constant objects. It is good programming practice to use `const` declared pointers whenever possible because this improves the compiler's possibilities to optimize the generated code and reduces the risk of application failure due to erroneously modified data.

Static and global objects declared `const` and located in the memories `data16` and `data32` are allocated in ROM. For `data8`, the objects are allocated in RAM and initialized by the runtime system at startup.

In C++, objects that require runtime initialization cannot be placed in ROM.

Data types in C++

In C++, all plain C data types are represented in the same way as described earlier in this chapter. However, if any Embedded C++ features are used for a type, no assumptions can be made concerning the data representation. This means, for example, that it is not legal to write assembler code that accesses class members.

Compiler extensions

This chapter gives a brief overview of the H8 IAR C/C++ Compiler extensions to the ISO/ANSI C standard. All extensions can also be used for the C++ programming language. More specifically the chapter describes the available C language extensions.

Compiler extensions overview

The compiler offers the standard features of ISO/ANSI C as well as a wide set of extensions, ranging from features specifically tailored for efficient programming in the embedded industry to the relaxation of some minor standards issues.

You can find the extensions available as:

- C/C++ language extensions

For a summary of available language extensions, see *C language extensions*, page 188. For reference information about the extended keywords, see the chapter *Extended keywords*. For information about C++, the two levels of support for the language, and C++ language extensions; see the chapter *Using C++*.

- Pragma directives

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

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

- Preprocessor extensions

The preprocessor of the compiler adheres to the ISO/ANSI standard. In addition, the compiler also makes a number of preprocessor-related extensions available to you. For more information, see the chapter *The preprocessor*.

- Intrinsic functions

The intrinsic functions provide direct access to low-level processor operations and can be very useful in, for example, time-critical routines. The intrinsic functions compile into inline code, either as a single instruction or as a short sequence of instructions. To read more about using intrinsic functions, see *Mixing C and assembler*, page 83. For a list of available functions, see the chapter *Intrinsic functions*.

- Library functions

The IAR DLIB Library provides most of the important C and C++ library definitions that apply to embedded systems. In addition, the library also provides some extensions, partly taken from the C99 standard. For more information, see *IAR DLIB Library*, page 256.

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

ENABLING LANGUAGE EXTENSIONS



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

For information about how to enable and disable language extensions from the command line, see the compiler options *-e*, page 156 and *--strict_ansi*, page 171.

C language extensions

This section gives a brief overview of the C language extensions available in the H8 IAR C/C++ Compiler. The compiler provides a wide set of extensions, so to help you to find the extensions required by your application, the extensions have been grouped according to their expected usefulness. In short, this means:

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

IMPORTANT LANGUAGE EXTENSIONS

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

- Memory attributes, type attributes, and object attributes

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

- Placement at an absolute address or in a named segment

The @ operator or the directive #pragma location can be used for placing global and static variables at absolute addresses, or placing a variable or function in a named segment. For more information about using these primitives, see *Controlling data and function placement in memory*, page 45 and *#pragma location*, page 220.

- Alignment

Each data type has its own alignment, for more details, see *Alignment*, page 175. If you want to change alignment, the #pragma pack and #pragma data_alignment directives are available. If you want to check the alignment of an object, use the __ALIGNOF__ () operator.

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

- __ALIGNOF__ (type)
- __ALIGNOF__ (expression)

In the second form, the expression is not evaluated.

- Anonymous structs and unions

C++ includes a feature named anonymous unions. The compiler allows a similar feature for both structs and unions in the C programming language. For more information, see *Anonymous structs and unions*, page 124.

- Bitfields and non-standard types

In ISO/ANSI C, a bitfield must be of the type int or unsigned int. Using IAR Systems language extensions, any integer type or enum may be used. The advantage is that the struct will be smaller. This matches G.5.8 in the appendix of the ISO standard, *ISO Portability Issues*. For more information, see *Bitfields*, page 177.

- Dedicated segment operators __segment_begin and __segment_end

The syntax for these operators are:

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

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

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

In the following example, the type of the __segment_begin operator is void __data16 *.

```
#pragma segment="MYSEG" __data16
...
segment_start_address = __segment_begin("MYSEG");
```

See also #pragma segment, page 225 and #pragma location, page 220.

USEFUL LANGUAGE EXTENSIONS

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

- Inline functions

The #pragma inline directive, alternatively the inline keyword, advises the compiler that the function whose declaration follows immediately after the directive should be inlined. This is similar to the C++ keyword inline. For more information, see #pragma inline, page 219.

- Mixing declarations and statements

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

- Declaration in for loops

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

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

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

- The `bool` data type

To use the `bool` type in C source code, you must include the file `stdbool.h`. This feature is part of the C99 standard and C++. (The `bool` data type is supported by default in C++.)

- C++ style comments

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

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

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

Inline assembler

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

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

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

The syntax is:

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

The string can be a valid assembler instruction or 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 83.

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++ and Extended EC++.
- This feature is part of the C99 standard.

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

Example

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

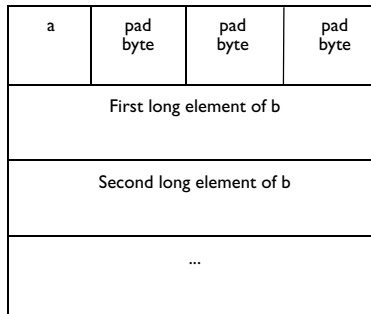
struct str * GetAStr(int size)
{
    return malloc(sizeof(struct str) +
                 sizeof(unsigned long) * size);
}

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

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.



This feature is copied from the C99 standard.

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. This feature is copied from the C99 standard.

Examples

`0x1p0` is 1

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

Designated initializers in structures and arrays

Any initialization of either a structure (`struct` or `union`) or an array can have a designation. A designation consists of one or more designators followed by an initializer. A designator for a structure is specified as `.elementname` and for an array [`constant index expression`]. Using designated initializers is not supported in C++.

Examples

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

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

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

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

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

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

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

MINOR LANGUAGE EXTENSIONS

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

- Arrays of incomplete types

An array may have an incomplete `struct`, `union`, or `enum` type as its element type. The types must be completed before the array is used (if it is), or by the end of the compilation unit (if it is not).

- Forward declaration of 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.

- Missing semicolon at end of struct or union specifier

A warning is issued if the semicolon at the end of a `struct` or union specifier is missing.

- NULL and void

In operations on pointers, a pointer to `void` is always implicitly converted to another type if necessary, and a null pointer constant is always implicitly converted to a null pointer of the right type if necessary. In ISO/ANSI C, some operators allow such things, while others do not allow them.

- Casting pointers to integers in static initializers

In an initializer, a pointer constant value may be cast to an integral type if the integral type is large enough to contain it. For more information about casting pointers, see *Casting*, page 180.

- Taking the address of a register variable

In ISO/ANSI C, it is illegal to take the address of a variable specified as a register variable. The compiler allows this, but a warning is issued.

- Duplicated size and sign specifiers

Should the size or sign specifiers be duplicated (for example, `short short` or `unsigned unsigned`), an error is issued.

- `long float` means double

The type `long float` is accepted as a synonym for `double`.

- Repeated `typedef` declarations

Redeclarations of `typedef` that occur in the same scope are allowed, but a warning is issued.

- Mixing pointer types

Assignment and pointer difference is allowed between pointers to types that are interchangeable but not identical; for example, `unsigned char *` and `char *`. This includes pointers to integral types of the same size. A warning is issued.

Assignment of a string constant to a pointer to any kind of character is allowed, and no warning will be issued.

- Non-top level `const`

Assignment of pointers is allowed in cases where the destination type has added type qualifiers that are not at the top level (for example, `int **` to `int const **`). Comparing and taking the difference of such pointers is also allowed.

- Non-`lvalue` arrays

A non-`lvalue` array expression is converted to a pointer to the first element of the array when it is used.

- 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.

- Comments at the end of preprocessor directives

This extension, which makes it legal to place text after preprocessor directives, is enabled, unless strict ISO/ANSI mode is used. The purpose of this language extension is to support compilation of legacy code; we do *not* recommend that you write new code in this fashion.

- An extra comma at the end of `enum` lists

Placing an extra comma is allowed at the end of an `enum` list. In strict ISO/ANSI mode, a warning is issued.

- A label preceding a `}`

In ISO/ANSI C, a label must be followed by at least one statement. Therefore, it is illegal to place the label at the end of a block. In the H8 IAR C/C++ Compiler, a warning is issued.

Note: This also applies to the labels of `switch` statements.

- Empty declarations

An empty declaration (a semicolon by itself) is allowed, but a remark is issued (provided that remarks are enabled).

- Single-value initialization

ISO/ANSI C requires that all initializer expressions of static arrays, `structs`, and `unions` are enclosed in braces.

Single-value initializers are allowed to appear without braces, but a warning is issued. In the H8 IAR C/C++ Compiler, the following expression is allowed:

```
struct str
{
    int a;
} x = 10;
```


- Declarations in other scopes

External and static declarations in other scopes are visible. In the following example, the variable `y` can be used at the end of the function, even though it should only be visible in the body of the `if` statement. A warning is issued.

```
int test(int x)
{
    if (x)
    {
        extern int y;
        y = 1;
    }

    return y;
}
```

- Expanding function names into strings with the function as context

Use any of the symbols `__func__` or `__FUNCTION__` inside a function body to make it expand into a string, with the function name as context. Use the symbol `__PRETTY_FUNCTION__` to also include the parameter types and return type. The result might, for example, look like this if you use the `__PRETTY_FUNCTION__` symbol:

```
"void func(char) "
```

These symbols are useful for assertions and other trace utilities and they require that language extensions are enabled, see `-e`, page 156.

Extended keywords

This chapter describes the extended keywords that support specific features of the H8/300H and H8S microcomputer and the general syntax rules for the keywords. Finally, the chapter gives a detailed description of each keyword.

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

General syntax rules for extended keywords

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

The H8 IAR C/C++ Compiler provides a set of attributes that can be used on functions or data objects to support specific features of the H8/300H and H8S microcomputer families. There are two types of attributes—*type attributes* and *object attributes*:

- Type attributes affect the *external functionality* of the data object or function
- Object attributes affect the *internal functionality* of the data object or function.

The syntax for the keywords differs slightly depending on whether it is a type attribute or an object attribute, and whether it is applied to a data object or a function.

For information about how to use attributes to modify data, see the chapter *Data storage*. For information about how to use attributes to modify functions, see the chapter *Functions*. For detailed information about each attribute, see *Descriptions of extended keywords*, page 204.

Note: The extended keywords are only available when language extensions are enabled in the H8 IAR C/C++ Compiler.



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



Use the `-e` compiler option to enable language extensions. See `-e`, page 156 for additional information.

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 microcomputer.

- Available *function memory attributes*: `__code16` and `__code24`
- Available *data memory attributes*: `__data8`, `__data16`, `__data32`, and `__bitvar`

Data objects, functions, and destinations of pointers or C++ references always have a memory attribute. If no attribute is explicitly specified in the declaration or by the pragma directive `#pragma type_attribute`, an appropriate default attribute is used. You can only specify one memory attribute for each level of pointer indirection.

General type attributes

The following general type attributes are available:

- *Function type attributes* change the calling convention of a function:
`__interrupt`, `__monitor`, `__trap`, `__cc_version1`, `__cc_version2`, `__cc_version3`, and `__vector_call`
- *Data type attributes*: `const` and `volatile`

You can specify as many type attributes as required for each level of pointer indirection.

To read more about the type qualifiers `const` and `volatile`, see *Type qualifiers*, page 183.

Syntax for type attributes used on data objects

In general, type attributes for data objects follow the same syntax as the type qualifiers `const` and `volatile`.

The following declaration assigns the `__data16` type attribute to the variables `i` and `j`; in other words, the variable `i` and `j` is placed in `data16` memory. The variables `k` and `l` behave in the same way:

```
__data16 int i, j;
int __data16 k, l;
```

Note that the attribute affects both identifiers.

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

```
#pragma type_attribute=__data16
int i, j;
```

The advantage of using pragma directives for specifying keywords is that it offers you a method to make sure that the source code is portable. Note that the pragma directive has no effect if a memory attribute is already explicitly declared.

For more examples of using memory attributes, see *More examples*, page 20.

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

```
typedef char __data16 Byte;
typedef Byte *BytePtr;
Byte b;
BytePtr bp;
```

and

```
__data16 char b;
char __data16 *bp;
```

Note that `#pragma type_attribute` can be used together with a typedef declaration.

Syntax for type attributes on data pointers

The syntax for declaring pointers using type attributes follows the same syntax as the type qualifiers `const` and `volatile`:

```
int __data16 * p;           The int object is located in __data16 memory.
int * __data16 p;         The pointer is located in __data16 memory.
__data16 int * p;        The pointer is located in __data16 memory.
```

Syntax for type attributes on functions

The syntax for using type attributes on functions, differs slightly from the syntax of type attributes on data objects. For functions, the attribute must be placed either in front of the return type, alternatively in parentheses, for example:

```
__interrupt void my_handler(void);
or
void (__interrupt my_handler)(void);
```

The following declaration of `my_handler` is equivalent with the previous one:

```
#pragma type_attribute=__interrupt
void my_handler(void);
```

Syntax for type attributes on function pointers

To declare a function pointer, use the following syntax:

```
int (__code16 * fp) (double);
```

After this declaration, the function pointer `fp` points to code16 memory.

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

```
typedef __code16 void FUNC_TYPE(int);
typedef FUNC_TYPE *FUNC_PTR_TYPE;
FUNC_TYPE func();
FUNC_PTR_TYPE funcptr;
```

Note that `#pragma type_attribute` can be used together with a `typedef` declaration.

OBJECT ATTRIBUTES

Object attributes affect the internal functionality of functions and data objects, but not how the function is called or how the data is accessed. This means that an object attribute does not need to be present in the declaration of an object.

The following object attributes are available:

- Object attributes that can be used for variables: `__no_init`
- Object attributes that can be used for functions and variables: `location`, `@`, and `__root`
- Object attributes that can be used for functions: `__intrinsic`, `__noreturn`, `__raw`, `__task`, and `vector`.

You can specify as many object attributes as required for a specific function or data object.

For more information about `location` and `@`, see *Controlling data and function placement in memory*, page 45. For more information about `vector`, see *#pragma vector*, page 226.

Syntax for object attributes

The object attribute must be placed in front of the type. For example, to place `myarray` in memory that is not initialized at startup:

```
__no_init int myarray[10];
```

The `#pragma object_attribute` directive can also be used. The following declaration is equivalent to the previous one:

```
#pragma object_attribute=__no_init
int myarray[10];
```

Note: Object attributes cannot be used in combination with the `typedef` keyword.

Summary of extended keywords

The following table summarizes the extended keywords:

| Extended keyword | Description |
|----------------------------|-----------------------------------------------------------------------------------|
| <code>__bitvar</code> | Controls the declaration of bit variables |
| <code>__cc_version1</code> | Specifies calling convention 1; available for backward compatibility |
| <code>__cc_version2</code> | Specifies calling convention 2; available for backward compatibility |
| <code>__cc_version3</code> | Specifies the default calling convention explicitly |
| <code>__code16</code> | Controls the storage of functions |
| <code>__code24</code> | Controls the storage of functions |
| <code>__data8</code> | Controls the storage of data objects |
| <code>__data16</code> | Controls the storage of data objects |
| <code>__data32</code> | Controls the storage of data objects |
| <code>__interrupt</code> | Supports interrupt functions |
| <code>__intrinsic</code> | Reserved for compiler internal use only |
| <code>__monitor</code> | Supports atomic execution of a function |
| <code>__no_init</code> | Supports non-volatile memory |
| <code>__noreturn</code> | Informs the compiler that the declared function will not return |
| <code>__raw</code> | Prevents saving used registers in interrupt functions |
| <code>__root</code> | Ensures that a function or variable is included in the object code even if unused |
| <code>__task</code> | Allows functions to exit without restoring registers |
| <code>__trap</code> | Supports trap functions |
| <code>__vector_call</code> | Allows functions to be called direct via the vector area |

Table 38: Extended keywords summary

Note: Some of the keywords can be used on both data and functions.

Descriptions of extended keywords

The following sections give detailed information about each extended keyword.

`__bitvar` Controls the storage of bit variables.

The `__bitvar` keyword is available for declaring single bit variables. A bit variable is placed in the highest 256 bytes of data memory addressable with the `@aa:8` addressing mode. If you want to place a bit variable in a different memory area, you must declare a struct with a single bit bitfield, which has no restrictions in placement. In this case, do not use the `__bitvar` keyword. A variable declared with the `__bitvar` keyword cannot be located.

For backward compatibility, the include file `migration.h` defines the macro `BIT(name)`, which you can use for declaring bit variables.

Note: You cannot take the address of a `__bitvar` variable, nor create a pointer to such a variable.

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 199.

Example

```
__bitvar struct {unsigned char name: 1;}
```

where `name` will be treated as a bit variable and placed in memory in the special BITVARS segment.

`__cc_version1` Specifies calling convention 1.

The `__cc_version1` keyword is available for backward compatibility of the interface for calling assembler routines from C. It makes a function use the calling convention of the old H8 IAR C Compiler up to version 1.20A instead of the default calling convention. This calling convention, alternatively `__cc_version2`, is preferred for calling assembler routines from C.

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 199.

Example

```
__cc_version1 int func(int arg1, double arg2)
```

See also

For information about the different calling conventions, see *Calling convention*, page 89.

`__cc_version2` Specifies calling convention 2.

The `__cc_version2` keyword is available for backward compatibility of the interface for calling assembler routines from C. It makes a function use the calling convention of the old ICCH8 IAR C Compiler from version 1.20A to 1.x instead of the default calling convention. This calling convention, alternatively `__cc_version1`, is preferred for calling assembler routines from C.

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 199.

Example

```
__cc_version2 int func(int arg1, double arg2)
```

See also

For information about the different calling conventions, see *Calling convention*, page 89.

`__cc_version3` Specifies the default calling convention explicitly.

The `__cc_version3` keyword makes a function use the calling convention of the ICCH8 IAR C Compiler version 2.x. This is the default calling convention, which means there is no practical use for this keyword in this version of the compiler.

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 199.

Example

```
__cc_version3 int func(int arg1, double arg2)
```

See also

For information about the different calling conventions, see *Calling convention*, page 89.

`__code16` Controls the storage of functions.

The `__code16` memory attribute places a function in the code16 memory range `0x2-0xFFFF`.

The `__code16` attribute is default in the Small code model and it is only available in the Normal operating mode. This attribute cannot be used in any other situation, which means there is no practical use for it in this version of the compiler:

| Address range | Pointer size | Max module size | Default in code model | Available in operating mode |
|-------------------------|--------------|-----------------|-----------------------|-----------------------------|
| <code>0x2-0xFFFF</code> | 2 bytes | 64 Kbytes | Small | Normal |

Table 39: `__code16` function memory attribute

Syntax

Follows the generic syntax rules for memory type attributes that can be used on functions, see *Type attributes*, page 199.

Example

```
__code16 void foo(void);
__code16 void foo(void)
{
    ...
}
```

See also

For more information about the memory models, see *Code models and memory attributes for function storage*, page 25.

`__code24` Controls the storage of functions.

The `__code24` memory attribute places a function in the code24 memory range `0x2-0xFFFFFFFF`.

The `__code24` attribute is default in the Large code model and it is only available in the Advanced operating mode. This attribute cannot be used in any other situation, which means there is no practical use for it in this version of the compiler:

| Address range | Pointer size | Max module size | Default in code model | Available in operating mode |
|----------------|--------------|-----------------|-----------------------|-----------------------------|
| 0x2-0xFFFFFFFF | 4 bytes | 16 Mbytes | Normal | Advanced |

Table 40: `__code24` function memory attribute

Syntax

Follows the generic syntax rules for memory type attributes that can be used on functions, see *Type attributes*, page 199.

Example

```
__code24 void foo(void);
__code24 void foo(void)
{
  ...
}
```

See also

For more information about the memory models, see *Code models and memory attributes for function storage*, page 25.

`__data8` Controls the storage of data objects.

The `__data8` memory attribute places a data object in the data8 memory type, in other words in the highest 256 bytes of memory, where the start address depends on which processor family, operating mode, and bus width that is being used. This part of the memory is accessible via the addressing mode `@aa:8`.

The following table summarizes the available address ranges for each possible combination of operating mode and processor core:

| Address range | Max object size | Pointer size | Description |
|---------------------|-----------------|--------------|-------------------------------------------------------------------------------------------------------------------------------|
| 0xFF01-0xFFFE | 256-2 bytes | 1 byte | <ul style="list-style-type: none"> Valid for the Normal operating mode Valid for both H8S and H8/300H |
| 0xFFFF01-0xFFFFFE | 256-2 bytes | 1 byte | <ul style="list-style-type: none"> Valid for the Advanced operating mode Valid for H8/300H |
| 0xFFFFF01-0xFFFFFFE | 256-2 bytes | 1 byte | <ul style="list-style-type: none"> Valid for the Advanced operating mode Valid for H8S |

Table 41: `__data8` data memory attribute

Note:

- An access via a `__data8` pointer requires that the pointer is extended to 16 or 32 bits, depending on the operating mode
- The address range also depends on the specified bus width, see *Data pointers*, page 180.

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 199.

Example

```
__data8 int x;
```

See also

For more information about memory types, see *Memory types*, page 15.

`__data16` Controls the storage of data objects.

The `__data16` memory attribute places a data object in the data16 memory range, in other words in the lowest 32 Kbytes and highest 32 Kbytes of memory, where the start address of the highest bytes of memory depends on which processor family, operating mode, and bus width that is being used. This part of the memory is accessible via the addressing mode `@aa:16`. The `__data16` memory attribute is default in the Small data model.

The following table summarizes the available address ranges for each possible combination of operating mode and processor core:

| Address range | Max object size | Pointer size | Description |
|--------------------------------------|--------------------------------------------------------------------|--------------|-----------------------------------------------------------------------------------------------------------------------------------|
| 0x1-0xFFFE | 64 Kbytes -2 | 2 bytes | <ul style="list-style-type: none"> • Valid for the Normal operating mode • Valid for both H8S and H8/300H |
| 0x1-0x7FFE, 0xFF8000- 0xFFFFFE | 32 Kbytes -2 (May not wrap around from 0xFFFFF to 0x0) | 2 bytes | <ul style="list-style-type: none"> • Valid for the Advanced operating mode • Valid for H8/300H |

Table 42: `__data16` data memory attribute

| Address range | Max object size | Pointer size | Description |
|------------------------------------------|-----------------------------------------------------------------------|--------------|----------------------------------------------------------------------------------------------------------------|
| 0x1-0x7FFE, 0xFFFF8000- 0xFFFFFFFF | 32 Kbytes -2 (May not wrap around from 0xFFFFFFFF to 0x0) | 2 bytes | <ul style="list-style-type: none"> Valid for the Advanced operating mode Valid for H8S |

Table 42: `__data16` data memory attribute (Continued)**Note:**

- An access via a `__data16` pointer requires that the pointer is extended to 32 bits, depending on the operating mode
- The address range also depends on the specified bus width, see *Data pointers*, page 180.

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 199.

Example

```
__data16 int x;
```

See also

For more information about memory types, see *Memory types*, page 15.

`__data32` Controls the storage of data objects.

The `__data32` memory attribute places a data object in the data32 memory range, in other words in the entire memory area, where the end address of memory depend on which processor family, operating mode, and bus width that is being used. This memory is accessible via the addressing mode `@aa:32`. The `__data32` memory attribute is default in the Huge data model.

The following table summarizes the available address ranges for each possible combination of operating mode and processor core:

| Address range | Max object size | Pointer size | Description |
|--------------------|-----------------|--------------|--------------------------------------------------------------------------------------------------------------------|
| 0x1- 0xFFFFFE | 16 Mbytes -2 | 4 bytes | <ul style="list-style-type: none"> Valid for the Advanced operating mode Valid for H8/300H |
| 0x1- 0xFFFFFFFF | 4 Gbytes -2 | 4 bytes | <ul style="list-style-type: none"> Valid for the Advanced operating mode Valid for H8S |

Table 43: `__data32` data memory attribute

Note: The address range also depends on the specified bus width, see *Data pointers*, page 180.

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 199.

Example

```
__data32 int x;
```

See also

For more information about memory types, see *Memory types*, page 15.

`__interrupt` The `__interrupt` keyword specifies interrupt functions. To specify one or several interrupt vectors, use the `#pragma vector` directive. The range of the interrupt vectors depends on the device used. It is possible to define an interrupt function without a vector, but then the compiler will not generate an entry in the interrupt vector table.

An interrupt function must have a `void` return type and cannot have any parameters.

The header file `ioderivative.h`, where `derivative` corresponds to the selected derivative, contains predefined names for the existing interrupt vectors.

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 199.

Example

The following example declares an interrupt function with the interrupt vector value `0x14`, which corresponds to offset `0x28` or `0x50` in the `INTVEC` segment, depending on the operating mode:

```
#pragma vector=0x14
__interrupt void my_interrupt_handler(void);
```

See also

For additional information, see *Interrupt functions*, page 26.

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

`__monitor` 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.

Note that it is vital to specify the interrupt mode in use when you compile your monitor function. The compiler will generate code to save the interrupt status on function entry, and restore it before exiting the function. If you specify an incorrect interrupt mode, the generated code will not work correctly in your application. For more information about specifying interrupt mode, see *--interrupt_mode*, page 159.

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, *Type attributes*, page 199.

Example

```
__monitor int get_lock(void);
```

See also

Read more about monitor functions in *Monitor functions*, page 28. Read also about the intrinsic functions `__disable_interrupt`, page 231, `__enable_interrupt`, page 235, `__get_interrupt_state`, page 235, and `__set_interrupt_state`, page 241.

`__no_init` Use the `__no_init` keyword to place a data object in non-volatile memory. This means that the initialization of the variable, for example at system startup, is suppressed.

Syntax

Follows the generic syntax rules for object attributes, *Object attributes*, page 202.

Example

```
__no_init int myarray[10];
```

`__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`.

Syntax

Follows the generic syntax rules for object attributes, *Object attributes*, page 202.

Example

```
__noreturn void terminate(void);
```

__raw Prevents saving used registers in interrupt functions.

Interrupt functions preserve the content of all used processor registers at function entrance and restore them at exit. However, for some very special applications, it can be desirable to prevent the registers from being saved at function entrance. This can be accomplished by the use of extended keyword `__raw`.

Syntax

Follows the generic syntax rules for object attributes, *Object attributes*, page 202.

Example

```
__raw __interrupt void my_interrupt_function()
```

__root A function or variable with the `__root` attribute is kept whether or not it is referenced from the rest of the application, provided its module is included. Program modules are always included and library modules are only included if needed.

Syntax

Follows the generic syntax rules for object attributes, *Object attributes*, page 202.

Example

```
__root int myarray[10];
```

See also

To read more about modules, segments, and the link process, see the *IAR Linker and Library Tools Reference Guide*.

__task This keyword allows functions to exit without restoring registers and it is typically used for the `main` function.

By default, functions save the contents of used non-scratch registers (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`.

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 199.

Example

```
__task void my_handler(void);
```

`__trap` A trap function is called and executed by the `TRAPA` assembler instruction and returned by the `RTE` instruction. A trap function must have one or several vectors, which are specified using the `#pragma vector` directive. See the chip manufacturer's hardware documentation for information about the trap vector range. If a trap vector is not given, an error will be issued if the function is called. A trap function can take parameters and return a value and it has the same calling convention as other functions. You can call the trap functions from your C or C++ application.

Note that you can also make the compiler generate the `TRAPA` instruction by using the intrinsic function `__TRAPA`. In comparison with functions declared with the extended keyword `__trap`, a `__TRAPA` call:

- Will not save any registers
- Cannot pass any parameters to the trap function
- Cannot receive a return value.

The intrinsic function is primarily intended for special debug functions written in assembler.

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 199.

Example

```
#pragma vector=2
__trap int my_trap_function(void);
```

See also

For additional information, see `__TRAPA`, page 242, *Calling convention*, page 89 and *Trap functions*, page 27.

`__vector_call` Allows functions to be called via the vector area.

By default, any function you write will be called as a normal subroutine via a direct function call. However, a `__vector_call` declared function will instead be called indirectly via the vector area, which means the addressing mode `@@aa:8` is used.

A call using the `@@aa:8` addressing mode is only 2 bytes in size, compared to 4 bytes for a normal function call using the `@aa:24` addressing mode.

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 199.

See also

For more details, see *Function vectors for non-interrupt functions*, page 44

Pragma directives

This chapter describes the pragma directives of the H8 IAR C/C++ Compiler.

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

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

The pragma directives are always enabled in the compiler.

Summary of pragma directives

The following table shows the pragma directives of the compiler:

| Pragma directive | Description |
|--------------------------------------------------------------|---------------------------------------------------|
| <code>#pragma</code> <code>basic_template_matching</code> | Makes a template function fully memory-aware |
| <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 44: 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 or trap function |

Table 44: 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 *H8 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 110.

Example

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

fun((int __data8 *) 0); // T = int __data8
```

```
#pragma bitfields #pragma bitfields={reversed|default}
```

The `#pragma bitfields` directive controls the order of bitfield members.

By default, the H8 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=__data16 MyOtherSeg
```

All constants defined following this directive will be placed in the segment `MyOtherSeg` and accessed using `data16` addressing. Note that data objects declared without the `const` keyword are not affected by this pragma directive.

```
#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=__data16 MyOtherSeg
```

All variables in `MyOtherSeg` will be accessed using `data16` 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 *Diagnostics*, page 139 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 *Diagnostics*, page 139 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 *Diagnostics*, page 139 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 *Diagnostics*, page 139 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 *Diagnostics*, page 139 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 border="0"> <tr> <td style="padding-right: 20px;"><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><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=0xFF2000 char PORT1; /* PORT1 is located at address 0xFF2000 */</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 43.</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 border="0"> <tr> <td style="padding-right: 20px;"><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 function:</p> <table border="0"> <tr> <td style="padding-right: 20px;"><code>__noreturn</code></td> <td>Informs the compiler that the function will not return.</td> </tr> <tr> <td><code>__raw</code></td> <td>Prevents saving used registers in interrupt functions.</td> </tr> </table> | <code>__no_init</code> | Suppresses initialization of a variable at startup. | <code>__noreturn</code> | Informs the compiler that the function will not return. | <code>__raw</code> | Prevents saving used registers in interrupt functions. |
| <code>__no_init</code> | Suppresses initialization of a variable at startup. | | | | | | |
| <code>__noreturn</code> | Informs the compiler that the function will not return. | | | | | | |
| <code>__raw</code> | Prevents saving used registers in interrupt functions. | | | | | | |

`__task` Allows functions to exit without restoring registers.

The following keyword can be used with `#pragma object_attribute` for a function or variable:

`__root` Ensures that a function or data object is included in the linked application, even if it is not referenced.

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 |
| <code>no_unroll</code> | Turns off loop unrolling |

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.

`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.

Note that accessing an object that is not aligned at its correct alignment requires code that is both larger and slower than the code needed to access the same kind of object when aligned correctly. If there are many accesses to such fields in the program, it is usually better to construct the correct values in a struct that is not packed, and access this instead.

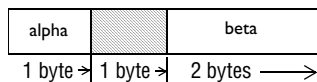
Also, special care is needed when creating and using pointers to misaligned fields. For direct access to such fields in a packed struct, the compiler will emit the correct (slower and larger) code when needed. However, when such a field is accessed through a pointer to the field, the normal (smaller and faster) code for accessing the type of the field is used, which will, in the general case, not work.

Example 1

This example declares a structure without using the `#pragma pack` directive:

```
struct First
{
    char alpha;
    short beta;
};
```

In this example, the structure `First` is not packed and has the following memory layout:



Note that one pad byte has been added.

Example 2

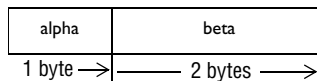
This example declares a similar structure using the `#pragma pack` directive:

```
#pragma pack(1)

struct FirstPacked
{
    char alpha;
    short beta;
};

#pragma pack()
```

In this example, the structure `FirstPacked` is packed and has the following memory layout:

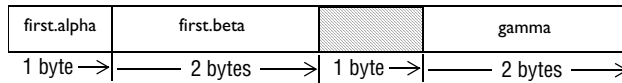


Example 3

This example declares a new structure, `Second`, that contains the structure `FirstPacked` declared in the previous example. The declaration of `Second` is not placed inside a `#pragma pack` block:

```
struct Second
{
    struct FirstPacked first;
    short gamma;
};
```

The following memory layout is used:



Note that the structure `FirstPacked` will use the memory layout, size, and alignment described in example 2. The alignment of the member `gamma` is 2, which means that alignment of the structure `Second` will become 2 and one pad byte will be added.

```
#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 and examples of using 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 operators `__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 operator. 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" __data16 4
```

See also *Dedicated segment operators* `__segment_begin` and `__segment_end`, page 189.

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 qualifiers*, page 183.

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 `__data8` is defined as `MyCharInData8`. Then a pointer is declared; the pointer is located in `data16` memory and it points to a `char` object that is located in `data8` memory.

```
#pragma type_attribute=__data8
typedef char MyCharInData8;
#pragma type_attribute=__data16
MyCharInData8 * ptr;
```

The following declarations, which use extended keywords, are equivalent. See the chapter *Extended keywords* for more details.

```
char __data8 * __data16 ptr;
```

```
#pragma vector #pragma vector=vector1[, vector2, vector3, ...]
```

The `#pragma vector` directive specifies the vector(s) of an interrupt or trap function whose declaration follows the pragma directive.

Example

```
#pragma vector=0x14
__interrupt void my_handler(void);
```

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.

Summary of intrinsic functions

The following table summarizes the intrinsic functions:

| Intrinsic function | Description |
|-----------------------------------|----------------------------------------------------------------------------------------------------|
| <code>__and_ccr</code> | Performs a logical AND between the contents of CCR and a parameter; the result is stored in CCR |
| <code>__and_exr</code> | Performs a logical AND between the contents of EXR and a parameter; the result is stored in EXR |
| <code>__bcd_add_char</code> | Returns the sum of two parameters represented as a two-digit BCD number |
| <code>__bcd_add_short</code> | Returns the sum of two parameters represented as a four-digit BCD number |
| <code>__bcd_subtract_char</code> | Returns the difference between two parameters represented as a two-digit BCD number |
| <code>__bcd_subtract_short</code> | Returns the difference between two parameters represented as a four-digit BCD number |
| <code>__dadd</code> | Adds the BCD array parameters and stores the result in memory |
| <code>__disable_interrupt</code> | Disables interrupts |
| <code>__do_byte_eepmov</code> | Inserts an <code>EEPMOV.B</code> instruction |
| <code>__do_word_eepmov</code> | Inserts an <code>EEPMOV.W</code> instruction |
| <code>__dsub</code> | Subtracts two BCD array parameters and stores the result in memory |
| <code>__eepmov</code> | Inserts an <code>EEPMOV.B</code> instruction or a loop using the <code>EEPMOV.W</code> instruction |

Table 45: Intrinsic functions summary

| Intrinsic function | Description |
|------------------------------------|-------------------------------------------------------------------------------------------------------------|
| <code>__eepmovi</code> | Inserts one or two <code>EEPMOV.B</code> instructions or a loop using the <code>EEPMOV.W</code> instruction |
| <code>__enable_interrupt</code> | Enables interrupts |
| <code>__get_imask_ccr</code> | Returns the value of the CCR I bit |
| <code>__get_imask_exr</code> | Returns the value of the EXR I2-I0 bits |
| <code>__get_interrupt_state</code> | Returns the interrupt state |
| <code>__mac</code> | Performs multiply and accumulate |
| <code>__mac1</code> | Performs multiply and accumulate with a mask |
| <code>__MOVFPPE</code> | Reads data using the <code>MOVFPPE</code> instruction |
| <code>__MOVTPPE</code> | Writes data using the <code>MOVTPPE</code> instruction |
| <code>__no_operation</code> | Generates a <code>NOP</code> instruction |
| <code>__or_ccr</code> | Performs a logical OR between the contents of CCR and parameter; the result is stored in CCR |
| <code>__or_exr</code> | Performs a logical OR between the contents of EXR and parameter; the result is stored in EXR |
| <code>__read_ccr</code> | Returns the value of the CCR register |
| <code>__read_exr</code> | Returns the value of the EXR register |
| <code>__rotlc</code> | Rotates a value to the left |
| <code>__rotlw</code> | Rotates a value to the left |
| <code>__rotll</code> | Rotates a value to the left |
| <code>__rotrc</code> | Rotates a value to the right |
| <code>__rotrw</code> | Rotates a value to the right |
| <code>__rotrl</code> | Rotates a value to the right |
| <code>__set_imask_ccr</code> | Sets the I bit in the CCR register |
| <code>__set_imask_exr</code> | Sets the I2-I0 bits in the EXR register |
| <code>__set_interrupt_mask</code> | Sets the interrupt control bits in the CCR register |
| <code>__set_interrupt_state</code> | Restores the interrupt state |
| <code>__sleep</code> | Enters sleep mode; inserts a <code>SLEEP</code> instruction |
| <code>__TAS</code> | Inserts a <code>TAS</code> instruction |
| <code>__TRAPA</code> | Inserts a <code>TRAPA</code> instruction |
| <code>__write_ccr</code> | Sets CCR to a specific value |
| <code>__write_exr</code> | Sets EXR to a specific value |

Table 45: Intrinsic functions summary (Continued)

| Intrinsic function | Description |
|------------------------|-------------------------------------------------------------------------------------------------|
| <code>__xor_ccr</code> | Performs a logical XOR between the contents of CCR and a parameter; the result is stored in CCR |
| <code>__xor_exr</code> | Performs a logical XOR between the contents of EXR and a parameter; the result is stored in EXR |

Table 45: 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
```

Also note that some of the intrinsic functions are only available under certain circumstances. Specifically, intrinsic functions operating on the EXR register are only available for the H8S core. Intrinsic functions operating on the MAC register are only available for devices with a MAC register.

Descriptions of intrinsic functions

The following section gives reference information about each intrinsic function.

```
__and_ccr void __and_ccr(unsigned char);
```

Performs a logical AND between the contents of the condition code register CCR and the parameter; the result is stored in CCR. The parameter must be a constant value.

Example

```
__and_ccr(0xEF);    /* Clear the U bit */
```

```
__and_exr void __and_exr(unsigned char);
```

Performs a logical AND between the contents of the extended control register EXR and the parameter; the result is stored in EXR. The parameter must be a constant value.

Note: This intrinsic function is only available for the H8S core.

Example

```
__and_exr(0x7F);    /* Clear the T bit */
```

```
__bcd_add_char unsigned char __bcd_add_char(unsigned char x, unsigned char y);
```

Returns the sum of the two parameters. The parameters and the return value are represented as two-digit binary coded decimal (BCD) numbers.

Example

```
c = __bcd_add_char(c, 0x09); /* Add decimal 9 to c */
```

```
__bcd_add_short unsigned short __bcd_add_short(unsigned short x,
                                               unsigned short y);
```

Returns the sum of the two parameters. The return value is represented as a four-digit binary coded decimal (BCD) number.

Example

```
s = __bcd_add_short(s, 0x0199); /* Add decimal 199 to s */
```

```
__bcd_subtract_char unsigned char __bcd_subtract_char(unsigned char x,
                                                       unsigned char y);
```

Returns the difference between the two parameters, where the operations is $x - y$. The parameters and the return value are represented as two-digit binary coded decimal (BCD) numbers.

Example

```
c = __bcd_subtract_char(c, 0x09); /* Subtract decimal 9
                                   from c */
```

```
__bcd_subtract_short unsigned short __bcd_subtract_short(unsigned short x,
                                                         unsigned short y);
```

Returns the difference between the two parameters, where the operation is $x - y$. The parameters and the return value are represented as four-digit binary coded decimal (BCD) numbers.

Example

```
s = __bcd_subtract_short(s, 0x0199); /* Subtract decimal 199
                                       from s */
```

```
__dadd void __dadd(unsigned char size, const void * ptr1,
                  const void * ptr2, void * res);
```

Adds two large binary numbers indicated by the parameters *ptr1* and *ptr2*. The result is stored in memory indicated by the parameter *res*.

The numbers are stored as character arrays, where each character contains two binary coded decimal (BCD) digits. The *size* parameter denotes the number of characters in the arrays (the array size), implying that the number of BCD digits is $size * 2$.

The BCD numbers are stored with the most significant byte at the lowest address (in the first array entry).

Example

```
char bcd_num_1[10];
char bcd_num_2[10];
char bcd_result[10];

__dadd(10,                                /* Perform bcd_num_1 + bcd_num_2 */
      bcd_num_1,
      bcd_num_2,
      bcd_result);
```

```
__disable_interrupt void __disable_interrupt(void);
```

The code generated to disable interrupts depends on the interrupt mode you have specified with the command line option `--interrupt_mode`.

Note that it is vital that you specify the interrupt mode you are using, when you use this intrinsic function. If you specify an incorrect interrupt mode, the generated code will not work correctly in your application. For more information about specifying the interrupt mode, see `--interrupt_mode`, page 159.

Example

```
__disable_interrupt(); /* Disable interrupts */
```

```
__do_byte_eepmov void __do_byte_eepmov(const void * src, void * dst,
                                       unsigned char count);
```

Inserts one `EPMOV.B` instruction to transfer up to 255 bytes of data from the memory indicated by *src* to the memory indicated by *dst*. Note that interrupts are disabled during the transfer, including the NMI interrupt.

In contrast to `EEPMOV.B` instructions generated automatically by the compiler, this intrinsic function does not consider the value specified by the command line option `--max_cycles_no_interrupt`. Even if the option specifies that a maximum of 100 cycles may pass with interrupts disabled, you can specify a transfer count, when using this intrinsic function, that causes interrupts to be disabled for a longer period.

Note that the order of the parameters *src* and *dst* differs between `__do_byte/word_eepmov` and `__eepmov/eepmovi`.

Example

```
__do_byte_eepmov(buf1, buf2, 100); /* Move 100 bytes from buf1
                                   to buf2 */
```

```
__do_word_eepmov void __do_word_eepmov(const void * src, void * dst,
                                       unsigned short count);
```

Inserts a program loop using an `EEPMOV.W` instruction to transfer up to 65535 byte of data from the memory indicated by *src* to the memory indicated by *dst*. Note that in contrast to `__do_byte_eepmov`, the NMI interrupt is enabled during the transfer.

In contrast to `EEPMOV.W` instructions generated automatically by the compiler, the intrinsic function does not consider the value specified by the command line option `--max_cycles_no_interrupt`. Even if the option specifies that a maximum of 100 cycles may pass with interrupts disabled, you can specify a transfer count, when using this intrinsic function, that causes interrupts to be disabled for a longer period.

Note that the order of the parameters *src* and *dst* differs between `__do_byte/word_eepmov` and `__eepmov/eepmovi`.

Example

```
__do_word_eepmov(buf1, buf2, 10000); /* Move 10000 bytes from
                                       buf1 to buf2 */
```

```
__dsub void __dsub(unsigned char size, const void * ptr1,
                  const void * ptr2, void * res);
```

Subtracts two large numbers indicated by the parameters *ptr1* and *ptr2*, where the operation is $ptr1 - ptr2$. The result is stored in memory indicated by the parameter *res*.

The numbers are stored as character arrays, where each character contains two binary coded decimal (BCD) digits. The *size* parameter denotes the numbers of characters in the arrays (the array size), implying that the number of BCD digits is $size * 2$.

The BCD numbers are stored with the most significant byte at the lowest address (in the first array entry).

Example

```
char bcd_num_1[10];
char bcd_num_2[10];
char bcd_result[10];

__dsub(10,                /* Perform bcd_num_1 - bcd_num_2 */
      bcd_num_1,
      bcd_num_2,
      bcd_result);
```

```
__eepmov void __eepmov(void * dst, const void * src,
                     unsigned short count);
```

Inserts code to transfer data from the memory indicated by *src* to the memory indicated by *dst*. The parameter *count* must be a constant value in the range 0–65535. If the value is 255 or less, the compiler inserts one `EEPMOV.B` instruction to transfer data. If the value is above 255, the compiler inserts a loop using an `EEPMOV.W` instruction to transfer data.

Note that interrupts are disabled during the transfer. If *count* is larger than 255, which means an `EEPMOV.W` instruction is used, NMI interrupts are accepted during the transfer.

In contrast to `EEPMOV.B` and `EEPMOV.W` instructions generated automatically by the compiler, this intrinsic function does not consider the value specified by the command line option `--max_cycles_no_interrupt`. Even if the option specifies that a maximum of 100 cycles may pass with interrupts disabled, you can specify a transfer count, when using this intrinsic function, that causes interrupts to be disabled for a longer period.

Note that the order of the parameters *src* and *dst* differs between

`__do_byte/word_eepmov` and `__eepmov/eepmovi`.

Example

```
__eepmov(buf2, buf1, 100);    /* Move 100 bytes from buf1 to
                             buf2 using EEPMOV.B */

__eepmov(buf2, buf1, 1000);  /* Move 1000 bytes from buf1 to
                             buf2 using EEPMOV.W */
```

```
__eepmovi void __eepmovi(void * dst, const void * src,
                        unsigned short count);
```

Inserts code to transfer data from the memory indicated by *src* to the memory indicated by *dst*. The argument *count* may be a constant value in the range 0–65535. If the value is 255 or less, the compiler will insert one `EEPMOV.B` instruction to transfer data. If the value is above 255, but 510 or less, the compiler will insert two `EEPMOV.B` instructions to transfer data. If the value is above 510, the compiler will insert a loop using an `EEPMOV.W` instruction to transfer data.

If the parameter *count* is a variable, the compiler will insert a loop using an `EEPMOV.W` instruction to transfer data.

Note that interrupts are disabled during the transfer. If the parameter *count* is a variable or constant larger than 510, which means an `EEPMOV.W` instruction is used, NMI interrupts will be accepted during the transfer. For constant values in the range 256 to 510, interrupts are accepted between the two `EEPMOV.B` instructions.

In contrast to `EEPMOV.B` and `EEPMOV.W` instructions generated automatically by the compiler, this intrinsic function does not consider the value specified by the command line option `--max_cycles_no_interrupt`. Even if the option specifies that a maximum of 100 cycles may pass with interrupts disabled, you can specify a transfer count, when using this intrinsic function, that causes interrupts to be disabled for a longer period.

Note that the order of the parameters *src* and *dst* differs between `--do_byte/word_eepmov` and `--eepmov/eepmovi`.

Example

```
__eepmovi(buf2, buf1, 100); /* Move 100 bytes from buf1 to
                          buf2 using EEPMOV.B */

__eepmovi(buf2, buf1, 400); /* Move 400 bytes from buf1 to
                          buf2 using two EEPMOV.B */

__eepmovi(buf2, buf1, 1000); /* Move 1000 bytes from buf1 to
                          buf2 using EEPMOV.W */

__eepmovi(buf2, buf1, x); /* Move x bytes from buf1 to
                          buf2 using EEPMOV.W */
```

```
__enable_interrupt void __enable_interrupt(void);
```

The code generated to enable interrupts depends on the interrupt mode you have specified with the command line option `--interrupt_mode`.

When you use this intrinsic function, it is vital that you specify the interrupt mode in use. If you specify an incorrect interrupt mode, the generated code will not work correctly in your application. For more information about specifying the interrupt mode, see *--interrupt_mode*, page 159.

Example

```
__enable_interrupt(); /* Enable interrupts */
```

```
__get_imask_ccr unsigned char __get_imask_ccr(void);
```

Returns the value (0 or 1) of the `I` bit in the condition code register `CCR`.

Example

```
c = __get_imask_ccr(); /* Read I bit value (0 or 1) */
```

```
__get_imask_exr unsigned char __get_imask_exr(void);
```

Returns the values (0 to 7) of the `I2-10` bits in the extended control register `EXR`.

Note: This intrinsic function is only available for H8S.

Example

```
c = __get_imask_exr(); /* Read I2-I0 bits value (0 to 7) */
```

```
__get_interrupt_state __istate_t __get_interrupt_state(void);
```

Returns the global interrupt state. The return type `__istate_t` has the following definition:

```
typedef unsigned short __istate_t;
```

The value of the `CCR` register is saved in the low byte of the short integer. If you compile for the H8S core, the `EXR` register is saved in the high byte.

The return value can be used as a parameter for the `__set_interrupt_state` function, which will restore the interrupt state.

Example

```
state = __get_interrupt_state(); /* Save interrupt state */
```

```
__mac long __mac(long val, const short * ptr1, const short * ptr2,
                unsigned long count);
```

Performs a series of multiply and accumulate instructions and returns the sum.

The `MACL` register is initialized with `val`, and the `MACH` register is cleared. Then, the two signed short integer vectors pointed to by `ptr1` and `ptr2` are multiplied and accumulated. Each pair of entries (one from vector `ptr1` and one from `ptr2`) is multiplied with signed multiplication, and the 32-bit result from each multiplication is added to the contents of the `MACL` and `MACH` registers. This procedure is repeated `count` times, in other words, `count` holds the number of values in the vectors. The contents of the `MACL` register after the operation is returned as the function value.

For more details about the `MAC` instruction, such as how to specify saturated or non-saturated mode, see the *H8S Programming Manual*.

For more advanced operations, such as using the resulting value of the `MACH` register, or using the value of the multiplier flags `N-MULT`, `Z-MULT`, or `V-MULT`, you should write your own assembler routines. Note that the H8 IAR C/C++ Compiler supports the data type `long long` (64 bits), which can be useful if you write a function that returns the combined value from `MACH` and `MACL`.

Note: This intrinsic function is only available for H8S devices with MAC hardware.

Example

```
short vec1[16];
short vec2[16];
```

```
result = __mac(0, vec1, vec2, 16); /* Multiply/accumulate vec1
                                   and vec2 */
```

```
__mac1 long __mac1(long val, const short * ptr1, const short * ptr2,
                  unsigned long count, unsigned long mask);
```

Performs a series of multiply and accumulate instructions, and returns the sum.

For a general description of the operation of this intrinsic function, see `__mac`, page 236. In addition, if a suitable value is set for `mask`, the `__mac1` intrinsic function also handles the vector pointed to by `ptr2` as a ring buffer. In other words, that vector can be shorter than the vector pointed to by `ptr1`. The first entries of the `ptr2` vector are repeatedly reused.

To work correctly, you must carefully consider the mask value. It is recommended that you set the value to the inverted value of the byte size of the vector pointed to by `ptr2`, for example `~4` if the size of the vector is 4 bytes (two `short` integer entries).

You must also make sure that the vector pointed to by `ptr2` is properly aligned. The alignment must be twice as high as the size of the ring buffer, for example 8 if the size of the vector pointed to by `ptr2` is 4.

Example

The following example and the corresponding figure illustrates the operation of the intrinsic function:

```
short vec1[8];
#pragma data_alignment=3    /* Align on 8-byte boundary */
short vec2[2];             /* Size is 4 bytes */

result = __mac1(0, vec1, vec2, 8, ~4);
        /* Mask is the inverted value of the size of vec2 */:
```

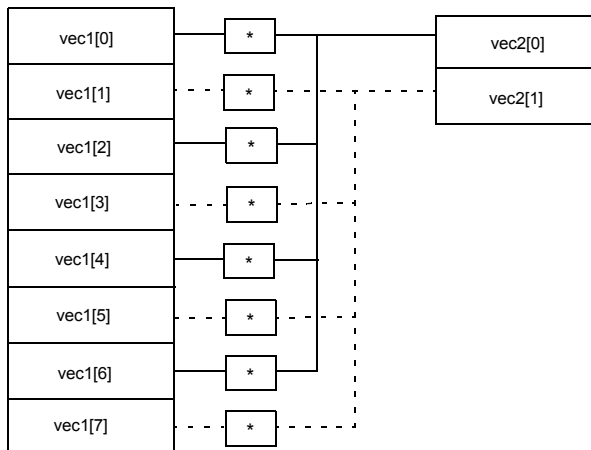


Figure 3: Graphical view of `__mac1` intrinsic function

Note: This intrinsic function is only available for H8S devices with MAC hardware.

```
__MOVFPPE unsigned char __MOVFPPE(const unsigned char __data16 *);
```

Reads data using the MOVFPPE instruction. To meet the requirements of the MOVFPPE instruction, the address must be an absolute data16 address.

Example

```
c = __MOVFPPE(&port); /* Read data from port using MOVFPPE */
```

```
__MOVTPPE void __MOVTPPE(unsigned char, unsigned char __data16 *);
```

Writes data using the MOVTPPE instruction. To meet the requirements of the MOVTPPE instruction, the address must be an absolute data16 address.

Example

```
__MOVTPPE(c, &port); /* Write data to port using MOVTPPE */
```

```
__no_operation void __no_operation(void);
```

Generates a NOP instruction.

Example

```
__no_operation(); /* Insert a NOP instruction */
```

```
__or_ccr void __or_ccr(unsigned char);
```

Performs a logical OR between the contents of the condition code register CCR and the parameter; the result is stored in CCR. The parameter must be a constant value.

Example

```
__or_ccr(0x10); /* Set the U bit */
```

```
__or_exr void __or_exr(unsigned char);
```

Performs a logical OR between the contents of the extended control register EXR and the parameter; the result is stored in EXR. The parameter must be a constant value.

Note: This intrinsic function is only available for H8S.

Example

```
__or_exr(0x80); /* Set the T bit */
```

```
__read_ccr unsigned char __read_ccr(void);
```

Returns the value of the condition code register CCR.

Example

```
c = __read_ccr(); /* Read contents of CCR */
```

```
__read_exr unsigned char __read_exr(void);
```

Returns the value of the extended control register EXR.

Note: This intrinsic function is only available for H8S.

Example

```
c = __read_exr(); /* Read contents of EXR */
```

```
__rotlc unsigned char __rotlc(unsigned short count, unsigned char value)
```

Returns *value* after the argument has been rotated left *count* number of times.

```
__rotlw unsigned short __rotlw(unsigned short count, unsigned short
                               value)
```

Returns *value* after the argument has been rotated left *count* number of times.

```
__rotll unsigned long __rotll(unsigned short count, unsigned long value)
```

Returns *value* after the argument has been rotated left *count* number of times.

```
__rotrc unsigned char __rotrc(unsigned short count, unsigned char value)
```

Returns *value* after the argument has been rotated right *count* number of times.

```
__rotrw unsigned short __rotrw(unsigned short count, unsigned short
                               value)
```

Returns *value* after the argument has been rotated right *count* number of times.

```
__rotr1 unsigned long __rotr1(unsigned short count, unsigned long value)
```

Returns *value* after the argument has been rotated right *count* number of times.

```
__set_imask_ccr void __set_imask_ccr(unsigned char);
```

Sets or clears the interrupt mask bit *I* in the condition code register *CCR*. The function takes a parameter, which must be a constant value representing the bit mask, where:

| | |
|---|-------------------------|
| 0 | Clears the <i>I</i> bit |
| 1 | Sets the <i>I</i> bit |

Example

```
__set_imask_ccr(0); /* Clear I bit in CCR */
```

```
__set_imask_exr void __set_imask_exr(unsigned char);
```

Sets or clears the interrupt mask bits *I2* to *I0* in the extended control register *EXR*. The function takes a parameter, which must be a constant value representing the bit mask.

The bits *I2* to *I0* in the *EXR* register are treated like a 3-bit bitfield, that can take the value 0 to 7. For example, if you specify the parameter value 3, binary 011, bit *I2* will be cleared, and *I1* and *I0* will be set.

Note: This intrinsic function is only available for H8S.

Example

```
__set_imask_exr(7); /* Set highest interrupt level in I2-I0 in
                    EXR */
```

```
__set_interrupt_mask void __set_interrupt_mask(unsigned char);
```

Sets or clears the interrupt control bits *I* and *UI* in the condition code register *CCR*. The function takes a parameter, which must be a constant value representing the bit mask, where:

| | |
|---|----------------------------------------------------|
| 0 | Clears the <i>I</i> and <i>UI</i> bits |
| 1 | Clears the <i>I</i> bit and sets the <i>UI</i> bit |
| 2 | Sets the <i>I</i> bit and clears the <i>UI</i> bit |

3 Sets the I and UI bits

Example

```
__set_interrupt_mask(3);    /* Set both I and UI bits in CCR */
```

```
__set_interrupt_state void __set_interrupt_state(__istate_t);
```

Restores the interrupt state by setting the value returned by the `__get_interrupt_state` function.

The contents of the CCR register is always restored. If you compile for the H8S core, the EXR register is also restored.

For information about the `__istate_t` type, see `__get_interrupt_state`, page 235.

Example

```
__set_interrupt_state(state);    /* Restore interrupt state */
```

```
__sleep void __sleep(void);
```

Inserts a SLEEP instruction.

Example

```
__sleep();    /* Enter power-down mode */
```

```
__TAS bool __TAS(unsigned char * addr);
```

Inserts the test-and-set TAS instruction.

The function reads the old value of the variable pointed to by `addr`, and in the same instruction also sets bit 7 of that variable. The old value of bit 7 in the variable is returned; true if bit 7 is set and false if bit 7 is cleared.

This intrinsic function is primarily intended for semaphore handling. If you use a normal instruction sequence (read old value, set new value), an interrupt routine might execute between the read of the old value and the assignment of a new value. If the interrupt modifies the value, the test of the old value will be incorrect at the interrupt function exit. To make the test and set an atomic operation, use the `__TAS` intrinsic function.

Note: This intrinsic function is only available for H8S.

Example

```
while (!__TAS(&semaphore)); /* Test and set semaphore */
    delay();                /* If not free, wait and retry */
```

```
__TRAPA void __TRAPA(unsigned char);
```

Inserts a TRAPA instruction. The parameter specifies the trap number, which can have the values 0–3.

Note the difference between declaring a function with the extended keyword `__trap`, and using this intrinsic function.

When you declare a function with the keyword `__trap`, the function can be called in the same way as any other function, including providing parameters and using a return value. The only difference to a normal function is that the TRAPA instruction is used for calling the function, and that the function executes while interrupts are disabled.

However, if you use the `__TRAPA` intrinsic function, the compiler does not perform the normal procedures at function calls:

- It does not generate code to save non-scratch registers
- It does not accept parameters to the trap function
- It does not handle a return value from the trap function
- It assumes that all registers preserve their values, except for the condition code register CCR.

This means that you cannot write the trap function in C, if you intend to call it with the `__TRAPA` intrinsic function. In that case, you have to write it in assembler and make sure that the trap function does not destroy any registers except for the CCR register.

The advantage using the `__TRAPA` intrinsic function is that there is no extra overhead involved; the only extra code generated by the compiler is the TRAPA instruction.

Example

```
__TRAPA(2); /* Call trap function with vector 2 */
```

```
__write_ccr void __write_ccr(unsigned char);
```

Sets the condition code register CCR to a specific value.

Example

```
__write_ccr(0); /* Clear entire CCR */
```

```
__write_exr void __write_exr(unsigned char);
```

Sets the extended control register `EXR` to a specific value.

Note: This intrinsic function is only available for H8S.

Example

```
__write_exr(0); /* Clear entire EXR (T and I2-I0) */
```

```
__xor_ccr void __xor_ccr(unsigned char);
```

Performs a logical `XOR` between the contents of the condition code register `CCR` and the parameter; the result is stored in `CCR`. The parameter must be a constant value.

Example

```
__xor_ccr(0x10); /* Toggle value of the U bit */
```

```
__xor_exr void __xor_exr(unsigned char);
```

Performs a logical `XOR` between the contents of the extended control register `EXR` and the parameter; the result is stored in `EXR`. The parameter must be a constant value.

Note: This intrinsic function is only available for H8S.

Example

```
__xor_exr(0x80); /* Toggle value of the T bit */
```


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 H8 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 preprocessor symbols*, page 246.

- User-defined preprocessor symbols

In addition to defining your own preprocessor symbols using the `#define` directive, you can also use the option `-D` to define your own preprocessor symbols, see *-D*, page 150.

- 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 related to the preprocessor, see *Description of miscellaneous preprocessor extensions*, page 252.

- Preprocessor output

Use the option `--preprocess` to direct preprocessor output to a named file, see *--preprocess*, page 168.

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 288.

Predefined preprocessor 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>__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>__BUILD_NUMBER__</code> | A unique integer that identifies the build number of the compiler currently in use |
| <code>__CODE_COMPATIBILITY_CHECK__</code> | An integer that identifies whether strict runtime model attribute control is made |
| <code>__CODE_MODEL__</code> | An integer that identifies the code model in use |
| <code>__CORE__</code> | An integer that identifies the chip core in use |
| <code>__cplusplus</code> | Determines whether the compiler runs in C++ mode* |
| <code>__DATA_MODEL__</code> | An integer that identifies the data model in use |
| <code>__DATE__</code> | Determines the date of compilation* |
| <code>__DIRECT_LIBRARY_CALLS__</code> | An integer that identifies whether direct library calls are made instead of vector calls |
| <code>__DOUBLE_SIZE__</code> | An integer that identifies which size is used for double |
| <code>__embedded_cplusplus</code> | Determines whether the compiler runs in C++ mode* |
| <code>__FILE__</code> | Identifies the name of the file being compiled* |
| <code>__IAR_SYSTEMS_ICC__</code> | Identifies the IAR compiler platform |
| <code>__ICCH8__</code> | Identifies the H8 IAR C/C++ Compiler |
| <code>__INTERRUPT_MODE__</code> | An integer that identifies the interrupt mode in use |
| <code>__LINE__</code> | Determines the current source line number* |
| <code>__LITTLE_ENDIAN__</code> | An integer that identifies the byte order in use |
| <code>__MAC_ENABLED__</code> | Identifies whether the MAC register is enabled |
| <code>__MAX_CYCLES_NO_INTERRUPT__</code> | Identifies the maximum number of cycles that interrupts may be disabled |

Table 46: Predefined symbols summary

| Predefined symbol | Identifies |
|-------------------------------------|------------------------------------------------------------------------------------------------------------------------|
| <code>__MODEL_XXX__</code> | A constant value to identify the code and data model in use |
| <code>__OPERATING_MODE__</code> | An integer that identifies the operating mode in use |
| <code>__STACK_POINTER_SIZE__</code> | An integer that identifies the maximum stack size that can be used |
| <code>__STDC__</code> | Identifies ISO/ANSI Standard C* |
| <code>__STDC_VERSION__</code> | Identifies the version of ISO/ANSI Standard C in use* |
| <code>__SUBVERSION__</code> | An integer that identifies the version letter of the version number, for example the C in 4.21C, as an ASCII character |
| <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 |

Table 46: Predefined symbols summary

* This symbol is required by the ISO/ANSI standard.

DESCRIPTIONS OF PREDEFINED SYMBOLS

The following section gives reference information about each predefined symbol.

| | |
|-------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>__BASE_FILE__</code> | 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. See also, <code>__FILE__</code> , page 249. |
| <code>__BUILD_NUMBER__</code> | A unique integer that identifies the build number of the compiler currently in use. The build number does not necessarily increase with a compiler that is released later. |
| <code>__CODE_COMPATIBILITY_CHECK__</code> | An integer that identifies whether strict runtime model attribute control is made. The value of this symbol is defined to 1 when the <code>--weak_rtmodel_check</code> option is not used, which means that specific values are generated for all runtime model attributes. |

| | |
|---------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>__CODE_MODEL__</code> | <p>An integer that identifies the code model in use.</p> <p>This symbol reflects the setting of the <code>--code_model</code> option. The value of the symbol is defined to <code>__MODEL_SMALL__</code> (1) or <code>__MODEL_LARGE__</code> (2) for the Small and Large code models, respectively. These symbolic names can be used when testing the <code>__CODE_MODEL__</code> symbol.</p> |
| <code>__CORE__</code> | <p>An integer that identifies the chip core being used.</p> <p>This symbol reflects the setting of the <code>--core</code> option. The value of the symbol is defined to <code>__CORE_H8300H__</code> (1) or <code>__CORE_H8S__</code> (2) for the H8300H and H8S cores, respectively. These symbolic names can be used when testing the <code>__CORE__</code> symbol.</p> |
| <code>__cplusplus</code> | <p>This predefined symbol expands to the number 199711L 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>__DATA_MODEL__</code> | <p>An integer that identifies the data model being used.</p> <p>This symbol reflects the setting of the <code>--data_model</code> option. The value of the symbol is defined to <code>__MODEL_SMALL__</code> (1) and <code>__MODEL_HUGE__</code> (3) for the Small and Huge data models, respectively. These symbolic names can be used when testing the <code>__DATA_MODEL__</code> symbol.</p> |
| <code>__DATE__</code> | <p>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".</p> |
| <code>__DIRECT_LIBRARY_CALLS__</code> | <p>An integer that identifies whether direct library calls are made instead of vector calls.</p> <p>The value of this symbol is defined to 1 when the <code>--direct_library_calls</code> option is used, which means that library functions are called directly instead of indirectly via the vector area.</p> |

| | |
|-----------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>__DOUBLE_SIZE__</code> | <p>An integer that identifies the size of <code>double</code> being used.</p> <p>This symbol reflects the setting of the <code>--double</code> option. The value of the symbol is defined to 4 when the size of <code>double</code> is 32, and 8 when the size of <code>double</code> is 64.</p> |
| <code>__embedded_cplusplus</code> | <p>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.</p> |
| <code>__FILE__</code> | <p>Use this symbol to identify which file is currently being compiled. This symbol expands to the name of that file.</p> <p>See also, <code>__BASE_FILE__</code>, page 247.</p> |
| <code>__IAR_SYSTEMS_ICC__</code> | <p>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.</p> <p>This symbol can be tested with <code>#ifdef</code> to detect whether the code was compiled by a compiler from IAR Systems.</p> |
| <code>__ICCH8__</code> | <p>This predefined symbol expands to the number 1 when the code is compiled with the H8 IAR C/C++ Compiler.</p> |
| <code>__INTERRUPT_MODE__</code> | <p>An integer that identifies the interrupt mode used.</p> <p>This symbol reflects the setting of the <code>--interrupt_mode</code> option. The value of the symbol is defined to <code>__INTERRUPT_MODE_0__</code> (0), <code>__INTERRUPT_MODE_1__</code> (1), <code>__INTERRUPT_MODE_2__</code> (2), or <code>__INTERRUPT_MODE_3__</code> (3), for the different interrupt modes, respectively. These symbolic names can be used when testing the <code>__INTERRUPT_MODE__</code> symbol.</p> |
| <code>__LINE__</code> | <p>This predefined symbol expands to the current line number of the file currently being compiled.</p> |
| <code>__LITTLE_ENDIAN__</code> | <p>An integer that identifies the byte order of the microcomputer.</p> <p>For the H8/300H and H8S microcomputer families, the value of this symbol is defined to 0, which means that the byte order is big-endian.</p> |

| | |
|------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>__MAC_ENABLED__</code> | <p>An integer that identifies if the <code>MAC</code> register is enabled.</p> <p>The value of this symbol is defined to 1 when the <code>--enable_mac</code> option is used, which means the <code>MAC</code> register is enabled for use by the compiler.</p> |
| <code>__MAX_CYCLES_NO_INTERRUPT__</code> | <p>An integer that identifies the maximum number of cycles interrupts may be disabled.</p> <p>If you write a sequence of code where it is critical that interrupts are not disabled too long, you may insert a test using this preprocessor symbol in your code.</p> <p>The value of this symbol is defined to the value of the parameter specified when using the <code>--max_cycles_no_interrupt</code> option.</p> <p>If the command line option is not used, the value of this symbol is 0.</p> |
| <code>__MODEL_XXX__</code> | <p>An integer that identifies the code and data model in use.</p> <p>The <code>__MODEL_XXX__</code> symbols can be used as symbolic names when testing the <code>__CODE_MODEL__</code> and <code>__DATA_MODEL__</code> symbols, which reflects the setting of the <code>--code_model</code> and <code>--data_model</code> options, respectively. The value of the symbols are defined to <code>__MODEL_SMALL__</code> (1) for the Small code and data model, <code>__MODEL_LARGE__</code> (2) for the Large code model, and <code>__MODEL_HUGE__</code> (3) for the Huge data model, respectively.</p> |
| <code>__OPERATING_MODE__</code> | <p>An integer that identifies the operating mode used.</p> <p>This symbol reflects the setting of the <code>--operating_mode</code> option. The value of the symbol is defined to <code>__OPERATING_MODE_NORMAL__</code> (1) or <code>__OPERATING_MODE_ADVANCED__</code> (2) for the Normal and Advanced operating modes, respectively. These symbolic names can be used when testing the <code>__OPERATING_MODE__</code> symbol.</p> |
| <code>__STACK_POINTER_SIZE__</code> | <p>An integer that identifies the maximum part of the stack pointer that must be updated when performing stack pointer arithmetics.</p> <p>The value of the symbol is defined to <code>__STACK_POINTER_SIZE_8__</code> (8), <code>__STACK_POINTER_SIZE_16__</code> (16), or <code>__STACK_POINTER_SIZE_32__</code> (32) depending on the setting of the <code>--stack_pointer_size</code> option. These symbolic names can be used when testing the <code>__STACK_POINTER_SIZE__</code> symbol.</p> |

| | |
|-----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>__STDC__</code> | This predefined symbol expands to the number 1. This symbol can be tested with <code>#ifdef</code> to detect whether the compiler in use adheres to ISO/ANSI C. |
|-----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|

| | |
|-------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>__STDC_VERSION__</code> | ISO/ANSI C and version identifier. This predefined symbol expands to 199409L. Note: This predefined symbol does not apply in EC++ mode. |
|-------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|

| | |
|-----------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>__SUBVERSION__</code> | An integer that identifies the version letter of the version number. An integer that identifies the version letter of the version number, for example the C in 4.21C, as an ascii character. |
|-----------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

| | |
|----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>__TID__</code> | Target identifier for the H8 IAR C/C++ Compiler. Expands to the target identifier which contains the following parts: <ul style="list-style-type: none"> ● A one-bit intrinsic flag (<i>i</i>) which is reserved for use by IAR ● A target-identifier (<i>t</i>) unique for each IAR compiler. For the H8/300H and H8S microcomputer, the target identifier is 37 ● A value (<i>c</i>) used in version 1.x of the compiler to specify the value of the chip used; command line option <code>-v</code> ● A value (<i>m</i>) used in version 1.x of the compiler to specify the value of the memory model; command line option <code>-m</code>. <p>In version 2.x of the compiler, the chip and memory model options have been replaced by the command line options <code>--core</code>, <code>--operating_mode</code>, <code>--code_model</code>, and <code>--data_model</code>. As there is no natural correspondence between these new options and the old options <code>-v</code> and <code>-m</code>, the new compiler will specify the values 15 (0xF) for both <i>c</i> and <i>m</i>.</p> <p>The <code>__TID__</code> value is constructed as:</p> $((i \ll 15) (t \ll 8) (c \ll 4) m)$ <p>You can extract the values as follows:</p> <pre>i = (__TID__ >> 15) & 0x01; /* intrinsic flag */ t = (__TID__ >> 8) & 0x7F; /* target identifier */ c = (__TID__ >> 4) & 0x0F; /* chip used; -v option version 1.x */ m = __TID__ & 0x0F; /* memory model; -m option version 1.x */</pre> |
|----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

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 `__ICCH8__`, `__CORE__`, `__OPERATING_MODE__`, `__CODE_MODEL__`, and `__DATA_MODEL__` 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.

Description of miscellaneous preprocessor extensions

The following section gives reference information about the preprocessor extensions that are available in addition to the predefined symbols, pragma directives and ISO/ANSI directives.

`NDEBUG` This preprocessor symbol determines whether any assert macros you have written in your application shall be included or not in the built application.

If this symbol is not defined, all assert macros are evaluated. If the symbol is defined, all assert macros are excluded from the compilation. In other words, if the symbol is:

- **defined**, the assert code will *not* be included
- **not defined**, the assert code will be included

This means that if you have written any assert code and build your application, you should define this symbol to exclude the assert code from the application.

Note that the `assert` macro is defined in the `assert.h` standard include file.



In the IAR Embedded Workbench IDE, the `NDEBUG` symbol is automatically defined if you build your application in the Release build configuration.

```
__Pragma() __Pragma("string")
```

where *string* follows the syntax of the corresponding `pragma` directive.

This preprocessor operator is part of the C99 standard and can be used, for example in `defines` and has the equivalent effect of the `#pragma` directive.

Note: The `-e` option—enable language extensions—is not required.

```
#if NO_OPTIMIZE
    #define NOOPT __Pragma("optimize=2")
#else
    #define NOOPT
#endif
```

See the chapter *Pragma 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);
```

Library functions

This chapter gives an introduction to the C and C++ library functions. It also lists the header files used for accessing library definitions.

For detailed reference information about the library functions, see the online help system.

Introduction

The H8 IAR C/C++ Compiler comes with the IAR DLIB Library, which 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, multibyte characters, et cetera.

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

For detailed information about the library functions, see the online documentation supplied with the product. There is also keyword reference information for the DLIB library functions. To obtain reference information for a function, select the function name in the editor window and press F1.

For additional information about library functions, see the chapter *Implementation-defined behaviour* 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.
- The system startup 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 H8/300H and H8S features. See the chapter *Intrinsic functions* for more information.

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>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 goto 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 47: 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 |

Table 48: Embedded C++ header files

| Header file | Usage |
|---------------------------|-----------------------------------------------------------------------------------|
| <code>exception</code> | Defining several functions that control exception handling |
| <code>fstream</code> | Defining several I/O stream classes that manipulate external files |
| <code>iomanip</code> | Declaring several I/O stream manipulators that take an argument |
| <code>ios</code> | Defining the class that serves as the base for many I/O streams classes |
| <code>iosfwd</code> | Declaring several I/O stream classes before they are necessarily defined |
| <code>iostream</code> | Declaring the I/O stream objects that manipulate the standard streams |
| <code>istream</code> | Defining the class that performs extractions |
| <code>new</code> | Declaring several functions that allocate and free storage |
| <code>ostream</code> | Defining the class that performs insertions |
| <code>sstream</code> | Defining several I/O stream classes that manipulate string containers |
| <code>stdexcept</code> | Defining several classes useful for reporting exceptions |
| <code>streambuf</code> | Defining classes that buffer I/O stream operations |
| <code>string</code> | Defining a class that implements a string container |
| <code>stringstream</code> | Defining several I/O stream classes that manipulate in-memory character sequences |

Table 48: Embedded C++ header files (Continued)

The following table lists additional C++ header files:

| Header file | Usage |
|-------------------------|-----------------------------------------------------------------------|
| <code>fstream.h</code> | Defining several I/O stream classes that manipulate external files |
| <code>iomanip.h</code> | Declaring several I/O stream manipulators that take an argument |
| <code>iostream.h</code> | Declaring the I/O stream objects that manipulate the standard streams |
| <code>new.h</code> | Declaring several functions that allocate and free storage |

Table 49: Additional Embedded C++ header files—DLIB

Extended Embedded C++ standard template library

The following table lists the Extended EC++ standard template library (STL) header files:

| Header file | Description |
|-------------------------|--------------------------------------------------------|
| <code>algorithm</code> | Defines several common operations on sequences |
| <code>deque</code> | A deque sequence container |
| <code>functional</code> | Defines several function objects |
| <code>hash_map</code> | A map associative container, based on a hash algorithm |

Table 50: Standard template library header files

| Header file | Description |
|-----------------------|--------------------------------------------------------|
| <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 50: Standard template library header files (Continued)

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 |
| <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>cstdio</code> | Performing input and output |

Table 51: New standard C header files—DLIB

| Header file | Usage |
|----------------------|--------------------------------------------------|
| <code>cstdlib</code> | Performing a variety of operations |
| <code>cstring</code> | Manipulating several kinds of strings |
| <code>ctime</code> | Converting between various time and date formats |

Table 51: New standard C header files—DLIB (Continued)

Added C functionality

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

The following include files provide these features:

- `ctype.h`
- `inttypes.h`
- `math.h`
- `stdbool.h`
- `stdint.h`
- `stdio.h`
- `stdlib.h`
- `wchar.h`
- `wctype.h`

CTYPE.H

In `ctype.h`, the C99 function `isblank` is defined.

INTTYPES.H

This include file defines the formatters for all types defined in `stdint.h` to be used by the functions `printf`, `scanf`, and all their variants.

MATH.H

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

The following C99 macro symbols are defined:

`HUGE_VALF`, `HUGE_VALL`, `INFINITY`, `NAN`, `FP_INFINITE`, `FP_NAN`, `FP_NORMAL`, `FP_SUBNORMAL`, `FP_ZERO`, `MATH_ERRNO`, `MATH_ERREXCEPT`, `math_errhandling`.

The following C99 macro functions are defined:

`fpclassify`, `signbit`, `isfinite`, `isinf`, `isnan`, `isnormal`, `isgreater`, `isless`, `islessequal`, `islessgreater`, `isunordered`.

The following C99 type definitions are added:

`float_t`, `double_t`.

STDBOOL.H

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

STDINT.H

This include file provides integer characteristics.

STDIO.H

In `stdio.h`, the following C99 functions are defined:

`vscanf`, `vfscanf`, `vsscanf`, `vsnprintf`, `snprintf`

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

The following functions providing I/O functionality for libraries built without `FILE` support are defined:

`__write_array` Corresponds to `fwrite` on `stdout`.

`__ungetchar` Corresponds to `ungetc` on `stdout`.

`__gets` Corresponds to `fgets` on `stdin`.

STDLIB.H

In `stdlib.h`, the following C99 functions are defined:

`_Exit`, `llabs`, `lldiv`, `strtoll`, `strtoull`, `atoll`, `strtof`, `strtold`.

The function `strtod` has added functionality from the C99 standard. For reference information about this functions, see the library reference available from the **Help** menu.

The `__qsorttbl1` function is defined; it provides sorting using a bubble sort algorithm. This is useful for applications that have a limited stack.

WCHAR.H

In `wchar.h`, the following C99 functions are defined:

`vwscanf`, `vswscanf`, `wscanf`, `wcstof`, `wcstolb`.

WCTYPE.H

In `wctype.h`, the C99 function `iswblank` is defined.

Segment reference

The H8 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 H8 IAR C/C++ Compiler. Note that *located* denotes absolute location using the @ operator or the #pragma location directive. The XLINK segment memory type BIT, CODE, CONST, or DATA indicates whether the segment should be placed in ROM or RAM memory areas; see Table 7, *XLINK segment memory types*, page 34.

| Segment | Description | Type |
|-------------|---------------------------------------------------------------------------------------|-------|
| BITVARS | Holds <code>__bitvar</code> declared data objects. | BIT |
| CHECKSUM | Holds a checksum optionally generated by the linker. | CODE |
| CODE16 | Holds <code>__code16</code> declared functions, that is the Small code model. | CODE |
| CODE24 | Holds <code>__code24</code> declared functions, that is the Large code model. | CODE |
| CSTACK | Holds the stack used by C or C++ programs. | DATA |
| DATA8_AC | Holds <i>located</i> <code>__data8</code> declared constant data. | CONST |
| DATA8_AN | Holds <i>located</i> <code>__data8</code> declared uninitialized data. | DATA |
| DATA8_I | Holds non-zero initialized <code>__data8</code> declared data objects. | DATA |
| DATA8_ID | Holds initial values for the non-zero initialized <code>__data8</code> declared data. | CONST |
| DATA8_N | Holds uninitialized <code>__data8</code> declared data. | DATA |
| DATA8_Z | Holds zero-initialized <code>__data8</code> declared data. | DATA |
| DATA16_AC | Holds <i>located</i> <code>__data16</code> declared constant data. | CONST |
| DATA16_AN | Holds <i>located</i> <code>__data16</code> declared uninitialized data. | DATA |
| DATA16_C | Holds <code>__data16</code> declared constant data, including string literals. | CONST |
| DATA16_HEAP | Holds the <code>__data16</code> heap used for dynamically allocated data. | DATA |

Table 52: Segment summary

| Segment | Description | Type |
|-------------|----------------------------------------------------------------------------------------|-------|
| DATA16_I | Holds non-zero initialized <code>__data16</code> declared data objects. | DATA |
| DATA16_ID | Holds initial values for the non-zero initialized <code>__data16</code> declared data. | CONST |
| DATA16_N | Holds uninitialized <code>__data16</code> declared data. | DATA |
| DATA16_Z | Holds zero-initialized <code>__data16</code> declared data. | DATA |
| DATA32_AC | Holds located <code>__data32</code> declared constant data. | CONST |
| DATA32_AN | Holds located <code>__data32</code> declared uninitialized data. | DATA |
| DATA32_C | Holds <code>__data32</code> declared constant data, including string literals. | CONST |
| DATA32_HEAP | Holds the <code>__data32</code> heap used for dynamically allocated data. | DATA |
| DATA32_I | Holds non-zero initialized <code>__data32</code> declared data objects. | DATA |
| DATA32_ID | Holds initial values for the non-zero initialized <code>__data32</code> declared data. | CONST |
| DATA32_N | Holds uninitialized <code>__data32</code> declared data. | DATA |
| DATA32_Z | Holds zero-initialized <code>__data32</code> declared data. | DATA |
| DIFUNCT | Holds the dynamic initialization vector used by C++. | CONST |
| FLIST | Holds the jump-table for functions called indirectly via the vector area. | CONST |
| INTVEC | Holds the interrupt and trap vector tables. | CONST |

Table 52: 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*.

Note that in many of the following examples, the memory range is specified to start at address `0x2`, even though there is memory from address `0x0`. In all cases, address `0x0` is occupied by the reset vector. As a minimum, the reset vector occupies two bytes. In practice, even more of the low memory is occupied by additional reset vectors, interrupt vectors, et cetera. The linker will automatically move the segments as required.

For the `data8` memory, the first byte (`0xFF00`, `0xFFFF00`, or `0xFFFFFFFF00` depending on operating mode or core) is not occupied, and is potentially available for variables. However, a `data8` pointer to this byte would have the value `0`, equal to the `NULL` pointer. For the `data8` memory, this means that segments must be placed so that they do not occupy the first byte.

In the same way, data segments cannot generally occupy the last byte of memory. In the C language, it is legal for a pointer to have a value which is one higher than the address of the last byte of the variable. If a variable is located at the top of memory, a pointer pointing to the last byte of that variable would wrap around and get the value 0, which is identical with the NULL pointer. This is not legal. The NULL pointer must have a value separate from all other legal pointers. In practice, it is not a problem to avoid the last byte of memory. In most devices, the top of memory is reserved for special function registers, and the data segments will be moved down in memory, accordingly.

BITVARS Holds `__bitvar` declared data objects.

Description

This segment holds `__bitvar` declared data objects. Although the H8S core is able to address single bits in the entire memory range, `__bitvar` declared objects can only be placed in the topmost 256 bytes of memory. To place bit variables in other parts of the memory, structures with bitfields can be used instead.

XLINK segment memory type

BIT

Memory range

The memory range for this segment depends on the operating mode and core being used:

- Normal operating mode: 0xFF00–0xFFFF
- Advanced operating mode and H8/300H: 0xFFFF00–0xFFFFFFFF
- Advanced operating mode and H8S: 0xFFFFFFFF00–0xFFFFFFFF

This segment contains bit variables. To identify an individual bit uniquely, it is not sufficient to provide the memory address. The bit number must also be specified. For this reason, this segment uses artificial addresses composed from both the memory address and the bit number:

$$\text{Bit address} = (\text{memory_address}) * 8 + (\text{bit_number})$$

where *bit_number* can be specified from 0 (least significant bit) to 7 (most significant bit).

In the linker command file, you must specify the bit address when declaring the BITVARS segment, not the normal memory address. Because the XLINK segment memory type of this segment is BIT, XLINK will automatically treat the addresses as bit addresses and place this segment correctly in memory.

Example 1

If you intend to locate the BITVARS segment within the memory range 0xFF00-0xFFFF in the linker command file, you must first convert the range to the bit range 0x7F800-0x7FFFF and specify that range instead. This range covers all bits from bit 0 at address 0xFF00 up to bit 7 at address 0xFFFF.

Example 2

If you intend to locate the BITVARS segment within the memory range 0xFFFFFFFF00-0xFFFFFFFF in the linker command file, you must first convert the range to the bit range 0x7FFFFFF800-0x7FFFFFFF. However, XLINK only accepts 32-bit addresses, which means that you must also truncate the range that you finally specify in the linker command file to 0xFFFFF8000-0xFFFFFFFF. XLINK will automatically convert these values to the correct memory range by adding bits to the left, as required.

Access type

Read/write

CHECKSUM Holds a checksum optionally generated by the linker.

Description

The linker can optionally calculate a checksum for all generated raw data bytes. This checksum is located in the CHECKSUM segment. For information about how to generate the checksum, see the *IAR Linker and Library Tools Reference Guide*.

XLINK segment memory type

CODE

Memory range

This segment can be placed anywhere in memory.

Access type

Read-only

CODE16 Holds `__code16` declared program code, that is code generated in the Small code model.

Description

Your functions are placed in this segment when you use the Small code model. The segment is only used in the Normal operating mode.

XLINK segment memory type

CODE

Memory range

This segment can be placed anywhere within the memory range:

0x00002-0xFFFF

Access type

Read-only

CODE24 Holds `__code24` declared program code, that is code generated in the Large code model.

Description

Your functions are placed in this segment when you use the Large code model. The segment is only used in the Advanced operating mode.

XLINK segment memory type

CODE

Memory range

This segment can be placed anywhere within the memory range:

0x000002-0xFFFFF

Access type

Read-only

CSTACK Holds the internal data stack.

Description

This segment holds the internal data stack used by C/C++ programs. This segment is not initialized by the system startup code. For information about specifying the segment placement and its length, see *The stack*, page 40.

XLINK segment memory type

DATA

Memory range

The address range depends on the data model, operating mode, and core being used:

- Normal operating mode (always the Small data model): 0x0002–0xFFFF
- Advanced operating mode, Small data model, and H8/300H: 0x0002–0x7FFF, 0xFF8000–0xFFFFFFFF¹⁾
- Advanced operating mode, Small data model, and H8S: 0x0002–0x7FFF, 0xFFFF8000–0xFFFFFFFF¹⁾
- Advanced operating mode, Huge data model, and H8/300H: 0x000002–FFFFFFF
- Advanced operating mode, Huge data model, and H8S: 0x00000002–FFFFFFF

¹⁾ Note that the stack segment must be contiguous and must be placed in one of the memory ranges.

Access type

Read/write

DATA8_AC Holds located `__data8` declared constants.

Description

This segment holds constants declared using the `__data8` memory attribute and the IAR absolute location placement extension (the `@` operator alternatively the `#pragma location` directive). These constants do not need to—and must not—be given a location using a segment placement directive to the linker.

XLINK segment memory type

CONST

Memory range

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

Access type

Read-only

DATA8_AN Holds located `__data8` declared non-initialized data.

Description

This segment holds non-initialized data declared using the `__data8` memory attribute and the IAR absolute location placement extension (the `@` operator alternatively the `#pragma location` directive). This data does not need to—and must not—be given a location using segment placements options to the linker.

XLINK segment memory type

DATA

Memory range

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

Access type

Read/write

DATA8_I Holds static and global non-zero initialized `__data8` declared data objects.

Description

This segment holds static and global `__data8` declared data objects, which have a non-zero initial value. The initial values are copied to this segment from the `DATA8_ID` segment by the system startup code.

When you define this segment in the linker command file, the `-Z` segment control directive must be used and not the `-P` directive. The `-P` directive does not guarantee a fixed order between the segment parts when they are copied from the `*_ID` segments to their corresponding `*_I` segments.

In addition to non-zero initialized variables, `__data8` declared constant data is silently converted to initialized variables instead of being placed in a `_C` segment. To have a `DATA8_C` segment would require programmable read-only memory in the topmost 256 bytes of memory, which is not the case for many H8 devices. Instead, the topmost 256 bytes are typically occupied by special function registers and a small RAM area.

XLINK segment memory type

DATA

Memory range

The address range depends on the used operating mode and core:

- Normal operating mode: `0xFF01-0xFFFE`
- Advanced operating mode and H8/300H: `0xFFFF01-0xFFFFFE`
- Advanced operating mode and H8S: `0xFFFFF01-0xFFFFFEE`

Access type

Read/write

`DATA8_ID` Holds initial values for the non-zero initialized `__data8` declared data objects.

Description

This segment holds the initial values for the non-zero initialized `__data8` declared data objects. The initial values are copied from `DATA8_ID` to `DATA8_I` by the system startup code.

When you define this segment in the linker command file, the `-Z` segment control directive must be used and not the `-P` directive. The `-P` directive does not guarantee a fixed order between the segment parts when they are copied from the `*_ID` segments to their corresponding `*_I` segments.

XLINK segment memory type

CONST

Memory range

This segment can be placed anywhere in the memory addressable by the selected operating mode.

Access type

Read-only

DATA8_N Holds static and global uninitialized `__data8` data objects.

Description

This segment holds static and global uninitialized `__data8` data objects that will not be initialized at system startup. Variables defined using the `__no_init` keyword will be placed in this segment.

XLINK segment memory type

DATA

Memory range

The address range depends on the operating mode and core being used:

- Normal operating mode: 0xFF01–0xFFFE
- Advanced operating mode and H8/300H: 0xFFFF01–0xFFFFFE
- Advanced operating mode and H8S: 0xFFFFF01–0xFFFFFEE

Access type

Read/write

DATA8_Z Holds static and global zero-initialized `__data8` declared data objects.

Description

This segment holds static and global `__data8` declared data objects, which have a zero initial value.

XLINK segment memory type

DATA

Memory range

The address range depends on the operating mode and core being used:

- Normal operating mode: 0xFF01–0xFFFE
- Advanced operating mode and H8/300H: 0xFFFF01–0xFFFFFE
- Advanced operating mode and H8S: 0xFFFFF01–0xFFFFFEE

Access type

Read/write

DATA16_AC Holds located `__data16` declared constants.

Description

This segment holds constants declared using the `__data16` memory attribute and the IAR absolute location placement extension (the `@` operator alternatively the `#pragma location` directive). These constants do not need to—and must not—be given a location using a segment placement directive to the linker.

XLINK segment memory type

CONST

Memory range

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

Access type

Read-only

DATA16_AN Holds located `__data16` declared non-initialized data.

Description

This segment holds non-initialized data declared using the `__data16` memory attribute and the IAR absolute location placement extension (the `@` operator alternatively the `#pragma location` directive). This data does not need to—and must not—be given a location using segment placements options to the linker.

XLINK segment memory type

DATA

Memory range

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

Access type

Read/write

 DATA16_C Holds `__data16` constant data, including string literals.
Description

This segment holds `__data16` constant data, including string literals, which can be placed in ROM memory.

XLINK segment memory type

CONST

Memory range

The address range depends on the operating mode and core being used:

- Normal operating mode: 0x0002–0xFFFFE
- Advanced operating mode and H8/300H: 0x000002–0x007FFF, 0xFF8000–0xFFFFFE
- Advanced operating mode and H8S: 0x00000002–0x00007FFF, 0xFFFF8000–0xFFFFF7FE

Access type

Read-only

 DATA16_HEAP Holds the `__data16` heap used for dynamically allocated data.
Description

This segment holds the `__data16` heap used for dynamically allocated data, in other words data used by `malloc` and `free`, and in C++, `new` and `delete`.

For more information about this segment and its length, dynamically allocated data and the heap, see *The heap*, page 41. For information about using the `new` and `delete` operators for a heap in different memory types, see *New and Delete operators*, page 109.

XLINK segment memory type

DATA

Memory range

The address range depends on the operating mode and core being used:

- Normal operating mode: 0x0002–0xFFFFE
- Advanced operating mode and H8/300H: 0x000002–0x007FFF, 0xFF8000–0xFFFFFE
- Advanced operating mode and H8S: 0x00000002–0x00007FFF, 0xFFFF8000–0xFFFFFEE

Access type

Read/write

DATA16_I Holds static and global non-zero initialized `__data16` declared data objects.

Description

This segment holds static and global `__data16` declared data objects, which have a non-zero initial value. The initial values are copied to this segment from the `DATA16_ID` segment by the system startup code.

When you define this segment in the linker command file, the `-z` segment control directive must be used and not the `-P` directive. The `-P` directive does not guarantee a fixed order between the segment parts when they are copied from the `*_ID` segments to their corresponding `*_I` segments.

XLINK segment memory type

DATA

Memory range

The address range depends on the operating mode and core being used:

- Normal operating mode: 0x0002–0xFFFFE
- Advanced operating mode and H8/300H: 0x000002–0x007FFF, 0xFF8000–0xFFFFFE
- Advanced operating mode and H8S: 0x00000002–0x00007FFF, 0xFFFF8000–0xFFFFFEE

Access type

Read/write

DATA16_ID Holds initial values for the non-zero initialized `__data16` declared data objects.

Description

This segment holds the initial values for the non-zero initialized `__data16` declared data objects. The initial values are copied from `DATA16_ID` to `DATA16_I` by the system startup code.

When you define this segment in the linker command file, the `-Z` segment control directive must be used and not the `-P` directive. The `-P` directive does not guarantee a fixed order between the segment parts when they are copied from the `*_ID` segments to their corresponding `*_I` segments.

XLINK segment memory type

CONST

Memory range

This segment can be placed anywhere in the memory addressable by the selected operating mode.

Access type

Read-only

DATA16_N Holds static and global uninitialized `__data16` data objects.

Description

This segment holds static and global uninitialized `__data16` data objects that will not be initialized at system startup. Variables defined using the `__no_init` keyword will be placed in this segment.

XLINK segment memory type

DATA

Memory range

The address range depends on the operating mode and core being used:

- Normal operating mode: `0x0002-0xFFFFE`
- Advanced operating mode and H8/300H: `0x000002-0x007FFF, 0xFF8000-0xFFFFFE`

- Advanced operating mode and H8S: 0x00000002-0x00007FFF, 0xFFFF8000-0xFFFFFFFFE

Access type

Read/write

DATA16_Z Holds static and global zero-initialized `__data16` declared data objects.

Description

This segment holds static and global `__data16` declared data objects, which have a zero initial value.

XLINK segment memory type

DATA

Memory range

The address range depends on the operating mode and core being used:

- Normal operating mode: 0x0002-0xFFFFE
- Advanced operating mode and H8/300H: 0x000002-0x007FFF, 0xFF8000-0xFFFFFE
- Advanced operating mode and H8S: 0x00000002-0x00007FFF, 0xFFFF8000-0xFFFFFFFFE

Access type

Read/write

DATA32_AC Holds located `__data32` declared constants.

Description

This segment holds constants declared using the `__data32` memory attribute and the IAR absolute location placement extension (the `@` operator alternatively the `#pragma location` directive). These constants do not need to—and must not—be given a location using a segment placement directive to the linker.

XLINK segment memory type

CONST

Memory range

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

Access type

Read-only

DATA32_AN Holds located `__data32` declared non-initialized data.

Description

This segment holds non-initialized data declared using the `__data32` memory attribute and the IAR absolute location placement extension (the `@` operator alternatively the `#pragma location` directive). This data does not need to—and must not—be given a location using segment placements options to the linker.

XLINK segment memory type

DATA

Memory range

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

Access type

Read/write

DATA32_C Holds `__data32` constant data, including string literals.

Description

This segment holds `__data32` constant data, including string literals, which can be placed in ROM memory.

XLINK segment memory type

CONST

Memory range

The address range depends on the core in use:

- H8/300H: 0x0000002-0xFFFFFE
- H8S: 0x00000002-0xFFFFFFFF

Access type

Read-only

DATA32_HEAP Holds the `__data32` heap used for dynamically allocated data.

Description

This segment holds the `__data32` heap used for dynamically allocated data, in other words data used by `malloc` and `free`, and in C++, `new` and `delete`.

For more information about this segment and its length, dynamically allocated data and the heap, see *The heap*, page 41. For information about using the `new` and `delete` operators for a heap in different memory types, see *New and Delete operators*, page 109.

XLINK segment memory type

DATA

Memory range

The address range depends on the core in use:

- H8/300H: 0x0000002-0xFFFFFE
- H8S: 0x00000002-0xFFFFFFFF

Access type

Read/write

DATA32_I Holds static and global non-zero initialized `__data32` declared data objects.

Description

This segment holds static and global `__data32` declared data objects, which have a non-zero initial value. The initial values are copied to this segment from the `DATA32_ID` segment by the system startup code.

When you define this segment in the linker command file, the `-Z` segment control directive must be used and not the `-P` directive. The `-P` directive does not guarantee a fixed order between the segment parts when they are copied from the `*_ID` segments to their corresponding `*_I` segments.

XLINK segment memory type

DATA

Memory range

The address range depends on the used core:

- H8/300H: 0x000002-0xFFFFFE
- H8S: 0x0000002-0xFFFFFEE

Access type

Read/write

DATA32_ID Holds initial values for the non-zero initialized `__data32` declared data objects.

Description

This segment holds the initial values for the non-zero initialized `__data32` declared data objects. The initial values are copied from `DATA8_ID` to `DATA8_I` by the system startup code.

When you define this segment in the linker command file, the `-Z` segment control directive must be used and not the `-P` directive. The `-P` directive does not guarantee a fixed order between the segment parts when they are copied from the `*_ID` segments to their corresponding `*_I` segments.

XLINK segment memory type

CONST

Memory range

This segment can be placed anywhere in memory.

Access type

Read-only

DATA32_N Holds static and global uninitialized `__data32` data objects.

Description

This segment holds static and global uninitialized `__data32` data objects that will not be initialized at system startup. Variables defined using the `__no_init` keyword will be placed in this segment.

XLINK segment memory type

DATA

Memory range

The address range depends on the core being used:

- H8/300H: 0x000002-0xFFFFFE
- H8S: 0x0000002-0xFFFFFEE

Access type

Read/write

DATA32_Z Holds static and global zero-initialized `__data32` declared data objects.

Description

This segment holds static and global `__data32` declared data objects, which have a zero initial value.

XLINK segment memory type

DATA

Memory range

The address range depends on the core being used:

- H8/300H: 0x000002-0xFFFFFE
- H8S: 0x0000002-0xFFFFFEE

Access type

Read/write

DIFUNCT Holds the dynamic initialization vector used by C++.

Description

This segment holds information required to call constructors for C++ static objects. In other words, this segment holds pointers to code, typically C++ constructors, which should be executed by the system startup code before `main` is called.

XLINK segment memory type

CONST

Memory range

The address range depends on the data model, operating mode, and core being used:

- Normal operating mode (always the Small data model): `0x0002-0xFFFF`
- Advanced operating mode, Small data model, and H8/300H: `0x0002-0x7FFF, 0xFF8000-0xFFFFFFFF`¹⁾
- Advanced operating mode, Small data model, and H8S: `0x0002-0x7FFF, 0xFFFF8000-0xFFFFFFFF`¹⁾
- Advanced operating mode, Huge data model, and H8/300H: `0x000002-FFFFFF`
- Advanced operating mode, Huge data model, and H8S: `0x00000002-FFFFFF`

¹⁾Note that the `DIFUNCT` segment must be contiguous and must be placed in one of the memory ranges.

Access type

Read-only

FLIST Holds the jump-table for functions called indirectly via the vector area.

Description

This segment holds the jump-table for functions called indirectly via the vector area, which means the `@aa:8` addressing mode is used. This includes:

- Many library routines, unless the compiler option `--direct_library_calls` are used
- All `__vector_call` declared functions.

XLINK segment memory type

CONST

Memory range

0x02-0xFF

Access type

Read-only

INTVEC Holds the interrupt and trap vector tables.

Description

This segment holds the interrupt and trap vector tables, which are generated for all functions declared either `__interrupt` or `__trap` in combination with the `#pragma vector` directive.

XLINK segment memory type

CONST

Memory range

Interrupt vectors are normally stored in the lower part of memory, from address 0x0 to at most address 0x3FF. However, there is no formal upper limit for the segment. In the default linker command files, 0x3FF is used as upper limit.

Access type

Read-only

Implementation-defined behavior

This chapter describes how the H8/300H and H8S 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 H8/300H and H8S 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 DLIB runtime environment, see *Customizing system initialization*, page 65.

Interactive devices (5.1.2.3)

The streams `stdin` and `stdout` are treated as interactive devices.

IDENTIFIERS

Significant characters without external linkage (6.1.2)

The number of significant initial characters in an identifier without external linkage is 200.

Significant characters with external linkage (6.1.2)

The number of significant initial characters in an identifier with external linkage is 200.

Case distinctions are significant (6.1.2)

Identifiers with external linkage are treated as case-sensitive.

CHARACTERS

Source and execution character sets (5.2.1)

The source character set is the set of legal characters that can appear in source files. The default source character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the source character set will be the host computer's default character set.

The execution character set is the set of legal characters that can appear in the execution environment. The default execution character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the execution character set will be the host computer's default character set. The IAR DLIB Library needs a multibyte character scanner to support a multibyte execution character set

See *Locale*, page 70.

Bits per character in execution character set (5.2.4.2.1)

The number of bits in a character is represented by the manifest constant `CHAR_BIT`. The standard include file `limits.h` defines `CHAR_BIT` as 8.

Mapping of characters (6.1.3.4)

The mapping of members of the source character set (in character and string literals) to members of the execution character set is made in a one-to-one way. In other words, the same representation value is used for each member in the character sets except for the escape sequences listed in the ISO standard.

Unrepresented character constants (6.1.3.4)

The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or in the extended character set for a wide character constant generates a diagnostic message, and will be truncated to fit the execution character set.

Character constant with more than one character (6.1.3.4)

An integer character constant that contains more than one character will be treated as an integer constant. The value will be calculated by treating the leftmost character as the most significant character, and the rightmost character as the least significant character, in an integer constant. A diagnostic message will be issued if the value cannot be represented in an integer constant.

A wide character constant that contains more than one multibyte character generates a diagnostic message.

Converting multibyte characters (6.1.3.4)

The only locale supported—that is, the only locale supplied with the IAR C/C++ Compiler—is the ‘C’ locale. If you use the command line option `--enable_multibytes`, the IAR DLIB Library will support multibyte characters if you add a locale with multibyte support or a multibyte character scanner to the library. See *Locale*, page 70.

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 176, 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 178, 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 181, for information about *size_t*.

Conversion from/to pointers (6.3.4)

See *Casting*, page 180, for information about casting of data pointers and function pointers.

ptrdiff_t (6.3.6, 7.1.1)

See *ptrdiff_t*, page 181, 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 176, for information about the alignment requirement for data objects.

Sign of 'plain' bitfields (6.5.2.1)

A 'plain' `int` bitfield is treated as a signed `int` bitfield. All integer types are allowed as bitfields.

Allocation order of bitfields within a unit (6.5.2.1)

Bitfields are allocated within an integer from least-significant to most-significant bit.

Can bitfields straddle a storage-unit boundary (6.5.2.1)

Bitfields cannot straddle a storage-unit boundary for the chosen bitfield integer type.

Integer type chosen to represent enumeration types (6.5.2.2)

The chosen integer type for a specific enumeration type depends on the enumeration constants defined for the enumeration type. The chosen integer type is the smallest possible.

QUALIFIERS

Access to volatile objects (6.5.3)

Any reference to an object with volatile qualified type is an access.

DECLARATORS

Maximum numbers of declarators (6.5.4)

The number of declarators is not limited. The number is limited only by the available memory.

STATEMENTS

Maximum number of case statements (6.6.4.2)

The number of case statements (case values) in a switch statement is not limited. The number is limited only by the available memory.

PREPROCESSING DIRECTIVES

Character constants and conditional inclusion (6.8.1)

The character set used in the preprocessor directives is the same as the execution character set. The preprocessor recognizes negative character values if a 'plain' character is treated as a `signed` character.

Including bracketed filenames (6.8.2)

For file specifications enclosed in angle brackets, the preprocessor does not search directories of the parent files. A parent file is the file that contains the `#include` directive. Instead, it begins by searching for the file in the directories specified on the compiler command line.

Including quoted filenames (6.8.2)

For file specifications enclosed in quotes, the preprocessor directory search begins with the directories of the parent file, then proceeds through the directories of any grandparent files. Thus, searching begins relative to the directory containing the source file currently being processed. If there is no grandparent file and the file has not been found, the search continues as if the filename was enclosed in angle brackets.

Character sequences (6.8.2)

Preprocessor directives use the source character set, with the exception of escape sequences. Thus, to specify a path for an include file, use only one backslash:

```
#include "mydirectory\myfile"
```

Within source code, two backslashes are necessary:

```
file = fopen("mydirectory\\myfile", "rt");
```

Recognized pragma directives (6.8.6)

In addition to the pragma directives described in the chapter *Pragma directives*, the following directives are recognized but will have no effect:

```
alignment
ARGSUSED
baseaddr
can_instantiate
codeseg
cspy_support
define_type_info
do_not_instantiate
function
hdrstop
instantiate
keep_definition
memory
module_name
none
no_pch
NOTREACHED
```

```

once
__printf_args
public_equ
__scanf_args
system_include
VARARGS
warnings

```

Default `__DATE__` and `__TIME__` (6.8.8)

The definitions for `__TIME__` and `__DATE__` are always available.

IAR DLIB LIBRARY FUNCTIONS

The information in this section is valid only if the runtime library configuration you have chosen supports file descriptors. See the chapter *The DLIB runtime environment* for more information about runtime library configurations.

NULL macro (7.1.6)

The `NULL` macro is defined to 0.

Diagnostic printed by the `assert` function (7.2)

The `assert()` function prints:

```
filename:linenr expression -- assertion failed
```

when the parameter evaluates to zero.

Domain errors (7.5.1)

`NaN` (Not a Number) will be returned by the mathematic functions on domain errors.

Underflow of floating-point values sets `errno` to `ERANGE` (7.5.1)

The mathematics functions set the integer expression `errno` to `ERANGE` (a macro in `errno.h`) on underflow range errors.

`fmod()` functionality (7.5.6.4)

If the second argument to `fmod()` is zero, the function returns `NaN`; `errno` is set to `EDOM`.

`signal()` (7.7.1.1)

The signal part of the library is not supported.

Note: Low-level interface functions exist in the library, but will not perform anything. Use the template source code to implement application-specific signal handling. See *Signal and raise*, page 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 69.

`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 69.

`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 69.

`%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 72.

system() (7.10.4.5)

How the command processor works depends on how you have implemented the `system` function. See *Environment interaction*, page 72.

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 |

Table 53: Message returned by `strerror()`—IAR DLIB library

| Argument | Message |
|------------|---------------------------|
| ERANGE | range error |
| EFPOS | file positioning error |
| EILSEQ | multi-byte encoding error |
| <0 >99 | unknown error |
| all others | error <i>nnn</i> |

Table 53: Message returned by `strerror()`—IAR DLIB library (Continued)

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.

A

abort, system termination (DLIB) 64
 absolute location
 data, placing at (@) 46
 language support for 189
 #pragma location 220
 addressing. *See* memory types
 algorithm (STL header file) 258
 alignment 175
 forcing stricter (#pragma data_alignment) 217
 of data types 176
 alignment (pragma directive) 289
 __ALIGNOF__ (operator) 189
 __and_ccr (intrinsic function) 229
 __and_exr (intrinsic function) 229
 anonymous structures 124
 anonymous symbols, creating 191
 applications
 building 4
 initializing 64
 terminating 64
 ARGFRAME (compiler function directive) 98
 ARGSUSED (pragma directive) 289
 arrays 194
 hints about index type 121
 implementation-defined behavior 287
 non-lvalue 196
 asm, __asm (language extension) 191
 assembler code
 calling from C 86
 calling from C++ 88
 inline 85
 language extension 191
 assembler directives
 CFI 101
 EQU 169
 PUBLIC 169
 RTMODEL 78

assembler instructions
 MOVFPPE 238
 MOVTPE 238
 NOP 238
 RTE 213
 SLEEP 241
 TAS 241
 TRAPA 27, 242
 assembler language interface 83
 creating skeleton code 86
 assembler list file 98
 asserts 74
 including in application (NDEBUG) 252
 assert.h (library header file) 257
 assumptions (programming experience) xvii
 atoll, in stdlib.h 261
 atomic operations 28
 __monitor 211
 auto variables 22

B

baseaddr (pragma directive) 289
 __BASE_FILE__ (predefined symbol) 247
 basic_template_matching (pragma directive) 216
 using 111
 __bcd_add_char (intrinsic function) 230
 __bcd_add_short (intrinsic function) 230
 __bcd_subtract_char (intrinsic function) 230
 __bcd_subtract_short (intrinsic function) 230
 bitfields
 data representation 177
 hints 121
 implementation-defined behavior 287
 non-standard types in 189
 specifying order of members (#pragma bitfields) 217
 __bitvar (extended keyword) 204
 bitvar (memory type) 16
 BITVARS (segment) 265

| | |
|-----------------------------------------------------|-----|
| bool (data type) | 176 |
| adding support for in DLIB | 257 |
| making available in C code | 261 |
| supported in C code | 176 |
| bubble sort algorithm, adding support for | 261 |
| __BUILD_NUMBER__ (predefined symbol) | 247 |
| --bus_width (compiler option) | 148 |
| __bus_width (runtime model attribute) | 78 |

C

| | |
|-----------------------------------------------------------------------------|-----|
| C and C++ linkage | 91 |
| C calling convention | 89 |
| C header files | 257 |
| call chains | 126 |
| call stack | 101 |
| callee-save registers, stored on stack | 22 |
| calling convention | |
| C | 89 |
| C++, requiring C linkage | 88 |
| overriding default (__cc_version1) | 204 |
| overriding default (__cc_version2) | 205 |
| overriding default (__cc_version3) | 205 |
| calloc (standard library function) | 23 |
| can_instantiate (pragma directive) | 289 |
| cassert (library header file) | 259 |
| cast operators | |
| in Extended EC++ | 104 |
| missing from Embedded C++ | 104 |
| casting | |
| between pointer types | 19 |
| of pointers and integers | 180 |
| cctype (library header file) | 259 |
| __cc_version1 (extended keyword) | 204 |
| __cc_version2 (extended keyword) | 205 |
| __cc_version3 (extended keyword) | 205 |
| cerrno (library header file) | 259 |
| CFI (assembler directive) | 101 |
| cfloat (library header file) | 259 |
| char (data type) | 176 |
| changing default representation (--char_is_signed) | 149 |
| signed and unsigned | 177 |
| characters, implementation-defined behavior | 284 |
| --char_is_signed (compiler option) | 149 |
| CHECKSUM (segment) | 266 |
| class memory (extended EC++) | 106 |
| class template partial specialization matching (extended EC++) | 110 |
| classes | 105 |
| climits (library header file) | 259 |
| locale (library header file) | 259 |
| __close (library function) | 70 |
| cmath (library header file) | 259 |
| code execution | 7 |
| code models | |
| configuration | 7 |
| Large | 26 |
| overview | 25 |
| Small | 26 |
| specifying on command line | 149 |
| code motion | |
| compiler transformation | 120 |
| disabling (--no_code_motion) | 163 |
| codeseg (pragma directive) | 289 |
| code, placement of | 263 |
| __CODE_COMPATIBILITY_CHECK__ (predefined symbol) | 247 |
| __CODE_MODEL__ (predefined symbol) | 248 |
| --code_model (compiler option) | 149 |
| __code_model (runtime model attribute) | 78 |
| __code16 (extended keyword) | 206 |
| CODE16 (segment) | 267 |
| __code24 (extended keyword) | 206 |
| CODE24 (segment) | 267 |
| comments | |
| after preprocessor directives | 196 |
| C++ style, using in C code | 191 |
| common subexpression elimination | |
| compiler transformation | 119 |

- disabling (`--no_cse`) 163
- compiler
 - environment variables 136
 - invocation syntax 135
- compiler listing, generating (`-l`). 159
- compiler object file
 - including debug information in (`--debug, -r`) 151
 - specifying filename of (`-o`). 166
- compiler options 143
 - specifying parameters 145
 - summary 146
 - typographic convention xx
 - `-D`. 150
 - `-e` 156
 - `-f` 158
 - `-I` 159
 - `-l` 87, 159
 - `-o` 166
 - `-r` 169
 - `-s` 170
 - `-z` 173
 - `--bus_width` 148
 - `--char_is_signed`. 149
 - `--code_model`. 149
 - `--core` 150
 - `--data_model` 151
 - `--debug`. 151
 - `--dependencies` 152
 - `--diagnostics_tables` 154
 - `--diag_error` 153
 - `--diag_remark` 153
 - `--diag_suppress` 153
 - `--diag_warning`. 154
 - `--direct_library_calls` 155
 - `--dlib_config` 155
 - `--double` 155
 - `--ec++` 156
 - `--eec++`. 157
 - `--enable_mac` 157
 - `--enable_multibytes` 157
 - `--error_limit` 158
 - `--header_context` 158
 - `--interrupt_mode` 159
 - `--library_module` 160
 - `--max_cycles_no_interrupt` 161
 - `--migration_preprocessor_extensions`. 161
 - `--misrac` 162
 - `--misrac_verbose` 162
 - `--module_name` 163
 - `--no_code_motion` 163
 - `--no_cse` 163
 - `--no_inline` 164
 - `--no_tbaa` 164
 - `--no_typedefs_in_diagnostics` 165
 - `--no_unroll` 165
 - `--no_warnings` 166
 - `--no_wrap_diagnostics` 166
 - `--omit_types`. 167
 - `--only_stdout` 167
 - `--operating_mode`. 167
 - `--preinclude` 168
 - `--preprocess` 168
 - `--public_equ`. 168
 - `--remarks` 169
 - `--require_prototypes`. 169
 - `--silent` 170
 - `--stack_pointer_size`. 171
 - `--strict_ansi` 171
 - `--warnings_affect_exit_code` 139, 172
 - `--warnings_are_errors` 172
 - `--weak_rtmodel_check`. 172
 - compiler subversion number 251
 - compiler version number 252
 - compiling, from the command line 4
 - complex numbers, supported in Embedded C++. 104
 - complex (library header file). 257
 - compound literals 191
 - computer style, typographic convention xx

| | |
|--------------------------------------------------------|-----|
| configuration | |
| basic project settings | 5 |
| hardware | 7 |
| __low_level_init | 65 |
| configuration symbols, in library configuration files | 61 |
| consistency, module | 77 |
| constseg (pragma directive) | 217 |
| const, declaring objects | 184 |
| const_cast (cast operator) | 104 |
| conventions, typographic | xx |
| copyright notice | ii |
| __CORE__ (predefined symbol) | 248 |
| core | |
| configuration | 6 |
| specifying on command line | 150 |
| --core (compiler option) | 150 |
| __CORE_H8S__ (predefined symbol) | 248 |
| __CORE_H8300H__ (predefined symbol) | 248 |
| __cplusplus (predefined symbol) | 248 |
| csetjmp (library header file) | 259 |
| csignal (library header file) | 259 |
| cspy_support (pragma directive) | 289 |
| CSTACK (segment) | |
| example | 40 |
| <i>See also</i> stack | |
| cstartup, customizing | 66 |
| cstdarg (library header file) | 259 |
| cstddef (library header file) | 259 |
| cstdlib (library header file) | 259 |
| cstdliblib (library header file) | 260 |
| cstring (library header file) | 260 |
| ctime (library header file) | 260 |
| ctype.h (library header file) | 257 |
| added C functionality | 260 |
| C++ | |
| calling convention | 88 |
| features excluded from EC++ | 103 |
| language extensions | 114 |
| <i>See also</i> Embedded C++ and Extended Embedded C++ | |

| | |
|----------------------------------------|---------|
| terminology | xx |
| C++ header files | 257–258 |
| C++ names, in assembler code | 89 |
| C++-style comments in C code | 191 |
| C-SPY | |
| low-level interface | 75 |
| STL container support | 113 |
| C_INCLUDE (environment variable) | 136 |
| C99 standard, added functionality from | 260 |

D

| | |
|----------------------------------------|-----|
| __dadd (intrinsic function) | 231 |
| data | |
| alignment of | 175 |
| at absolute location | 46 |
| placement of | 263 |
| data memory attributes, using | 18 |
| data models | 14 |
| configuration | 6 |
| data pointers | 180 |
| data representation | 175 |
| data storage | 13 |
| data types | 176 |
| floating-point | 178 |
| integers | 176 |
| dataseg (pragma directive) | 218 |
| data_alignment (pragma directive) | 217 |
| __DATA_MODEL__ (predefined symbol) | 248 |
| --data_model (compiler option) | 151 |
| __data_model (runtime model attribute) | 78 |
| data8 (memory type) | 16 |
| __data8 (extended keyword) | 207 |
| DATA8_AC (segment) | 268 |
| DATA8_AN (segment) | 269 |
| DATA8_I (segment) | 269 |
| DATA8_ID (segment) | 270 |
| DATA8_N (segment) | 271 |
| DATA8_Z (segment) | 271 |

- `__data16` (extended keyword) 208
- `data16` (memory type) 17
- `DATA16_AC` (segment) 272
- `DATA16_AN` (segment) 272
- `DATA16_C` (segment) 273
- `DATA16_I` (segment) 274
- `DATA16_ID` (segment) 270, 275, 279
- `DATA16_N` (segment) 271, 275, 280
- `DATA16_Z` (segment) 276
- `__data32` (extended keyword) 209
- `data32` (memory type) 17
- `DATA32_AC` (segment) 276
- `DATA32_AN` (segment) 277
- `DATA32_C` (segment) 277
- `DATA32_HEAP` (segment) 278
- `DATA32_I` (segment) 278
- `DATA32_ID` (segment) 279
- `DATA32_N` (segment) 280
- `DATA32_Z` (segment) 280
- `__DATE__` (predefined symbol) 248
- `date` (library function), configuring support for 74
- `--debug` (compiler option) 151
- debug information, including in object file 151, 169
- declaration
 - functions 91
- declarations
 - empty 196
 - in for loops 190
- declarations and statements, mixing 190
- declarators, implementation-defined behavior 288
- `define_type_info` (pragma directive) 289
- delete operator (extended EC++) 109
- delete (keyword) 23
- `--dependencies` (compiler option) 152
- deque (STL header file) 258
- destructors and interrupts, using 114
- diagnostic messages 139
 - classifying as errors 153
 - classifying as remarks 153
 - classifying as warnings 154
 - disabling warnings 166
 - disabling wrapping of 166
 - enabling remarks 169
 - listing all used 154
 - suppressing 153
- `--diagnostics_tables` (compiler option) 154
- `diag_default` (pragma directive) 218
- `--diag_error` (compiler option) 153
- `diag_error` (pragma directive) 218
- `--diag_remark` (compiler option) 153
- `diag_remark` (pragma directive) 218
- `--diag_suppress` (compiler option) 153
- `diag_suppress` (pragma directive) 218
- `--diag_warning` (compiler option) 154
- `diag_warning` (pragma directive) 219
- `DIFUNCT` (segment) 45, 264, 281
- directives
 - function 98
 - pragma 10, 215
- directory, specifying as parameter 145
- `__DIRECT_LIBRARY_CALLS__` (predefined symbol) . 248
- `--direct_library_calls` (compiler option) 155
- `__disable_interrupt` (intrinsic function) 231
- disclaimer ii
- `DLIB` 8, 256
 - reference information. *See* the online help system . . . 255
- `--dlib_config` (compiler option) 155
- document conventions xx
- documentation, library 255
- `--double` (compiler option) 155
- double (data type) 178
- double, configuring size of floating-point type 8
- `__DOUBLE_SIZE__` (predefined symbol) 249
- `__double_size` (runtime model attribute) 78
- `double_t`, in `math.h` 261
- `__do_byte_eepmov` (intrinsic function) 231
- `do_not_instantiate` (pragma directive) 289
- `__do_word_eepmov` (intrinsic function) 232

| | |
|--------------------------------------|-----------|
| __dsb (intrinsic function) | 232 |
| dynamic initialization | 58–59, 63 |
| in Embedded C++ | 45 |
| dynamic memory | 23 |

E

| | |
|---------------------------------------------------------|--------|
| --ec++ (compiler option) | 156 |
| EC++ header files | 257 |
| --eec++ (compiler option) | 157 |
| __eepmov (intrinsic function) | 233 |
| __eepmovi (intrinsic function) | 234 |
| EEPMOV.B, enable use by disabling interrupts | 161 |
| Embedded C++ | 103 |
| absolute location | 46, 48 |
| differences from C++ | 103 |
| dynamic initialization in | 45 |
| enabling | 156 |
| function linkage | 91 |
| language extensions | 103 |
| overview | 103 |
| special function types | 31 |
| static member variables | 46, 48 |
| Embedded C++ objects, placing in memory type | 21 |
| __embedded_cplusplus (predefined symbol) | 249 |
| __enable_interrupt (intrinsic function) | 235 |
| --enable_mac (compiler option) | 157 |
| --enable_multibytes (compiler option) | 157 |
| enumerations, implementation-defined behavior | 287 |
| enums | |
| data representation | 177 |
| forward declarations of | 194 |
| environment | |
| implementation-defined behavior | 284 |
| environment variables | 136 |
| C_INCLUDE | 136 |
| QCCH8 | 136 |
| EQU (assembler directive) | 169 |
| error messages | 140 |

| | |
|---------------------------------------------------------|---------|
| classifying | 153 |
| error return codes | 139 |
| exception handling, missing from Embedded C++ | 103 |
| exception vectors | 44 |
| exception (library header file) | 258 |
| _Exit (exit function) | 64 |
| exit (exit function) | 64 |
| _exit (exit function) | 64 |
| __exit (exit function) | 64 |
| experience, programming | xvii |
| export keyword, missing from Extended EC++ | 110 |
| Extended Embedded C++ | 104 |
| enabling | 157 |
| standard template library (STL) | 258 |
| extended keywords | 199 |
| enabling | 156 |
| overview | 10 |
| summary | 203 |
| syntax | 18 |
| __bitvar | 204 |
| __cc_version1 | 204 |
| __cc_version2 | 205 |
| __cc_version3 | 205 |
| __code16 | 206 |
| __code16 (function pointer) | 179 |
| __code24 | 206 |
| __code24 (function pointer) | 179 |
| __data8 | 207 |
| __data8 (data pointer) | 180 |
| __data16 | 208 |
| __data16 (data pointer) | 180 |
| __data32 | 209 |
| __data32 (data pointer) | 180 |
| __interrupt | 27, 210 |
| <i>See also</i> INTVEC (segment) | |
| using in pragma directives | 226 |
| __intrinsic | 210 |
| __monitor | 211 |
| __noreturn | 211 |

- using in pragma directives. 220
- `__no_init` 131, 211
- `__no_init`, using in pragma directives. 220
- `__raw` 212
 - using in pragma directives. 220
- `__root` 212
 - using in pragma directives. 221
- `__task` 212
 - using in pragma directives. 221
- `__trap` 27, 213
 - See also* INTVEC (segment)
- `__vector_call` 214

F

- `-f` (compiler option). 158
- fatal error messages 140
- `__FILE__` (predefined symbol). 249
- file dependencies, tracking 152
- file paths, specifying for `#include` files. 159
- filename
 - of object file. 166
 - specifying as parameter 145
- FLIST (segment). 281
- float (data type). 178
- floating-point constants
 - hexadecimal notation 193
 - hints 122
- floating-point format. 178
 - hints 121–122
 - implementation-defined behavior. 286
 - special cases. 179
 - 32-bits 178
 - 64-bits 178
- floating-point type, configuring size of double 8
- `float.h` (library header file) 257
- `float_t`, in `math.h`. 261
- for loops, declarations in. 190

- formats
 - floating-point values 178
 - standard IEEE (floating-point). 178
- `_formatted_write` (library function) 57
- `fpclassify`, in `math.h` 260
- `FP_INFINITE`, in `math.h` 260
- `FP_NAN`, in `math.h` 260
- `FP_NORMAL`, in `math.h` 260
- `FP_SUBNORMAL`, in `math.h` 260
- `FP_ZERO`, in `math.h`. 260
- fragmentation, of heap memory 24
- `free` (standard library function) 23
- `fstream` (library header file) 258
- `fstream.h` (library header file) 258
- `__func__` (language extension) 197
- `FUNCALL` (compiler function directive) 98
- `__FUNCTION__` (language extension) 197
- function directives. 98
- function inlining
 - compiler transformation. 120
 - disabling (`--no_inline`) 164
- function pointers 179
- function prototypes 127
- function template parameter deduction (extended EC++) . 110
- function type information, omitting in object output. . . . 167
- function vectors for non-interrupt functions 44
- `FUNCTION` (compiler function directive) 98
- function (pragma directive). 289
- functional (STL header file) 258
- functions 25
 - declaring 91
 - Embedded C++ and special function types 31
 - executing 14
 - inlining. 190
 - interrupt 26, 28
 - intrinsic 83, 126
 - monitor 28
 - omitting type info 167
 - parameters 93

| | |
|------------------------|-------|
| placing in segments | 49 |
| recursive | 126 |
| storing data on stack | 22–23 |
| reentrancy (DLIB) | 256 |
| related extensions | 25 |
| return values from | 95 |
| special function types | 26 |
| trap | 27 |

G

| | |
|-----------------------------------------------------|------|
| getenv (library function), configuring support for | 72 |
| getzone (library function), configuring support for | 74 |
| __get_imask_ccr (intrinsic function) | 235 |
| __get_imask_exr (intrinsic function) | 235 |
| __get_interrupt_state (intrinsic function) | 235 |
| glossary | xvii |
| guidelines, reading | xvii |

H

| | |
|--------------------------------------|----------|
| hardware, configuration for | 7 |
| hash_map (STL header file) | 258 |
| hash_set (STL header file) | 259 |
| hdrstop (pragma directive) | 289 |
| header files | |
| library | 255 |
| C | 257 |
| C++ | 257–258 |
| EC++ | 257 |
| special function registers | 130 |
| stdbool.h | 176, 257 |
| stddef.h | 177 |
| STL | 258 |
| using as templates | 130 |
| --header_context (compiler option) | 158 |
| heap | 23 |
| changing default size (command line) | 42 |
| changing default size (IDE) | 42 |

| | |
|----------------------|----------|
| size | 40–42 |
| storing data | 14 |
| HEAP (segment) | 273, 278 |
| hidden parameters | 94 |
| hints, optimization | 126 |
| HUGE_VALF, in math.h | 260 |
| HUGE_VALL, in math.h | 260 |

I

| | |
|----------------------------------------------|-----------|
| -I (compiler option) | 159 |
| IAR Systems Technical Support | 141 |
| __IAR_SYSTEMS_ICC__ (predefined symbol) | 249 |
| __ICCH8__ (predefined symbol) | 249 |
| identifiers, implementation-defined behavior | 284 |
| IEEE format, floating-point values | 178 |
| implementation-defined behavior | 283 |
| include_alias (pragma directive) | 219 |
| INFINITY, in math.h | 260 |
| inheritance, in Embedded C++ | 103 |
| initialization | |
| dynamic | 58–59, 63 |
| single-value | 196 |
| initializers, static | 195 |
| inline assembler | 85, 191 |
| <i>See also</i> assembler language interface | |
| inline functions | 190 |
| inline (pragma directive) | 219 |
| inlining of functions, in compiler | 120 |
| instantiate (pragma directive) | 289 |
| integer characteristics, adding | 261 |
| integers | 176 |
| casting | 180 |
| implementation-defined behavior | 286 |
| intptr_t | 182 |
| ptrdiff_t | 181 |
| size_t | 181 |
| uintptr_t | 182 |
| integral promotion | 129 |

- internal error 140
- __interrupt (extended keyword) 27, 210
 - using in pragma directives 226
- interrupt functions 26
 - placement in memory 44
- interrupt vector table 27–28
 - INTVEC segment 282
- interrupt vectors, specifying with pragma directive 226
- interrupts
 - disabling 211
 - disabling during function execution 28
 - processor state 22
 - specifying cycles for disabling 161
- interrupts and EC++ destructors, using 114
- __INTERRUPT_MODE__ (predefined symbol) 249
- interrupt_mode (compiler option) 159
- __interrupt_mode (runtime model attribute) 78
- __INTERRUPT_MODE_0__ (predefined symbol) 249
- __INTERRUPT_MODE_1__ (predefined symbol) 249
- __INTERRUPT_MODE_2__ (predefined symbol) 249
- __INTERRUPT_MODE_3__ (predefined symbol) 249
- intptr_t (integer type) 182
- __intrinsic (extended keyword) 210
- intrinsic functions 126
 - overview 83
 - summary 227
 - __and_ccr 229
 - __and_exr 229
 - __bcd_add_char 230
 - __bcd_add_short 230
 - __bcd_subtract_char 230
 - __bcd_subtract_short 230
 - __dadd 231
 - __disable_interrupt 231
 - __do_byte_eepmov 231
 - __do_word_eepmov 232
 - __dsub 232
 - __eepmov 233
 - __eepmovi 234
 - __enable_interrupt 235
 - __get_imask_ccr 235
 - __get_imask_exr 235
 - __get_interrupt_state 235
 - __mac 236
 - __MOVFPPE 238
 - __MOVTPPE 238
 - __no_operation 238
 - __or_ccr 238
 - __or_exr 238
 - __read_ccr 239
 - __read_exr 239
 - __rotlc 239
 - __rotll 239
 - __rotlw 239
 - __rotrc 239
 - __rotrl 240
 - __rotrw 239
 - __set_imask_ccr 240
 - __set_imask_exr 240
 - __set_interrupt_mask 240
 - __set_interrupt_state 241
 - __sleep 241
 - __TAS 241
 - __TRAPA 242
 - __write_ccr 242
 - __write_exr 243
 - __xor_ccr 243
 - __xor_exr 243
- intrinsics.h (header file) 229
- inttypes.h, added C functionality 260
- INTVEC (segment) 44, 282
- __int_size (runtime model attribute) 78
- invocation syntax 135
- iomaniip (library header file) 258
- iomaniip.h (library header file) 258
- ios (library header file) 258
- iosfwd (library header file) 258
- iostream (library header file) 258

| | |
|--------------------------------------------------|-----|
| iostream.h (library header file) | 258 |
| isblank, in ctype.h | 260 |
| isfinite, in math.h | 260 |
| isgreater, in math.h | 260 |
| isinf, in math.h | 260 |
| islessequal, in math.h | 260 |
| islessgreater, in math.h | 260 |
| isless, in math.h | 260 |
| isnan, in math.h | 260 |
| isnormal, in math.h | 260 |
| ISO/ANSI C | |
| C++ features excluded from EC++ | 103 |
| specifying strict usage | 171 |
| ISO/ANSI standard, compiler extensions | 187 |
| iso646.h (library header file) | 257 |
| istream (library header file) | 258 |
| isunordered, in math.h | 260 |
| iswblank, in wctype.h | 262 |
| iterator (STL header file) | 259 |

K

| | |
|----------------------------------------------|-----|
| keep_definition (pragma directive) | 289 |
| keywords, extended | 10 |

L

| | |
|---------------------------------------|---------|
| -l (compiler option) | 87, 159 |
| labels | 196 |
| __program_start | 64 |
| language extensions | |
| descriptions | 187 |
| Embedded C++ | 103 |
| enabling | 156 |
| language (pragma directive) | 220 |
| Large (code model) | 26 |
| libraries | 4 |
| runtime | 54 |
| standard template library | 258 |

| | |
|-------------------------------------------------------|----------------------|
| library calls, specifying on command line | 155 |
| library configuration file | |
| modifying | 62 |
| option for specifying | 155 |
| library documentation | 255 |
| library features, missing from Embedded C++ | 104 |
| library functions | 255 |
| choosing printf formatter | 57 |
| choosing scanf formatter | 58 |
| choosing sprintf formatter | 57 |
| choosing sscanf formatter | 58 |
| reference information | xix |
| remove | 70 |
| rename | 70 |
| summary | 257 |
| __close | 70 |
| __lseek | 70 |
| __open | 70 |
| __read | 70 |
| __write | 70 |
| library header files | 255 |
| library modules, creating | 160 |
| library object files | 255 |
| --library_module (compiler option) | 160 |
| limits.h (library header file) | 257 |
| __LINE__ (predefined symbol) | 249 |
| linkage, C and C++ | 91 |
| linker command files | |
| contents | 34 |
| customizing | 35, 37, 39–41, 43–44 |
| introduction | 34 |
| template | 35 |
| using the -Z command | 36 |
| viewing default | 40 |
| linking, from the command line | 5 |
| list (STL header file) | 259 |
| listing, generating | 159 |
| literals, compound | 191 |
| literature, recommended | xix |

__LITTLE_ENDIAN__ (predefined symbol) 249
llabs, in `stdlib.h` 261
lldiv, in `stdlib.h` 261
locale.h (library header file) 257
location (pragma directive) 46, 220
LOCFRAME (compiler function directive) 98
long double (data type) 178
long float (data type), synonym for `double` 195
loop overhead, reducing 165
Loop unrolling (compiler option) 119
loop unrolling, disabling 165
loop-invariant expressions 120
low-level processor operations 188, 227
__low_level_init, customizing 65
__lseek (library function) 70

M

__mac (intrinsic function) 236
mac instructions
 enabling from command line (`--enable_mac`) 157
macros, variadic 253
__MAC_ENABLED__ (predefined symbol) 250
malloc (standard library function) 23
map (STL header file) 259
math.h (library header file) 257
math.h, added C functionality 260
MATH_ERREXCEPT, in `math.h` 260
math_errhandling, in `math.h` 260
MATH_ERRNO, in `math.h` 260
__MAX_CYCLES_NO_INTERRUPT__
 (predefined symbol) 250
--max_cycles_no_interrupt (compiler option) 161
memory
 allocating in Embedded C++ 23
 dynamic 23
 heap 23
 non-initialized 130
 RAM, saving 126

 releasing in Embedded C++ 23
 stack 22
 saving 126
 static 14
 used by executing functions 14
 used by global or static variables 14
memory access methods
 data8 100
 data16 99
 data32 100
memory management, type-safe 103
memory types 15
 bitvar 16
 data8 16
 data16 17
 data32 17
 Embedded C++ 21
 hints 122
 placing variables in 21
 pointers 19
 specifying 18
 structures 20
 summary 18
memory (pragma directive) 289
memory (STL header file) 259
message (pragma directive) 220
--migration_preprocessor_extensions (compiler option) . . 161
--misrac (compiler option) 162
--misrac_verbose (compiler option) 162
__MODEL_XXX__ (predefined symbol) 250
module consistency 77
 rtmodel 225
module name, specifying 163
--module_name (compiler option) 163
module_name (pragma directive) 289
__monitor (extended keyword) 211
monitor functions 28
__MOVFPPE (intrinsic function) 238
MOVFPPE (assembler instruction) 238

| | |
|-----------------------------------------------------------|-----|
| __MOVTPE (intrinsic function) | 238 |
| MOVTPE (assembler instruction) | 238 |
| multibyte character support | 157 |
| multiple inheritance, missing from Embedded C++ | 103 |
| mutable attribute, in Extended EC++ | 113 |

N

| | |
|------------------------------------------------------|----------|
| namespace support | |
| in Extended EC++ | 104, 113 |
| missing from Embedded C++ | 104 |
| NAN, in math.h | 260 |
| NDEBUG (preprocessor extension) | 252 |
| new operator (extended EC++) | 109 |
| new (keyword) | 23 |
| new (library header file) | 258 |
| new.h (library header file) | 258 |
| none (pragma directive) | 289 |
| non-initialized variables | 131 |
| non-interrupt functions and vectors | |
| placement in memory | 44 |
| non-interrupt functions in vectors | 44 |
| non-scalar parameters | 126 |
| NOP (assembler instruction) | 238 |
| __noreturn (extended keyword) | 211 |
| using in pragma directives | 220 |
| NOTREACHED (pragma directive) | 289 |
| --no_code_motion (compiler option) | 163 |
| --no_cse (compiler option) | 163 |
| __no_init (extended keyword) | 131, 211 |
| using in pragma directives | 220 |
| --no_inline (compiler option) | 164 |
| __no_operation (intrinsic function) | 238 |
| --no_path_in_file_macros (compiler option) | 164 |
| no_pch (pragma directive) | 289 |
| --no_unroll (compiler option) | 165 |
| --no_warnings (compiler option) | 166 |
| --no_wrap_diagnostics (compiler option) | 166 |
| numeric (STL header file) | 259 |

O

| | |
|------------------------------------------------------------|----------|
| -o (compiler option) | 166 |
| object attributes | 202 |
| object filename, specifying | 166 |
| object module name, specifying | 163 |
| object_attribute (pragma directive) | 131, 220 |
| --omit_types (compiler option) | 167 |
| once (pragma directive) | 290 |
| --only_stdout (compiler option) | 167 |
| __open (library function) | 70 |
| operating mode, configuration for | 6 |
| __OPERATING_MODE__ (predefined symbol) | 250 |
| --operating_mode (compiler option) | 167 |
| __operating_mode (runtime model attribute) | 79 |
| __OPERATING_MODE_ADVANCED__ | |
| (predefined symbol) | 250 |
| __OPERATING_MODE_NORMAL__ | |
| (predefined symbol) | 250 |
| operators | |
| @ | 189 |
| __ALIGNOF__ | 189 |
| __memory_of | 107 |
| __segment_begin | 189 |
| __segment_end | 189 |
| optimization | |
| code motion, disabling (--no_code_motion) | 163 |
| common subexpression elimination, | |
| disabling (--no_cse) | 163 |
| configuration | 8 |
| function inlining, disabling (--no_inline) | 164 |
| hints | 126 |
| loop unrolling, disabling (--no_unroll) | 165 |
| size, specifying | 173 |
| speed, specifying | 170 |
| types and levels | 118 |
| type-based alias analysis, disabling (--no_tbaa) | 164 |
| optimization techniques | 119 |
| optimize (pragma directive) | 221 |

options summary, compiler 146
 __or_ccr (intrinsic function) 238
 __or_exr (intrinsic function) 238
 ostream (library header file) 258
 output
 specifying 5
 specifying file name 5
 output files, from XLINK 5
 output, preprocessor 168

P

pack (pragma directive) 183, 222
 packed structure types 183
 parameters
 function 93
 hidden 94
 non-scalar 126
 register 93
 rules for specifying a file or directory 145
 specifying 145
 stack 93, 95
 typographic convention xx
 placement of code and data 263
 pointer types
 casting between 19
 mixing, language extensions 195
 pointers
 casting 180
 data 180
 function 179
 implementation-defined behavior 287
 using instead of large non-scalar parameters 126
 polymorphism, in Embedded C++ 103
 porting of code, containing pragma directives 216
 _Pragma (preprocessor extension) 253
 pragma directives 215
 alignment 289
 ARGSUSED 289
 baseaddr 289
 basic_template_matching 216
 basic_template_matching,using 111
 bitfields 177, 217
 can_instantiate (pragma directive) 289
 codeseg (pragma directive) 289
 constseg 217
 cspy_support 289
 dataseg 218
 data_alignment 217
 define_type_info 289
 diag_default 218
 diag_error 218
 diag_remark 218
 diag_suppress 218
 diag_warning 219
 do_not_instantiate 289
 function 289
 hdrstop 289
 include_alias 219
 inline 219
 instantiate 289
 keep_definition 289
 language 220
 location 46, 220
 memory 289
 message 220
 module_name 289
 none 289
 NOTREACHED 289
 no_pch 289
 object_attribute 131, 220
 once 290
 optimize 221
 overview 10
 pack 183, 222
 public_equ 290
 required 224
 rtmodel 225

| | | | |
|------------------------------|---------|--------------------------------------------------|------|
| segment | 225 | __OPERATING_MODE_NORMAL__ | 250 |
| syntax | 216 | __STACK_POINTER_SIZE__ | 250 |
| system_include | 290 | __STACK_POINTER_SIZE_8__ | 250 |
| type_attribute | 226 | __STACK_POINTER_SIZE_16__ | 250 |
| VARARGS | 290 | __STACK_POINTER_SIZE_32__ | 250 |
| vector | 27, 226 | __STDC_VERSION__ | 251 |
| warnings | 290 | __STDC__ | 251 |
| __printf_args | 290 | __SUBVERSION__ | 251 |
| __scanf_args | 290 | __TID__ | 251 |
| predefined symbols | 246 | __TIME__ | 252 |
| overview | 10 | __VER__ | 252 |
| __BASE_FILE__ | 247 | --preinclude (compiler option) | 168 |
| __BUILD_NUMBER__ | 247 | --preprocess (compiler option) | 168 |
| __CODE_COMPATIBILITY_CHECK__ | 247 | preprocessing directives, implementation-defined | |
| __CODE_MODEL__ | 248 | behavior | 288 |
| __CORE__ | 248 | preprocessor extensions | |
| __CORE_H8S__ | 248 | NDEBUG | 252 |
| __CORE_H8300H__ | 248 | #warning message | 253 |
| __cplusplus | 248 | _Pragma | 253 |
| __DATA_MODEL__ | 248 | __VA_ARGS__ | 253 |
| __DATE__ | 248 | preprocessor output | 168 |
| __DIRECT_LIBRARY_CALLS__ | 248 | preprocessor symbols | 246 |
| __DOUBLE_SIZE__ | 249 | defining | 150 |
| __embedded_cplusplus | 249 | preprocessor, extending | 161 |
| __FILE__ | 249 | prerequisites (programming experience) | xvii |
| __IAR_SYSTEMS_ICC__ | 249 | preserved registers | 92 |
| __ICCH8__ | 249 | __PRETTY_FUNCTION__ (language extension) | 197 |
| __INTERRUPT_MODE__ | 249 | print formatter, selecting | 58 |
| __INTERRUPT_MODE_0__ | 249 | printf (library function) | 57 |
| __INTERRUPT_MODE_1__ | 249 | choosing formatter | 57 |
| __INTERRUPT_MODE_2__ | 249 | processor operations, low-level | 227 |
| __INTERRUPT_MODE_3__ | 249 | programming experience, required | xvii |
| __LINE__ | 249 | programming hints | 126 |
| __LITTLE_ENDIAN__ | 249 | __program_start (label) | 64 |
| __MAC_ENABLED__ | 250 | ptrdiff_t (integer type) | 181 |
| __MAX_CYCLES_NO_INTERRUPT__ | 250 | PUBLIC (assembler directive) | 169 |
| __MODEL_XXX__ | 250 | --public_equ (compiler option) | 168 |
| __OPERATING_MODE__ | 250 | public_equ (pragma directive) | 290 |
| __OPERATING_MODE_ADVANCED__ | 250 | | |

Q

QCCH8 (environment variable) 136

qualifiers

- const 184
- implementation-defined behavior. 288
- volatile 183

queue (STL header file) 259

R

-r (compiler option). 169

raise (library function), configuring support for 73

RAM memory, saving. 126

__raw (extended keyword) 212

- using in pragma directives 220

__read (library function). 70

read formatter, selecting 59

reading guidelines xvii

reading, recommended xix

__read_ccr (intrinsic function) 239

__read_exr (intrinsic function) 239

realloc (standard library function). 23

recursive functions 126

- storing data on stack 22–23

reentrancy (DLIB). 256

reference information, typographic convention. xx

register parameters 93

registered trademarks ii

registers

- assigning to parameters 94
- callee-save, stored on stack 22
- implementation-defined behavior. 287
- preserved 92
- scratch 92

reinterpret_cast (cast operator) 104

remark (diagnostic message)

- classifying 153
- enabling 169

- remarks (compiler option) 169

remarks (diagnostic message). 140

remove (library function) 70

rename (library function) 70

required (pragma directive). 224

--require_prototypes (compiler option). 169

return values, from functions 95

__root (extended keyword) 212

- using in pragma directives 221

__rotlc (intrinsic function) 239

__rotll (intrinsic function). 239

__rotlw (intrinsic function). 239

__rotrc (intrinsic function) 239

__rotrl (intrinsic function). 240

__rotrw (intrinsic function). 239

routines, time-critical 83, 188, 227

RTE (assembler instruction) 213

RTMODEL (assembler directive) 78

rtmodel (pragma directive) 225

rtti support, missing from STL 104

__rt_version (runtime model attribute) 79

runtime libraries 54

- introduction 255
- naming convention. 55
- summary 55

runtime model attributes 77

- __bus_width. 78
- __code_model 78
- __data_model. 78
- __double_size 78
- __interrupt_mode. 78
- __int_size. 78
- __operating_mode 79
- __rt_version. 79
- __use_h8s_instr 79
- __use_mac_instr 79

runtime type information, missing from Embedded C++ . 103

| | |
|----------------------------------------------------------|---------------|
| S | |
| -s (compiler option) | 170 |
| scanf (library function), choosing formatter | 58 |
| scratch registers | 92 |
| segment memory types, in XLINK | 34 |
| segment operators | 189 |
| segment (pragma directive) | 225 |
| segments | 263 |
| BITVARS | 265 |
| CHECKSUM | 266 |
| CODE16 | 267 |
| CODE24 | 267 |
| CSTACK, example | 40 |
| DATA8_AC | 268 |
| DATA8_AN | 269 |
| DATA8_I | 269 |
| DATA8_ID | 270 |
| DATA8_N | 271 |
| DATA8_Z | 271 |
| DATA16_AC | 272 |
| DATA16_AN | 272 |
| DATA16_C | 273 |
| DATA16_I | 274 |
| DATA16_ID | 270, 275, 279 |
| DATA16_N | 271, 275, 280 |
| DATA16_Z | 276 |
| DATA32_AC | 276 |
| DATA32_AN | 277 |
| DATA32_C | 277 |
| DATA32_HEAP | 278 |
| DATA32_I | 278 |
| DATA32_ID | 279 |
| DATA32_N | 280 |
| DATA32_Z | 280 |
| DIFUNCT | 45, 264, 281 |
| FLIST | 281 |
| HEAP | 273, 278 |
| introduction | 33 |
| INTVEC | 44, 282 |
| summary | 263 |
| __segment_begin (extended operator) | 189 |
| __segment_end (extended operator) | 189 |
| semaphores (__monitor) | 211 |
| set (STL header file) | 259 |
| setjmp.h (library header file) | 257 |
| settings, basic for project configuration | 5 |
| __set_imask_ccr (intrinsic function) | 240 |
| __set_imask_exr (intrinsic function) | 240 |
| __set_interrupt_mask (intrinsic function) | 240 |
| __set_interrupt_state (intrinsic function) | 241 |
| severity level, of diagnostic messages | 140 |
| specifying | 140 |
| SFR (special function registers) | 130 |
| declaring extern | 46 |
| shared object | 138 |
| signal (library function), configuring support for | 73 |
| signal.h (library header file) | 257 |
| signbit, in math.h | 260 |
| signed char (data type) | 176–177 |
| specifying | 149 |
| signed int (data type) | 176 |
| signed long long (data type) | 176 |
| signed long (data type) | 176 |
| signed short (data type) | 176 |
| --silent (compiler option) | 170 |
| silent operation, specifying | 170 |
| 64-bits (floating-point format) | 178 |
| size optimization, specifying | 173 |
| size_t (integer type) | 181 |
| skeleton code, creating for assembler language interface | 86 |
| __sleep (intrinsic function) | 241 |
| SLEEP (assembler instruction) | 241 |
| slist (STL header file) | 259 |
| Small (code model) | 26 |
| snprintf, in stdio.h | 261 |
| source files, list all referred | 158 |
| special function registers (SFR) | 130 |

- special function types 26
 - overview 10
- speed optimization, specifying 170
- sprintf (library function) 57
 - choosing formatter 57
- sscanf (library function), choosing formatter 58
- sstream (library header file) 258
- stack 22
 - advantages and problems using 22
 - changing default size (from command line) 40
 - changing default size (in Embedded Workbench) 40
 - contents of 22
 - function usage 14
 - internal data 268
 - saving space 126
 - size 41
- stack parameters 93, 95
- stack pointer 22
 - configuring for arithmetics 127
- stack (STL header file) 259
- __STACK_POINTER_SIZE__ (predefined symbol) 250
- stack_pointer_size (compiler option) 171
- __STACK_POINTER_SIZE_8__ (predefined symbol) 250
- __STACK_POINTER_SIZE_16__ (predefined symbol) 250
- __STACK_POINTER_SIZE_32__ (predefined symbol) 250
- standard error 167
- standard output, specifying 167
- standard template library (STL)
 - in Extended EC++ 104, 112, 258
 - missing from Embedded C++ 104
- startup, system 64
- statements, implementation-defined behavior 288
- static memory 14
- static overlay 97
- static_cast (cast operator) 104
- std namespace, missing from EC++
 - and Extended EC++ 113
- stdarg.h (library header file) 257
- stdbool.h (library header file) 176, 257
 - added C functionality 261
- __STDC__ (predefined symbol) 251
- __STDC_VERSION__ (predefined symbol) 251
- stddef.h (library header file) 177, 257
- stderr 70, 167
- stdexcept (library header file) 258
- stdin 70
- stdint.h, added C functionality 261
- stdio.h (library header file) 257
- stdio.h, additional C functionality 261
- stdlib.h (library header file) 257
- stdlib.h, additional C functionality 261
- stdout 70, 167
- STL 112
- streambuf (library header file) 258
- streams, supported in Embedded C++ 104
 - strict_ansi (compiler option) 171
- string (library header file) 258
- strings, supported in Embedded C++ 104
- string.h (library header file) 257
- strstream (library header file) 258
- strtod (library function), configuring support for 74
- strtod, in stdlib.h 261
- strtof, in stdlib.h 261
- strtold, in stdlib.h 261
- strtoll, in stdlib.h 261
- strtoull, in stdlib.h 261
- structs 192
 - anonymous 189
- structure types
 - alignment 182–183
 - layout 182
 - packed 183
- structures
 - anonymous 124
 - implementation-defined behavior 287
 - placing in memory type 20
- __SUBVERSION__ (predefined symbol) 251
- support, technical 141

| | |
|----------------------------------------------------|-----|
| symbols | |
| anonymous, creating | 191 |
| overview of predefined | 10 |
| preprocessor, defining | 150 |
| syntax | |
| compiler options | 143 |
| extended keywords | 199 |
| system startup | 64 |
| system termination | 64 |
| system (library function), configuring support for | 72 |
| system_include (pragma directive) | 290 |

T

| | |
|-----------------------------------------------------------|--------------|
| __TAS (intrinsic function) | 241 |
| TAS (assembler instruction) | 241 |
| __task (extended keyword) | 212 |
| using in pragma directives | 221 |
| technical support, IAR Systems | 141 |
| template support | |
| in Extended EC++ | 104, 110 |
| missing from Embedded C++ | 103 |
| termination, system | 64 |
| terminology | xvii, xx |
| 32-bits (floating-point format) | 178 |
| this pointer, referring to a class object (extended EC++) | 105 |
| this (pointer) | 88 |
| __TID__ (predefined symbol) | 251 |
| __TIME__ (predefined symbol) | 252 |
| time (library function), configuring support for | 74 |
| time-critical routines | 83, 188, 227 |
| time.h (library header file) | 257 |
| tips, programming | 126 |
| trademarks | ii |
| translation, implementation-defined behavior | 283 |
| __trap (extended keyword) | 27, 213 |
| trap functions | 27 |
| trap vectors, specifying with pragma directive | 226 |
| __TRAPA (intrinsic function) | 242 |

| | |
|------------------------------------------------------|---------|
| TRAPA (assembler instruction) | 27, 242 |
| type attributes | 199 |
| type definitions, used for specifying memory storage | 201 |
| type information, omitting | 167 |
| type qualifiers, const and volatile | 183 |
| typedefs, repeated declarations | 195 |
| type-based alias analysis | |
| compiler transformation | 120 |
| disabling (--no_tbaa) | 164 |
| type-safe memory management | 103 |
| type_attribute (pragma directive) | 226 |
| typographic conventions | xx |

U

| | |
|-------------------------------------------|----------|
| uintptr_t (integer type) | 182 |
| unions | |
| anonymous | 124, 189 |
| implementation-defined behavior | 287 |
| unsigned char (data type) | 176–177 |
| changing to signed char | 149 |
| unsigned int (data type) | 176 |
| unsigned long long (data type) | 176 |
| unsigned long (data type) | 176 |
| unsigned short (data type) | 176 |
| __use_h8s_instr (runtime model attribute) | 79 |
| __use_mac_instr (runtime model attribute) | 79 |
| utility (STL header file) | 259 |

V

| | |
|------------------------------------------------------|-----|
| VARARGS (pragma directive) | 290 |
| variable type information, omitting in object output | 167 |
| variables | |
| auto | 22 |
| defined inside a function | 22 |
| global, placement in memory | 14 |
| local. <i>See</i> auto variables | |
| non-initialized | 131 |

omitting type info 167
 placing at absolute addresses 46
 placing in named segments 48
 static, placement in memory 14
 vector (pragma directive) 27, 226
 vector (STL header file) 259
 __vector_call (extended keyword) 214
 __VER__ (predefined symbol) 252
 version, of compiler 251–252
 vfprintf, in stdio.h 261
 vfwscanf, in wchar.h 261
 void, pointers to 195
 volatile, declaring objects 183
 vscanf, in stdio.h 261
 vsnprintf, in stdio.h 261
 vsscanf, in stdio.h 261
 vswscanf, in wchar.h 261
 vwscanf, in wchar.h 261

W

#warning message (preprocessor extension) 253
 warnings 140
 classifying 154
 disabling 166
 exit code 172
 warnings (pragma directive) 290
 --warnings_affect_exit_code (compiler option) 139
 --warnings_are_errors (compiler option) 172
 wchar.h (library header file) 257
 wchar.h, added C functionality 261
 wchar_t (data type), adding support for in C 177
 wcstof, in wchar.h 261
 wcstolb, in wchar.h 261
 wctype.h (library header file) 257
 wctype.h, added C functionality 262
 --weak_rtmodel_check (compiler option) 172
 __write (library function) 70
 __write_cxr (intrinsic function) 242

__write_cxr (intrinsic function) 243

X

XLINK output files 5
 XLINK segment memory types 34
 __xor_cxr (intrinsic function) 243
 __xor_cxr (intrinsic function) 243

Z

-z (compiler option) 173

Symbols

#include files, specifying 136, 159
 #warning message (preprocessor extension) 253
 -D (compiler option) 150
 -e (compiler option) 156
 -f (compiler option) 158
 -I (compiler option) 159
 -l (compiler option) 87, 159
 -o (compiler option) 166
 -r (compiler option) 169
 -s (compiler option) 170
 -z (compiler option) 173
 --bus_width (compiler option) 148
 --char_is_signed (compiler option) 149
 --code_model (compiler option) 149
 --core (compiler option) 150
 --data_model (compiler option) 151
 --debug (compiler option) 151
 --dependencies (compiler option) 152
 --diagnostics_tables (compiler option) 154
 --diag_error (compiler option) 153
 --diag_remark (compiler option) 153
 --diag_suppress (compiler option) 153
 --diag_warning (compiler option) 154
 --direct_library_calls (compiler option) 155

| | | | |
|-----------------------------------------------------------------|----------|-----------------------------------------------------|-----|
| --dlib_config (compiler option) | 155 | _exit (exit function) | 64 |
| --double (compiler option) | 155 | _Exit, in stdlib.h | 261 |
| --ec++ (compiler option) | 156 | _formatted_write (library function) | 57 |
| --eec++ (compiler option) | 157 | _Pragma (preprocessor extension) | 253 |
| --enable_mac (compiler option) | 157 | __ALIGNOF__ (operator) | 189 |
| --enable_multibytes (compiler option) | 157 | __and_ccr (intrinsic function) | 229 |
| --error_limit (compiler option) | 158 | __and_exr (intrinsic function) | 229 |
| --header_context (compiler option) | 158 | __asm (language extension) | 191 |
| --interrupt_mode (compiler option) | 159 | __BASE_FILE__ (predefined symbol) | 247 |
| --library_module (compiler option) | 160 | __bcd_add_char (intrinsic function) | 230 |
| --max_cycles_no_interrupt (compiler option) | 161 | __bcd_add_short (intrinsic function) | 230 |
| --migration_preprocessor_extensions (compiler option) | 161 | __bcd_subtract_char (intrinsic function) | 230 |
| --misrac (compiler option) | 162 | __bcd_subtract_short (intrinsic function) | 230 |
| --misrac_verbose (compiler option) | 162 | __bitvar (extended keyword) | 204 |
| --module_name (compiler option) | 163 | __BUILD_NUMBER__ (predefined symbol) | 247 |
| --no_code_motion (compiler option) | 163 | __bus_width (runtime model attribute) | 78 |
| --no_cse (compiler option) | 163 | __cc_version1 (extended keyword) | 204 |
| --no_inline (compiler option) | 164 | __cc_version2 (extended keyword) | 205 |
| --no_path_in_file_macros (compiler option) | 164 | __cc_version3 (extended keyword) | 205 |
| --no_tbaa (compiler option) | 164 | __close (library function) | 70 |
| --no_typedefs_in_diagnostics (compiler option) | 165 | __code_model (runtime model attribute) | 78 |
| --no_unroll (compiler option) | 165 | __CODE_MODEL__ (predefined symbol) | 248 |
| --no_warnings (compiler option) | 166 | __code16 (extended keyword) | 206 |
| --no_wrap_diagnostics (compiler option) | 166 | __code16 (function pointer) | 179 |
| --omit_types (compiler option) | 167 | __code24 (extended keyword) | 206 |
| --only_stdout (compiler option) | 167 | __code24 (function pointer) | 179 |
| --operating_mode (compiler option) | 167 | __CORE__ (predefined symbol) | 248 |
| --preinclude (compiler option) | 168 | __CORE_H8S__ (predefined symbol) | 248 |
| --preprocess (compiler option) | 168 | __CORE_H8300H__ (predefined symbol) | 248 |
| --remarks (compiler option) | 169 | __cplusplus (predefined symbol) | 248 |
| --require_prototypes (compiler option) | 169 | __dadd (intrinsic function) | 231 |
| --silent (compiler option) | 170 | __data_model (runtime model attribute) | 78 |
| --stack_pointer_size (compiler option) | 171 | __DATA_MODEL__ (predefined symbol) | 248 |
| --strict_ansi (compiler option) | 171 | __data8 (data pointer) | 180 |
| --warnings_affect_exit_code (compiler option) | 139, 172 | __data8 (extended keyword) | 207 |
| --warnings_are_errors (compiler option) | 172 | __data16 (data pointer) | 180 |
| --weak_rtmodel_check (compiler option) | 172 | __data16 (extended keyword) | 208 |
| @ (operator) | 189 | __data32 (data pointer) | 180 |
| _Exit (exit function) | 64 | __data32 (extended keyword) | 209 |

- `__DATE__` (predefined symbol) 248
- `__DIRECT_LIBRARY_CALLS__` (predefined symbol) . 248
- `__disable_interrupt` (intrinsic function) 231
- `__double_size` (runtime model attribute) 78
- `__DOUBLE_SIZE__` (predefined symbol) 249
- `__do_byte_eepmov` (intrinsic function) 231
- `__do_word_eepmov` (intrinsic function) 232
- `__dsub` (intrinsic function) 232
- `__eepmov` (intrinsic function) 233
- `__eepmovi` (intrinsic function) 234
- `__embedded_cplusplus` (predefined symbol) 249
- `__enable_interrupt` (intrinsic function) 235
- `__exit` (exit function) 64
- `__FILE__` (predefined symbol) 249
- `__FUNCTION__` (language extension) 197
- `__func__` (language extension) 197
- `__gets`, in `stdio.h` 261
- `__get_imask_ccr` (intrinsic function) 235
- `__get_imask_exr` (intrinsic function) 235
- `__get_interrupt_state` (intrinsic function) 235
- `__IAR_SYSTEMS_ICC__` (predefined symbol) 249
- `__ICCH8__` (predefined symbol) 249
- `__interrupt` (extended keyword) 27, 210
 - using in pragma directives 226
- `__interrupt_mode` (runtime model attribute) 78
- `__INTERRUPT_MODE__` (predefined symbol) 249
- `__INTERRUPT_MODE_0__` (predefined symbol) 249
- `__INTERRUPT_MODE_1__` (predefined symbol) 249
- `__INTERRUPT_MODE_2__` (predefined symbol) 249
- `__INTERRUPT_MODE_3__` (predefined symbol) 249
- `__intrinsic` (extended keyword) 210
- `__int_size` (runtime model attribute) 78
- `__LINE__` (predefined symbol) 249
- `__LITTLE_ENDIAN__` (predefined symbol) 249
- `__low_level_init`, customizing 65
- `__lseek` (library function) 70
- `__mac` (intrinsic function) 236
- `__MAC_ENABLED__` (predefined symbol) 250
- `__memory_of`, operator 107
- `__MODEL_XXX__` (predefined symbol) 250
- `__monitor` (extended keyword) 211
- `__MOVFPPE` (intrinsic function) 238
- `__MOVTPE` (intrinsic function) 238
- `__noreturn` (extended keyword) 211
 - using in pragma directives 220
- `__no_init` (extended keyword) 131, 211
 - using in pragma directives 220
- `__no_operation` (intrinsic function) 238
- `__open` (library function) 70
- `__operating_mode` (runtime model attribute) 79
- `__OPERATING_MODE__` (predefined symbol) 250
- `__OPERATING_MODE_ADVANCED__`
 (predefined symbol) 250
- `__OPERATING_MODE_NORMAL__`
 (predefined symbol) 250
- `__or_ccr` (intrinsic function) 238
- `__or_exr` (intrinsic function) 238
- `__PRETTY_FUNCTION__` (language extension) 197
- `__printf_args` (pragma directive) 290
- `__program_start` (label) 64
- `__qsorttbl`, in `stdlib.h` 261
- `__raw` (extended keyword) 212
 - using in pragma directives 220
- `__read` (library function) 70
- `__read_ccr` (intrinsic function) 239
- `__read_exr` (intrinsic function) 239
- `__root` (extended keyword) 212
 - using in pragma directives 221
- `__rotlc` (intrinsic function) 239
- `__rotll` (intrinsic function) 239
- `__rotlw` (intrinsic function) 239
- `__rotrc` (intrinsic function) 239
- `__rotrl` (intrinsic function) 240
- `__rotrw` (intrinsic function) 239
- `__rt_version` (runtime model attribute) 79
- `__scanf_args` (pragma directive) 290
- `__segment_begin` (extended operators) 189
- `__segment_end` (extended operators) 189
- `__set_imask_ccr` (intrinsic function) 240

| | |
|---------------------------------------------------------|---------|
| __set_imask_exr (intrinsic function) | 240 |
| __set_interrupt_mask (intrinsic function) | 240 |
| __set_interrupt_state (intrinsic function) | 241 |
| __sleep (intrinsic function) | 241 |
| __STACK_POINTER_SIZE__ (predefined symbol) | 250 |
| __STACK_POINTER_SIZE_8__ (predefined symbol) | 250 |
| __STACK_POINTER_SIZE_16__ (predefined symbol) | 250 |
| __STACK_POINTER_SIZE_32__ (predefined symbol) | 250 |
| __STDC_VERSION__ (predefined symbol) | 251 |
| __STDC__ (predefined symbol) | 251 |
| __SUBVERSION__ (predefined symbol) | 251 |
| __TAS (intrinsic function) | 241 |
| __task (extended keyword) | 212 |
| using in pragma directives | 221 |
| __TID__ (predefined symbol) | 251 |
| __TIME__ (predefined symbol) | 252 |
| __trap (extended keyword) | 27, 213 |
| __TRAPA (intrinsic function) | 242 |
| __ungetchar, in stdio.h | 261 |
| __use_h8s_instr (runtime model attribute) | 79 |
| __use_mac_instr (runtime model attribute) | 79 |
| __VA_ARGS__ (preprocessor extension) | 253 |
| __vector_call (extended keyword) | 214 |
| __VER__ (predefined symbol) | 252 |
| __write (library function) | 70 |
| __write_array, in stdio.h | 261 |
| __write_ccr (intrinsic function) | 242 |
| __write_exr (intrinsic function) | 243 |
| __xor_ccr (intrinsic function) | 243 |
| __xor_exr (intrinsic function) | 243 |

Numerics

| | |
|-------------------------------------------|-----|
| 32-bits (floating-point format) | 178 |
| 64-bits (floating-point format) | 178 |