IAR C/C++ Compiler

Reference Guide

for Renesas

M16C/1X–3X, 5X–6X and R8C

Series of CPU cores

CM16C-4

IAR SYSTEMS

COPYRIGHT NOTICE

Copyright © 1995–2010 IAR Systems AB.

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

DISCLAIMER

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

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

TRADEMARKS

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

Microsoft and Windows are registered trademarks of Microsoft Corporation.

Renesas is a registered trademark of Renesas Electronics Corporation. M16C/R8C Series is a trademark of Renesas Electronics Corporation.

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

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

EDITION NOTICE

Fourth edition: June 2010

Part number: CM16C-4

This guide applies to version 3.5 of IAR Embedded Workbench® for Renesas M16C/R8C Series of CPU cores.

Internal reference: R8, 5.5, IJOA.

Brief contents

Contents

Tables

Preface

Welcome to the IAR C/C++ Compiler Reference Guide for M16C/R8C. The purpose of this guide is to provide you with detailed reference information that can help you to use the compiler to best suit your application requirements. This guide also gives you suggestions on coding techniques so that you can develop applications with maximum efficiency.

Who should read this guide

Read this guide if you plan to develop an application using the C or C++ language for the M16C/R8C Series CPU core and need detailed reference information on how to use the compiler. You should have working knowledge of:

- The architecture and instruction set of the M16C/R8C Series of CPU cores. Refer to the documentation from Renesas for information about the M16C/R8C Series of CPU cores
- The C or C++ programming language
- Application development for embedded systems
- The operating system of your host computer.

How to use this guide

When you start using the IAR C/C++ Compiler for M16C/R8C, you should read *Part 1. Using the compiler* in this guide.

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

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

What this guide contains

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

Part 1. Using the compiler

- *Getting started* gives the information you need to get started using the compiler for efficiently developing your application.
- *Data storage* describes how to store data in memory, focusing on the different data models and data memory type attributes.
- *Functions* gives a brief overview of function-related extensions—mechanisms for controlling functions—and describes some of these mechanisms in more detail.
- *Placing code and data* describes the concept of segments, introduces the linker command file, and describes how code and data are placed in memory.
- *The DLIB runtime environment* describes the DLIB runtime environment in which an application executes. It covers how you can modify it by setting options, overriding default library modules, or building your own library. The chapter also describes system initialization introducing the file `cstartup`, how to use modules for locale, and file I/O.
- *The CLIB runtime environment* gives an overview of the CLIB runtime libraries and how to customize them. The chapter also describes system initialization and introduces the file `cstartup`.
- *Assembler language interface* contains information required when parts of an application are written in assembler language. This includes the calling convention.
- *Using C++* gives an overview of the two levels of C++ support: The industry-standard EC++ and IAR Extended EC++.
- *Efficient coding for embedded applications* gives hints about how to write code that compiles to efficient code for an embedded application.

Part 2. Reference information

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

- *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 M16C/R8C-specific keywords that are extensions to the standard C/C++ language.
- *Pragma directives* gives reference information about the pragma directives.
- *Intrinsic functions* gives reference information about functions to use for accessing M16C/R8C-specific low-level features.
- *The preprocessor* gives a brief overview of the preprocessor, including reference information about the different preprocessor directives, symbols, and other related information.
- *Library functions* gives an introduction to the C or C++ library functions, and summarizes the header files.
- *Segment reference* gives reference information about the compiler's use of segments.
- *Implementation-defined behavior* describes how the compiler handles the implementation-defined areas of the C language standard.

Other documentation

The complete set of IAR Systems development tools for the M16C/R8C Series CPU core is described in a series of guides. For information about:

- Using the IDE and the IAR C-SPY Debugger®, refer to the *IAR Embedded Workbench® IDE User Guide*
- Programming for the M16C/R8C IAR Assembler, refer to the *M16C/R8C IAR Assembler Reference Guide*
- Using the IAR XLINK Linker, the IAR XAR Library Builder, and the IAR XLIB Librarian, refer to the *IAR Linker and Library Tools Reference Guide*
- Using the IAR DLIB Library functions, refer to the online help system
- Using the IAR CLIB Library functions, refer to the *IAR C Library Functions Reference Guide*, available from the online help system.
- Porting application code and projects created with a previous IAR Embedded Workbench for M16C/R8C, refer to the *M16C/R8C IAR Embedded Workbench® Migration Guide*
- Using the MISRA-C:1998 rules or the MISRA-C:2004 rules, refer to the *IAR Embedded Workbench® MISRA C:1998 Reference Guide* or the *IAR Embedded Workbench® MISRA C:2004 Reference Guide*, respectively.

All of these guides are delivered in hypertext PDF or HTML format on the installation media.

FURTHER READING

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

- Barr, Michael, and Andy Oram, ed. *Programming Embedded Systems in C and C++*. O'Reilly & Associates.
- Harbison, Samuel P. and Guy L. Steele (contributor). *C: A Reference Manual.* Prentice Hall.
- Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language. Prentice Hall*. [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 these web sites:

- The Renesas web site, **www.renesas.com**, contains information and news about the M16C/R8C Series of CPU cores.
- The IAR Systems web site, **www.iar.com**, holds application notes and other product information.
- Finally, the Embedded C++ Technical Committee web site, **www.caravan.net/ec2plus**, contains information about the Embedded C++ standard.

Document conventions

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

When referring to a directory in your product installation, for example `m16c\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench 6.`*`n`*`\m16c\doc`.

TYPOGRAPHIC CONVENTIONS

This guide uses the following typographic conventions:

Style	Used for
`computer`	• Source code examples and file paths. • Text on the command line. • Binary, hexadecimal, and octal numbers.
`parameter`	A placeholder for an actual value used as a parameter, for example *`filename`*`.h` where *`filename`* represents the name of the file.
`[option]`	An optional part of a command.
`[a|b|c]`	An optional part of a command with alternatives.
`{a|b|c}`	A mandatory part of a command with alternatives.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
italic	• A cross-reference within this guide or to another guide. • Emphasis.
…	An ellipsis indicates that the previous item can be repeated an arbitrary number of times.
🔧	Identifies instructions specific to the IAR Embedded Workbench® IDE interface.
🖥	Identifies instructions specific to the command line interface.
💡	Identifies helpful tips and programming hints.
⚠	Identifies warnings.

Table 1: Typographic conventions used in this guide

NAMING CONVENTIONS

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

Brand name	Generic term
IAR Embedded Workbench® for M16C/R8C	IAR Embedded Workbench®
IAR Embedded Workbench® IDE for M16C/R8C	the IDE
IAR C-SPY® Debugger for M16C/R8C	C-SPY, the debugger
IAR C-SPY® Simulator	the simulator
IAR C/C++ Compiler™ for M16C/R8C	the compiler

Table 2: Naming conventions used in this guide

Brand name	Generic term
IAR Assembler™ for M16C/R8C	the assembler
IAR XLINK Linker™	XLINK, the linker
IAR XAR Library Builder™	the library builder
IAR XLIB Librarian™	the librarian
IAR DLIB Library™	the DLIB library
IAR CLIB Library™	the CLIB library

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

Part 1. Using the compiler

This part of the *IAR C/C++ Compiler Reference Guide for M16C/R8C* includes these chapters:

Getting started

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

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

Next, the compiler is introduced. You will get an overview of the basic settings needed for a project setup, including an overview of the techniques that enable applications to take full advantage of the M16C/R8C Series of CPU cores. In the following chapters, these techniques are studied in more detail.

IAR language overview

There are two high-level programming languages you can use with the IAR C/C++ Compiler for M16C/R8C:

- C, the most widely used high-level programming language in the embedded systems industry. Using the IAR C/C++ Compiler for M16C/R8C, 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 Embedded C++, 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. For more details, see the chapter *Compiler extensions*.

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

It is also possible to implement parts of the application, or the whole application, in assembler language. See the *M16C/R8C IAR Assembler Reference Guide*.

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

Supported M16C/R8C Series devices

The IAR C/C++ Compiler for M16C/R8C supports all devices based on Renesas M16C/1X, 2X, 3X, 5X, 6X, and R8C Series of CPU cores, but not for the M16C/80 core. To compile code for the M16C/80 core, you need the IAR C/C++ Compiler for M32C.

In this guide, all examples are made for the M16C/1X, 2X, 3X, 5X, and 6X Series of CPU cores, but they are valid also for the R8C CPU core, unless otherwise stated.

Building applications—an overview

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

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

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

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

COMPILING

In the command line interface, the following line compiles the source file `myfile.c` into the object file `myfile.r34` using the default settings:

```
iccm16c myfile.c
```

You must also specify some critical options, see *Basic settings for project configuration*, page 5.

LINKING

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

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

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

```
xlink myfile.r34 myfile2.r34 -s __program_start -f lnkm16c.xcl
clm16cffffwc.r34 -o aout.a34 -r
```

In this example, `myfile.r34` and `myfile2.r34` are object files, `lnkm16c.xcl` is the linker command file, and `clm16cffffwc.r34` 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 `intel-standard`.)

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 M16C/R8C Series device you are using. You can specify the options either from the command line interface or in the IDE.

The basic settings are:

- CPU core
- Data model
- Size of `double` floating-point type
- Optimization settings

- Runtime environment.

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

CPU CORE

The compiler supports both the M16C/1X, 2X, 3X, 5X, and 6X Series of CPU cores and the R8C Series of CPU cores. The compiler can be used in two different modes, depending on for which CPU core you want code to be produced.

Use the `--cpu={M16C|R8C}` option to select the core for which the code will be generated.

In the IDE, choose **Project>Options>General Options>Target** and choose an appropriate device from the **Device** drop-down list.

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

These compiler features are not supported if the compiler is used in R8C mode:

- The `__tiny_func` keyword for making a function to be called with the instruction `jsrs` via an entry in the special page area
- The `FLIST` segment holding the special page vector table.

DATA MODEL

One of the characteristics of the M16C/R8C Series of CPU cores is a trade-off in how memory is accessed, between the range from cheap access to small memory areas, up to more expensive access methods that can access any location.

In the compiler, you can set a default memory access method by selecting a data model. These data models are supported:

- The *near* data model
- The *far* data model
- The *huge* data model.

Use the `--data_model` option to specify the data model for your project; see *--data_model*, page 160.

In the IDE, choose **Project>Options>General Options>Target** and choose an appropriate device from the **Device** drop-down list.

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

SIZE OF DOUBLE FLOATING-POINT TYPE

Floating-point values are represented by 32- and 64-bit numbers in standard IEEE 754 format. If you use the compiler option `--64bit_doubles`, you can 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 compiler is a state-of-the-art compiler with an optimizer that performs, among other things, dead-code elimination, constant propagation, inlining, common sub-expression elimination, and precision reduction. It also performs loop optimizations, such as unrolling and induction variable elimination.

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

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

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

RUNTIME ENVIRONMENT

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

Two different sets of runtime libraries are provided:

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

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

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

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

Choosing a runtime library in the IDE

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

Note that for the DLIB library there are different configurations—Normal and Full—which include different levels of support for locale, file descriptors, multibyte characters, et cetera. See *Library configurations*, page 47, for more information.

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

Choosing runtime environment from the command line

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

Command line	Description
`-I m16c\inc`	Specifies the include path to device-specific I/O definition files.
`-I m16c\inc\{clib|dlib}`	Specifies the library-specific include path. Use `clib` or `dlib` depending on which library you are using.
`libraryfile``.r34`	Specifies the library object file
`--dlib_config C:\...\`*`configfile`*`.h`	Specifies the library configuration file (for the IAR DLIB Library only)

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 *Prebuilt libraries*, page 49. The table also shows how the object files correspond to the dependent project options, and the corresponding configuration files. Make sure to use the object file that matches your other project options.

For a list of all prebuilt object files for the IAR CLIB Library, see *Runtime libraries*, page 78. The table also shows how the object files correspond to the dependent project options. Make sure to use the object file that matches your other project options.

Setting library and runtime environment options

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

- The formatters used by the functions `printf`, `scanf`, and their variants, see *Choosing formatters for printf and scanf*, page 53 (DLIB) and *Input and output*, page 81 (CLIB).
- The size of the stack and the heap, see *The stack*, page 38, and *The heap*, page 40, respectively.

Special support for embedded systems

This section briefly describes the extensions provided by the compiler to support specific features of the M16C/R8C Series CPU core.

EXTENDED KEYWORDS

The compiler provides a set of keywords that can be used for configuring how the code is generated. For example, there are keywords for controlling the memory type for individual variables as well as for declaring special function types.

By default, language extensions are enabled in the IDE.

The command line option `-e` makes the extended keywords available, and reserves them so that they cannot be used as variable names. See, *-e*, page 165 for additional information.

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

PRAGMA DIRECTIVES

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

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

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

PREDEFINED SYMBOLS

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

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

SPECIAL FUNCTION TYPES

The special hardware features of the M16C/R8C Series of CPU cores are supported by the compiler's special function types: interrupt, register bank interrupt, monitor, and special page. 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 24. For information about special page functions, see *Special page functions*, page 23.

ACCESSING LOW-LEVEL FEATURES

For hardware-related parts of your application, accessing low-level features is essential. The compiler supports several ways of doing this: intrinsic functions, mixing C and assembler modules, and inline assembler. For information about the different methods, see *Mixing C and assembler*, page 87.

Data storage

This chapter gives a brief introduction to the memory layout of the M16C/R8C Series CPU core and the fundamental ways data can be stored in memory: on the stack, in static (global) memory, or in heap memory. For efficient memory usage, the compiler provides a set of data models and data memory attributes, allowing you to fine-tune the access methods, resulting in smaller code size. The concepts of data models and memory types are described in relation to pointers, structures, Embedded C++ class objects, and non-initialized memory. Finally, detailed information about data storage on the stack and the heap is provided.

Introduction

The compiler supports the M16C/1X–3X, 5X–6X and R8C Series of CPU cores. Different types of physical memory can be placed in the memory range. A typical application will have both read-only memory (ROM) and read/write memory (RAM). In addition, some parts of the memory range contain processor control registers and peripheral units.

The compiler can access memory in different ways. The access methods range from generic but expensive methods that can access the full memory space, to cheap methods that can access limited memory areas. To read more about this, see *Memory types*, page 13.

DIFFERENT WAYS TO STORE DATA

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

- Auto variables.

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

  In this case, the memory is allocated once and for all. The word static in this context means that the amount of memory allocated for this kind of variables does not change while the application is running. For more information, see *Data models*, page 12 and *Memory types*, page 13.

- Dynamically allocated data.

  An application can allocate data on the *heap*, where the data it remains valid until it is explicitly released back to the system by the application. This type of memory is useful when the number of objects is not known until the application executes. Note that there are potential risks connected with using dynamically allocated data in systems with a limited amount of memory, or systems that are expected to run for a long time. For more information, see *Dynamic memory on the heap*, page 20.

Data models

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

- The default placement of static and global variables, and constant literals
- Dynamically allocated data, for example data allocated with `malloc`, or, in C++, the operator `new`
- The default pointer type.

The data model only specifies the default memory type. It is possible to override this for individual variables and pointers. For information about how to specify a memory type for individual objects, see *Using data memory attributes*, page 15.

SPECIFYING A DATA MODEL

Three data models are implemented: *near*, *far*, 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 near data model.

Your project can only use one data model at a time, and the same model must be used by all user modules and all library modules. However, you can override the default memory type for individual data objects and pointers by explicitly specifying a memory attribute, see *Using data memory attributes*, page 15.

This table summarizes the different data models:

Data model	Default variable memory attribute	Default constant memory attribute	Default pointer attribute	Placement of data
Near (default)	`__data16`	`__data16`	`__data16`	Can address the entire 1 Mbyte of memory. Variables are by default placed in the first 64 Kbytes.
Far	`__data16`	`__far`	`__far`	Can address the entire 1 Mbyte of memory. Variables are by default placed in the first 64 Kbytes.
Huge	`__data16`	`__data20`	`__data20`	Can address the entire 1 Mbyte of memory. Variables are by default placed in the first 64 Kbytes.

Table 4: Data model characteristics

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

Use the `--data_model` option to specify the data model for your project; see *--data_model*, page 160.

For a description of the memory types that are used by default in each data model, see *Memory types*, page 13.

Memory types

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

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

For example, the memory accessed using 16-bit addressing is called data16 memory.

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

For more information about memory access methods, see *Memory access methods*, page 101.

DATA13

The data13 memory consists of the low 8 Kbytes of data memory. In hexadecimal notation, this is the address range `0x0000-0x1FFF`. The advantage of data13 is that the compiler can choose to use instructions with bit addressing on data13 objects.

DATA16

Using this memory type, you can place the data objects in the range `0x0000-0xFFFF` in memory.

FAR

Using this memory type, you can place the data objects in the entire memory range `0x0000-0xFFFFF`. However, the size of such an object is limited to 64 Kbytes-1, and it cannot cross a 64-Kbyte physical segment boundary.

The drawback of the far memory type is that the code generated to access the memory is larger and slower than that of data13 and data16. The code also uses more processor registers, which might force local variables to be stored on the stack rather than being allocated in registers.

DATA20

Using this memory type, you can place the data objects in the entire memory range `0x0000-0xFFFFF` with no limitation on size and boundaries.

The drawback of the far memory type is that the code generated to access the memory is larger and slower than that of data13 and data16. The code also uses more processor registers, which might force local variables to be stored on the stack rather than being allocated in registers.

Compared to the far memory type, operations using index-based addressing such as array, struct and class accesses, will be slower and require more code.

USING DATA MEMORY ATTRIBUTES

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

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

Memory type	Keyword	Address range	Pointer keyword	Default in data model
Data13	`__data13`	`0x0-0x1FFF`	`__data16`*	--
Data16	`__data16`	`0x0-0xFFFF`	`__data16`	Near
Far	`__far`	`0x0-0xFFFFF`	`__far`	Far
Data20	`__data20`	`0x0-0xFFFFF`	`__data20`	Huge

Table 5: Memory types and their corresponding memory attributes

*** You cannot create a data13 pointer. However, a data16 pointer can pointer can point to a data13 object.**

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

For backward compatibility, the keywords `__near` and `__huge` are available as aliases for `__data16` and `__data20`, respectively.

In the IDE, language extensions are enabled by default.

Use the `-e` compiler option to enable language extensions. See *-e*, page 165 for additional information.

For reference information about each keyword, see *Descriptions of extended keywords*, page 210. For more information about pointers, see *Pointer types*, page 187.

Syntax

The keywords follow the same syntax as the type qualifiers `const` and `volatile`. The memory attributes are *type attributes* and therefore they must be specified both when variables are defined and in the declaration, see *General syntax rules for extended keywords*, page 205.

The following declarations place the variable `i` and `j` in data16 memory. The variables `k` and `l` will also be placed in data16 memory. The position of the keyword does not have any effect in this case:

```
__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. These two declarations are equivalent:

```
/* Defines via a typedef */
typedef char __data16 Byte;
typedef Byte *BytePtr;
Byte AByte;
BytePtr ABytePointer;

/* Defines directly  */
__data16 char AByte;
char __data16 *ABytePointer;
```

POINTERS AND MEMORY TYPES

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

In the compiler, a pointer also points to some type of memory. The memory type is specified using a keyword before the asterisk. For example, a pointer that points to an integer stored in data16 memory is declared by:

```
int __data16 * MyPtr;
```

Note that the location of the pointer variable `MyPtr` is not affected by the keyword. In the following example, however, the pointer variable `MyPtr2` is placed in data16 memory. Like `MyPtr`, `MyPtr2` points to a character in data20 memory.

```
char __data20 * __data16 MyPtr2;
```

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

If no memory type is specified, the default memory type is used.

Differences between pointer types

A pointer must contain information needed to specify a memory location of a certain memory type. This means that the pointer sizes are different for different memory types. For the IAR C/C++ Compiler for M16C/R8C, the size of the `__data16` pointers is 16 bits. The size of `__data20` and `__far` pointers is 20 bits.

In the compiler, it is legal, with one exception, to convert pointers between different types without explicit casts. The exception is when casting from a larger pointer type to a smaller. Because the pointer size is the same for pointers to all memory types except for the `__data16` pointer, it is illegal to cast other pointer types to a `__data16` pointer.

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 data20 memory.

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

__data20 struct MyStruct Gamma;
```

This declaration is incorrect:

```
struct MyStruct
{
  int mAlpha;
  __data20 int mBeta; /* Incorrect */
};
```

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 data20 memory. To read the following examples, start from the left and add one qualifier at each step

`int MyA;`	A variable defined in default memory determined by the data model in use.
`int __data16 MyB;`	A variable in data16 memory.
`__data20 int MyC;`	A variable in data20 memory.

`int * MyD;`	A pointer stored in default memory. The pointer points to an integer in default memory.
`int __data16 * MyE;`	A pointer stored in default memory. The pointer points to an integer in data16 memory.
`int __data16 * __data20 MyF;`	A pointer stored in data20 memory pointing to an integer stored in data16 memory.

`int __data20 * MyFunction(`
`   int __data16 *);`
	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 data20 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.

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

In the near 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 data20 memory.

```
// A class definition (may be placed in a header file)
class MyClass
{
public:
  int mAlpha;
  int mBeta;

  __data20 static int mGamma;
};


// Needed definitions (should be placed in a source file)
__data20 int MyClass::mGamma;


// An object of class type MyClass
__data16 MyClass Delta;
```

Auto variables—on the stack

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

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

THE STACK

The stack can contain:

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

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

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

Advantages

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

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

Potential problems

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

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

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

Dynamic memory on the heap

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

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

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

Potential problems

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

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

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

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

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

Functions

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

Function-related extensions

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

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

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

For more information about optimizations, see *Writing efficient code*, page 135. For information about the available intrinsic functions for accessing hardware operations, see the chapter *Intrinsic functions*.

Special page functions

A special page function is declared using the `__tiny_func` keyword. It will be called using the `jsrs` instruction (Jump SubRoutine Special page), which is shorter than a `jsr` instruction. The `jsrs` instruction works by looking up the destination address in a table. This will generate less code when the special page function is called from more than one location. However, the table lookup will take extra cycles.

All `__tiny_func` functions must be located within the address range `0xF0000-0xFFFFF`, and the destination table will be located in a segment `FLIST` that must be located at `0xFFE00-0xFFFDB`.

For additional information about the `__tiny_func keyword`, see *__tiny_func*, page 215.

The `__tiny_func` keyword and the `FLIST` segment are only available when the compiler is used in M16C mode, that is when the option `--cpu=M16C` is used.

Primitives for interrupts, concurrency, and OS-related programming

The IAR C/C++ Compiler for M16C/R8C provides the following primitives related to writing interrupt functions, concurrent functions, and OS-related functions:

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

INTERRUPT FUNCTIONS

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

In general, when an interrupt occurs in the code, the CPU core 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 is handled; this includes the values of processor registers and the processor status register. This makes it possible to continue the execution of the original code after the code that handled the interrupt was executed.

The M16C/R8C Series of CPU cores supports many interrupt sources. For each interrupt source, an interrupt routine can be written. Each interrupt routine is associated with a vector number, which is specified in the M16C/R8C Series of CPU cores documentation from the chip manufacturer. If you want to handle several different interrupts using the same interrupt function, you can specify several interrupt vectors.

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

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

```
#pragma vector = TMRA0  /* Symbol defined in I/O header file */
__interrupt void MyTimerA0Interrupt(void)
{
  /* Do something */
}
```

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

For the M16C/R8C Series of CPU cores, there are two types of interrupt vectors, fixed and dynamic. The fixed interrupt vector, `INTVEC1`, is always located at address `0xFFFDC` when you are using the compiler in M16C mode and `0xFFDC` in R8C mode.

The dynamic interrupt vector, `INTVEC`, is pointed at by the `INTB` register. `INTB` is loaded by `cstartup.s34`.

An interrupt function for the fixed interrupt vector must not have a vector number. The name of the function must have one of eight predefined names. For a list of these names, see *INTVEC1*, page 271.

If a vector is specified in the definition of an interrupt function, the dynamic 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 documentation for more information about the interrupt vector table.

The chapter *Assembler language interface* in this guide contains more information about the runtime environment used by interrupt routines.

REGISTER BANK INTERRUPT FUNCTIONS

A register bank interrupt function is declared using the `__regbank_interrupt` keyword. It is a regular interrupt function that switches to register bank 1 on entry, then back to bank 0 on exit, instead of saving and restoring registers. This can be very efficient if the interrupt function uses many registers. For this to work, a register bank interrupt function must not be interrupted by another register bank interrupt function, and no switching between register banks should be performed by any code that can be interrupted by a register bank interrupt function.

MONITOR FUNCTIONS

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

To define a monitor function, you can use the `__monitor` keyword. For reference information, see *__monitor*, page 213.

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

Example of implementing a semaphore in C

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

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


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

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

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

    return 0;
  }
}


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

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


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

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

```
/* An example of using the semaphore. */

void MyProgram(void)
{
  GetLock();

  /* Do something here. */

  ReleaseLock();
}
```

Example of implementing a semaphore in C++

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

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

```
#include <intrinsics.h>

/* Class for controlling critical blocks. */
class Mutex
{
public:
  Mutex()
  {
    // Get hold of current interrupt state.
    mState = __get_interrupt_state();

    // Disable all interrupts.
    __disable_interrupt();
  }

  ~Mutex()
  {
    // Restore the interrupt state.
    __set_interrupt_state(mState);
  }

private:
  __istate_t mState;
};

class Tick
```

```
{
public:
  // Function to read the tick count safely.
  static long GetTick()
  {
    long t;

    // Enter a critical block.
    {
      Mutex m;

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

private:
  static volatile long smTickCount;
};

volatile long Tick::smTickCount = 0;

extern void DoStuff();

void MyMain()
{
  static long nextStop = 100;

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

C++ AND SPECIAL FUNCTION TYPES

C++ member functions can be declared using special function types. However, this restriction applies:

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

Using interrupts and C++ destructors

If interrupts are enabled and the interrupt functions use class objects that have destructors, there might 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()`.

Placing code and data

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

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

Segments and memory

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

WHAT IS A SEGMENT?

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

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

The compiler has several 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. Ready-made linker command files are provided, but, if necessary, they can be easily modified according to the requirements of your target system and application. It is important to remember that, from the linker's point of view, all segments are equal; they are simply named parts of memory.

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

Segment memory type

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

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

Segment memory type	Description
`BIT`	Bit memory, addresses are specified in bits, not in bytes.
`CODE`	For executable code
`CONST`	For data placed in ROM
`FARCONST`	For data placed in far ROM
`HUGECONST`	For data placed in data20 ROM
`NEARCONST`	For data placed in data16 ROM
`DATA`	For data placed in RAM
`FARDATA`	For data placed in far RAM
`HUGEDATA`	For data placed in data20 RAM
`NEARDATA`	For data placed in data16 RAM

Table 6: XLINK segment memory types

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

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

In particular, the linker command file specifies:

- The placement of segments in memory

- The maximum stack size
- The maximum heap size (only for the IAR DLIB runtime environment).

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

CUSTOMIZING THE LINKER COMMAND FILE

The `config` directory contains ready-made linker command files for all supported devices (filename extension `xcl`). The files contain the information required by the linker, and are 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 must add details about the external RAM memory area.

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

Range	Type
`0x00400–0x7FFF`	RAM
`0xF000–0xFFFF`	ROM
`0xC0000–0xFFFFF`	ROM

Table 7: 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

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

- The CPU used:

  ```
  -cM16C
  ```

  This specifies your target CPU core. Note that the parameter should always be `M16C`, also when you use the compiler in R8C mode, that is when you use the compiler option `--cpu=R8C`.
- Definitions of constants used in the file. These are defined using the XLINK option `-D`.

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

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

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

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

Using the -Z command for sequential placement

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

The following illustrates how to use the `-Z` command to place the segment `MYSEGMENTA` followed by the segment `MYSEGMENTB` in `CONST` memory (that is, ROM) in the memory range `0x04000-0x0CFFF`.

```
-Z(CONST)MYSEGMENTA,MYSEGMENTB=4000-CFFF
```

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

```
-Z(CONST)MYSEGMENTA=4000-CFFF
-Z(CODE)MYCODE
```

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

```
-Z(CONST)MYSMALLSEGMENT=4000-20FF
-Z(CONST)MYLARGESEGMENT=4000-CFFF
```

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

Using the -P command for packed placement

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

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

```
-P(DATA)MYDATA=400-1FFF,10000-11FFF
```

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

```
-P(DATA)MYDATA=400-1FFF,F000–F7FF,10000-11FFF
```

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

Note: Copy initialization segments—*`BASENAME`*`_I` and *`BASENAME`*`_ID`—must be placed using `-Z`.

Symbols for available memory areas

To make things easier, the start and end addresses of the memory areas available for your application are defined as symbols in the linker command file.

Example

```
// Memory areas available for the application
-D_USER_RAM_BEGIN=400
-D_USER_RAM_END=7FF
-D_USER_ROM_BEGIN=FA000
-D_USER_ROM_END=FFFFF
```

Data segments

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

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

STATIC MEMORY SEGMENTS

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

- Variables that are initialized to a non-zero value

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

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

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

Segment naming

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

Memory type	Segment group	Memory range
Data13	DATA13	0x0-0x1FFF
Data16	DATA16	0x0-0xFFFF
Far	FAR	0x0-0xFFFFF
Data20	DATA20	0x0-0xFFFFF

Table 8: Memory types with corresponding segment groups

Some of the declared data is placed in non-volatile memory, for example ROM, and some of the data is placed in RAM. For this reason, it is also important to know the XLINK segment memory type of each segment. For more details about segment memory types, see *Segment memory type*, page 32.

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

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

Table 9: Segment name suffixes

For a list of all supported segments, see *Summary of segments*, page 257.

Examples

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

`__data20 int j;` `__data20 int i = 0;`	The data20 variables that are to be initialized to zero when the system starts are placed in the segment `DATA20_Z`.
`__no_init __data20 int j;`	The data20 non-initialized variables are placed in the segment `DATA20_N`.
`__data20 int j = 4;`	The data20 non-zero initialized variables are placed in the segment `DATA20_I` in RAM, and the corresponding initializer data in the segment `DATA20_ID` in ROM.

Initialized data

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

1. It clears the memory of the variables that should be initialized to zero.
2. It initializes the non-zero variables by copying a block of 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 these ranges, the copy will fail:

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

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

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

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

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 these directives to place the static data segments:

```
/* First, the segments to be placed in ROM are defined. */
-Z(CONST)DATA16_C=2000-_USER_ROM_END
-Z(CONST)DATA20_C=_USER_ROM_BEGIN-_USER_ROM_END
-Z(CONST)DATA16_ID,DATA20_ID

/* Then, the RAM data segments are placed in memory. */
-Z(DATA)DATA16_I,DATA16_Z,DATA16_N=_USER_RAM_BEGIN-_USER_RAM_END
-Z(DATA)DATA20_I,DATA20_Z,DATA20_N=10000-11FFF
```

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

THE STACK

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

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

The ISTACK segment holds the special stack used by interrupts and exceptions, see *ISTACK*, page 272.

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

Stack size allocation in the IDE

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

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

Stack size allocation from the command line

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

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

```
-D_CSTACK_SIZE=size
-D_ISTACK_SIZE=size
```

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

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

Placement of stack segment

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

```
-Z(DATA)CSTACK+_CSTACK_SIZE#_USER_RAM_BEGIN-_USER_RAM_END
-Z(DATA)ISTACK+_ISTACK_SIZE#_USER_RAM_BEGIN-_USER_RAM_END
```

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 remainig memory at the same time as it is guaranteed that it will be at least _CSTACK_SIZE bytes in size.

Stack size considerations

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

THE HEAP

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

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

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

Heap segments in DLIB

If you use the compiler in M16C mode, that is, the option `--cpu=M16C` is used, you can take advantage of the different memory types for allocating the heap. 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 data16.

Each heap will reside in a segment with the name `_HEAP` prefixed by a memory attribute, for example `DATA16_HEAP`.

If you use the compiler in R8C mode, that is, the option `--cpu=R8C` is used, the memory allocated to the heap is placed in the segment `DATA16_HEAP`. This segment is only included if dynamic memory allocation is actually used.

For information about available heaps, see *Heaps*, page 71.

Heap segments in the CLIB runtime environment

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

Heap size allocation in the IDE

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

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

Heap size allocation from the command line

The size of the heap segment is defined in the linker 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_DATA20_HEAP_SIZE=size
-D_FAR_HEAP_SIZE=size
```

Note: Normally, these lines are prefixed with the comment character `//`. To make the directive take effect, remove the comment character.

Specify the appropriate size for your application.

Placement of heap segment

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

```
-Z(DATA)DATA16_HEAP+_DATA16_HEAP_SIZE=USER_RAM_BEGIN-USER_RAM_END
-Z(DATA)FAR_HEAP+_FAR_HEAP_SIZE=USER_RAM_BEGIN-USER_RAM_END
-Z(DATA)DATA20_HEAP+_DATA20_HEAP_SIZE=USER_RAM_BEGIN-USER_RAM_END
```

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

Heap size and standard I/O

If your DLIB runtime environment is configured to use `FILE` descriptors, as in the Full configuration, input and output buffers for file handling will be allocated. In that case, 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 M16C/R8C Series of CPU cores. If you use the standard I/O library, you should set the heap size to a value which accommodates the needs of the standard I/O buffer.

LOCATED DATA

A variable that is explicitly placed at an address, for example by using the `#pragma location` directive or the `@` operator, is placed in either the *`SEGMENTNAME`*`_AC` or the *`SEGMENTNAME`*`_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.

USER-DEFINED SEGMENTS

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

Code segments

This section contains descriptions of the segments used for storing code, and the interrupt vector table. For a complete list of all segments, see *Summary of segments*, page 257.

STARTUP CODE

The segment `CSTART` contains code used during system setup (`cstartup`). The system setup code should be placed at the location where the chip starts executing code after a reset.

In this example, this line in the linker command file will place the `CSTART` segment at the address `0xD0000`:

```
-P(CODE)CSTART=D0000
```

NORMAL CODE

Code for normal functions is placed in the `CODE` segment. Again, this is a simple operation in the linker command file:

```
-P(CODE)CODE=_USER_ROM_BEGIN-_USER_ROM_END
```

TINYFUNC-DECLARED CODE

All functions that you declare using the extended keyword `__tiny_func` are located in the segment `TINYFUNC`. This segment must be placed in the special page area `F0000-FFFFF`, which means the linker directive would look like this:

```
-Z(CODE)TINYFUNC=F0000-FFDFF
```

The vector table for `__tiny_func` declared functions are located in the segment `FLIST`, which must be placed in the special page vector area `FFE00-FFFDB`.

EXCEPTION VECTORS

The exception vectors are typically placed in the segments `INTVEC` and `INTVEC1`. The location of `INTVEC1` depends on the chip core and the location of `INTVEC` is user-defined.

C++ dynamic initialization

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

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

For example:

```
-Z(CONST)DIFUNCT=USER_ROM_BEGIN-USER_ROM_END
```

For additional information, see *DIFUNCT*, page 267.

Verifying the linked result of code and data placement

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

SEGMENT TOO LONG ERRORS AND RANGE ERRORS

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

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

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

LINKER MAP FILE

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

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

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

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

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

The DLIB runtime environment

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

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

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

Note that the DLIB runtime environment is the default when you use the C++ language; DLIB can be used with both C and the C++ languages. CLIB on the other hand can only be used with the C language. For information about the CLIB runtime environment, see the chapter *The CLIB runtime environment*.

Introduction to the runtime environment

The runtime environment is the environment in which your application executes. The runtime environment depends on the target hardware, the software environment, and the application code. The IAR DLIB runtime environment can be used as is together with the debugger. However, to be able to run the application on hardware, you must adapt the runtime environment.

This section gives an overview of:

- The runtime environment and its components
- Library selection.

RUNTIME ENVIRONMENT FUNCTIONALITY

The *runtime environment* supports ISO/ANSI C 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 (depending on your product package) as source files, and you can find them in the product subdirectories `m16c\lib` and `m16c\src\lib`, respectively.

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

- Support for hardware features:
  - Direct access to low-level processor operations by means of *intrinsic* functions, such as functions for register handling
  - Peripheral unit registers and interrupt definitions in include files
- Runtime environment support, that is, startup and exit code and low-level interface to some library functions.
- Special compiler support for some functions, for instance functions for floating-point arithmetics.

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

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

LIBRARY SELECTION

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

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

- Setting library options, for example, for choosing `scanf` input and `printf` output formatters, and for specifying the size of the stack and the heap
- Overriding certain library functions, for example `cstartup.s34`, with your own customized versions
- Choosing the level of support for certain standard library functionality, for example, locale, file descriptors, and multibyte characters, by choosing a *library configuration*: normal or full.

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. Therefore, consider carefully whether it is really necessary.

You must build your own library when:

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

For information about how to build a customized library, see *Building and using a customized library*, page 57.

LIBRARY CONFIGURATIONS

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

These DLIB library configurations are available:

Library configuration	Description
Normal DLIB	No locale interface, C locale, no file descriptor support, no multibyte characters in `printf` and `scanf`, and no hexadecimal floating-point numbers in `strtod`.
Full DLIB	Full locale interface, C locale, file descriptor support, multibyte characters in `printf` and `scanf`, and hexadecimal floating-point numbers in `strtod`.

Table 10: Library configurations

You can also 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 57.

The prebuilt libraries are based on the default configurations, see *Prebuilt libraries*, page 49. 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.

This table describes the different levels of debugging support:

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

Table 11: 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 are replaced by functions that communicate with the IAR C-SPY Debugger. For further information, see *C-SPY runtime interface*, page 71.

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 these features:

- The runtime library—CLIB or DLIB
- The CPU core—always M16C, regardless of selected CPU core
- Data model
- Variable data

- The placement of constants
- Size of the `double` floating-point type
- Data alignment
- Copies constants to near
- Library configuration—Normal or Full.

These prebuilt runtime libraries are available:

Library	Data model	Variables area	Constants area	Size of doubles	Data alignment	Copies constants to near	Library configuration
`dlm16cfffdbcf.r34`	Far	Far	Far	64 bits	1 byte	no	Full
`dlm16cfffdbcn.r34`	Far	Far	Far	64 bits	1 byte	no	Normal
`dlm16cfffdwcf.r34`	Far	Far	Far	64 bits	2 bytes	no	Full
`dlm16cfffdwcn.r34`	Far	Far	Far	64 bits	2 bytes	no	Normal
`dlm16cffffbcf.r34`	Far	Far	Far	32 bits	1 byte	no	Full
`dlm16cffffbcn.r34`	Far	Far	Far	32 bits	1 byte	no	Normal
`dlm16cffffwcf.r34`	Far	Far	Far	32 bits	2 bytes	no	Full
`dlm16cffffwcn.r34`	Far	Far	Far	32 bits	2 bytes	no	Normal
`dlm16cfnfdbcf.r34`	Far	Near	Far	64 bits	1 byte	no	Full
`dlm16cfnfdbcn.r34`	Far	Near	Far	64 bits	1 byte	no	Normal
`dlm16cfnfdwcf.r34`	Far	Near	Far	64 bits	2 bytes	no	Full
`dlm16cfnfdwcn.r34`	Far	Near	Far	64 bits	2 bytes	no	Normal
`dlm16cfnffbcf.r34`	Far	Near	Far	32 bits	1 byte	no	Full
`dlm16cfnffbcn.r34`	Far	Near	Far	32 bits	1 byte	no	Normal
`dlm16cfnffwcf.r34`	Far	Near	Far	32 bits	2 bytes	no	Full
`dlm16cfnffwcn.r34`	Far	Near	Far	32 bits	2 bytes	no	Normal
`dlm16chhhdbcf.r34`	Huge	Huge	Huge	64 bits	1 byte	no	Full
`dlm16chhhdbcn.r34`	Huge	Huge	Huge	64 bits	1 byte	no	Normal
`dlm16chhhdwcf.r34`	Huge	Huge	Huge	64 bits	2 bytes	no	Full
`dlm16chhhdwcn.r34`	Huge	Huge	Huge	64 bits	2 bytes	no	Normal
`dlm16chhhfbcf.r34`	Huge	Huge	Huge	32 bits	1 byte	no	Full
`dlm16chhhfbcn.r34`	Huge	Huge	Huge	32 bits	1 byte	no	Normal
`dlm16chhhfwcf.r34`	Huge	Huge	Huge	32 bits	2 bytes	no	Full
`dlm16chhhfwcn.r34`	Huge	Huge	Huge	32 bits	2 bytes	no	Normal

Table 12: Prebuilt libraries

Library	Data model	Variables area	Constants area	Size of doubles	Data alignment	Copies constants to near	Library configuration
`dlm16chnhdbcf.r34`	Huge	Near	Huge	64 bits	1 byte	no	Full
`dlm16chhhdbcn.r34`	Huge	Huge	Huge	64 bits	1 byte	no	Normal
`dlm16chnhdwcf.r34`	Huge	Near	Huge	64 bits	2 bytes	no	Full
`dlm16chnhdwcn.r34`	Huge	Near	Huge	64 bits	2 bytes	no	Normal
`dlm16chnhfbcf.r34`	Huge	Near	Huge	32 bits	1 byte	no	Full
`dlm16chnhfbcn.r34`	Huge	Near	Huge	32 bits	1 byte	no	Normal
`dlm16chnhfwcf.r34`	Huge	Near	Huge	32 bits	2 bytes	no	Full
`dlm16chnhfwcn.r34`	Huge	Near	Huge	32 bits	2 bytes	no	Normal
`dlm16cnnfdbcf.r34`	Near	Near	Far	64 bits	1 byte	no	Full
`dlm16cnnfdbcn.r34`	Near	Near	Far	64 bits	1 byte	no	Normal
`dlm16cnnfdwcf.r34`	Near	Near	Far	64 bits	2 bytes	no	Full
`dlm16cnnfdwcn.r34`	Near	Near	Far	64 bits	2 bytes	no	Normal
`dlm16cnnffbcf.r34`	Near	Near	Far	32 bits	1 byte	no	Full
`dlm16cnnffbcn.r34`	Near	Near	Far	32 bits	1 byte	no	Normal
`dlm16cnnffwcf.r34`	Near	Near	Far	32 bits	2 bytes	no	Full
`dlm16cnnffwcn.r34`	Near	Near	Far	32 bits	2 bytes	no	Normal
`dlm16cnnhdbcf.r34`	Near	Near	Huge	64 bits	1 byte	no	Full
`dlm16cnnhdbcn.r34`	Near	Near	Huge	64 bits	1 byte	no	Normal
`dlm16cnnhdwcf.r34`	Near	Near	Huge	64 bits	2 bytes	no	Full
`dlm16cnnhdwcn.r34`	Near	Near	Huge	64 bits	2 bytes	no	Normal
`dlm16cnnhfbcf.r34`	Near	Near	Huge	32 bits	1 byte	no	Full
`dlm16cnnhfbcn.r34`	Near	Near	Huge	32 bits	1 byte	no	Normal
`dlm16cnnhfwcf.r34`	Near	Near	Huge	32 bits	2 bytes	no	Full
`dlm16cnnhfwcn.r34`	Near	Near	Huge	32 bits	2 bytes	no	Normal
`dlm16cnnndbcf.r34`	Near	Near	Near	64 bits	1 byte	no	Full
`dlm16cnnndbcn.r34`	Near	Near	Near	64 bits	1 byte	no	Normal
`dlm16cnnndbwf.r34`	Near	Near	Near	64 bits	1 byte	yes	Full
`dlm16cnnndbwn.r34`	Near	Near	Near	64 bits	1 byte	yes	Normal
`dlm16cnnndwcf.r34`	Near	Near	Near	64 bits	2 bytes	no	Full
`dlm16cnnndwcn.r34`	Near	Near	Near	64 bits	2 bytes	no	Normal

Table 12: Prebuilt libraries

Library	Data model	Variables area	Constants area	Size of doubles	Data alignment	Copies constants to near	Library configuration
`dlm16cnnndwwf.r34`	Near	Near	Near	64 bits	2 bytes	yes	Full
`dlm16cnnndwwn.r34`	Near	Near	Near	64 bits	2 bytes	yes	Normal
`dlm16cnnnfbcf.r34`	Near	Near	Near	32 bits	1 byte	no	Full
`dlm16cnnnfbcn.r34`	Near	Near	Near	32 bits	1 byte	no	Normal
`dlm16cnnnfbwf.r34`	Near	Near	Near	32 bits	1 byte	yes	Full
`dlm16cnnnfbwn.r34`	Near	Near	Near	32 bits	1 byte	yes	Normal
`dlm16cnnnfwcf.r34`	Near	Near	Near	32 bits	2 bytes	no	Full
`dlm16cnnnfwcn.r34`	Near	Near	Near	32 bits	2 bytes	no	Normal
`dlm16cnnnfwwf.r34`	Near	Near	Near	32 bits	2 bytes	yes	Full
`dlm16cnnnfwwn.r34`	Near	Near	Near	32 bits	2 bytes	yes	Normal

Table 12: Prebuilt libraries

The names of the libraries are constructed in this way:

```
<library><cpu><data_model><variables><constants><doubles>
<data_alignment><constants_to_near><library_config>.r34
```

where

- *<library>* is `dl` for the IAR DLIB runtime environment
- *<cpu>* is always `m16c` (regardless of M16C or R8C)
- *<data_model>* is one of `n`, `f`, or `h` for near, far and huge data model, respectively
- *<variables>* is one of `n`, `f`, or `h`, for placing data in near, far, and huge memory, respectively
- *<constants>* is one of `n`, `f`, or `h`, for placing constant data in near, far, and huge memory, respectively
- *<doubles>* is `f` or `d`, depending on whether doubles are 32 (`f`) or 64 (`d`) bits
- *<data_alignment>* is `w` or `b`, depending on whether the data alignment is 2 bytes (`w`) or 1 byte (`b`)
- *<constants_to_near>* is either `w`, which means that constants are copied to near memory, or `c`, which means that they are not
- *<library_config>* is one of `n` or `f` for normal and full library configuration, respectively.

Examples

- `dlm16chhhfwcf.r34` is an IAR DLIB library that uses the huge data model, places both variable and constant data in huge memory and uses 32-bit doubles. It is 2-byte-aligned, does not copy constants to near memory, and it uses the full library configuration.
- `dlm16cfffdbcn.r34` is an IAR DLIB library that uses the far data model, places both variable and constant data in far memory and uses 64-bit doubles. It is 1-byte-aligned, does not copy constants to near memory, and it uses the normal library configuration.
- `dlm16cnnnfbwf.r34` is an IAR DLIB library that uses the near data model, places both variable and constant data in near memory, and uses 32-bit doubles. It is 1-byte-aligned, copies constants to near memory, and it uses the full library configuration.

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

If you build your application from the command line, you must specify these items to get the required runtime library:

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

  ```
  dlm16cffffwc.r34
  ```
- Specify the include paths for the compiler and assembler:

  ```
  -I m16c\inc\dlib
  ```
- Specify the library configuration file for the compiler:

  ```
  --dlib_config C:\...\dlm16cffffwc.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 `m16c\lib`.

CUSTOMIZING A PREBUILT LIBRARY WITHOUT REBUILDING

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

- Setting options for:
  - Formatters used by `printf` and `scanf`
  - The sizes of the heap and the stack

- Overriding library modules with your own customized versions.

These items can be customized:

Items that can be customized	Described in
Formatters for printf and scanf	*Choosing formatters for printf and scanf*, page 53
Startup and termination code	*System startup and termination*, page 58
Low-level input and output	*Standard streams for input and output*, page 62
File input and output	*File input and output*, page 65
Low-level environment functions	*Environment interaction*, page 68
Low-level signal functions	*Signal and raise*, page 69
Low-level time functions	*Time*, page 70
Size of heaps, stacks, and segments	*Placing code and data*, page 31

Table 13: Customizable items

For a description about how to override library modules, see *Overriding library modules*, page 55.

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 64.

CHOOSING PRINTF FORMATTER

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

This table summarizes the capabilities of the different formatters:

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

Table 14: Formatters for printf

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

For information about how to fine-tune the formatting capabilities even further, see *Configuration symbols for printf and scanf*, page 64.

Specifying the print formatter in the IDE

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

Specifying printf formatter from the command line

To use any other formatter than the default (`_PrintfFull`), add one of these 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.

This table summarizes the capabilities of the different formatters:

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

Table 15: Formatters for scanf

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

For information about how to fine-tune the formatting capabilities even further, see *Configuration symbols for printf and scanf*, page 64.

Specifying scanf formatter in the IDE

To use any other formatter than the default (_ScanfFull), 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 these 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 m16c\src\lib directory.

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

Overriding library modules using the IDE

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

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

Overriding library modules from the command line

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

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

```
iccm16c library_module.c
```

   This creates a replacement object module file named *`library_module`*`.r34`.

   Note: The library configuration file and some other project options must be the same for *`library_module`* as for the rest of your code. For a list of necessary project options, see the release notes provided with the IAR product installation.

4. Add *`library_module`*`.r34` to the XLINK command line, either directly or by using an extended linker command file, for example:

```
xlink library_module.r34 dlm16ccffffwcn.r34
```

   Make sure that *`library_module`*`.r34` 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.r34`, 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 47, it is necessary to rebuild the library. In those cases you must:

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

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

Information about the build process and the IAR Command Line Build Utility, see the *IAR Embedded Workbench® IDE User Guide*.

SETTING UP A LIBRARY PROJECT

The IDE provides a library project template which can be used for customizing the runtime environment configuration. This library template has full library configuration, see Table 10, *Library configurations*, page 47.

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

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

MODIFYING THE LIBRARY FUNCTIONALITY

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

The library functionality is determined by a set of *configuration symbols*. The default values of these symbols are defined in the file `DLib_defaults.h`. This read-only file describes the configuration possibilities. Your library also has its own library configuration file `dlm16ccustom.h`, which sets up that specific library with full library configuration. For more information, see Table 13, *Customizable items*, page 53.

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

Modifying the library configuration file

In your library project, open the file `dlm16ccustom.h` and customize it by setting the values of the configuration symbols according to the application requirements.

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

USING A CUSTOMIZED LIBRARY

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

In the IDE you must do these steps:

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

System startup and termination

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

The code for handling startup and termination is located in the source files `cstartup.s34`, `cexit.s34`, and `low_level_init.c` located in the `m16c\src\lib` directory.

For information about how to customize the system startup code, see *Customizing system initialization*, page 61.

SYSTEM STARTUP

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

For the hardware initialization, it looks like this:

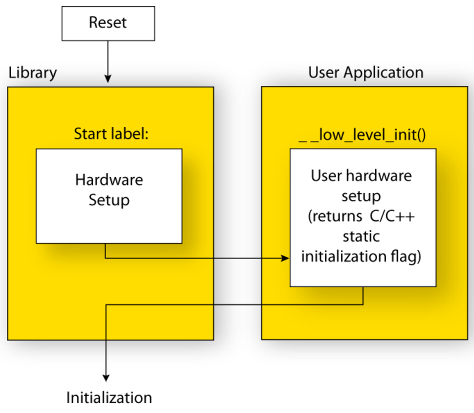


Figure 1: Target hardware initialization phase

- When the CPU is reset it will jump to the program entry label `__program_start` in the system startup code.
- The interrupt stack, `ISTACK`, is initialized.
- The C stack, or user stack, `CSTACK`, is initialized.
- The dynamic interrupt vector is initialized.
- The function `__low_level_init` is called if you defined it, giving the application a chance to perform early initializations.

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

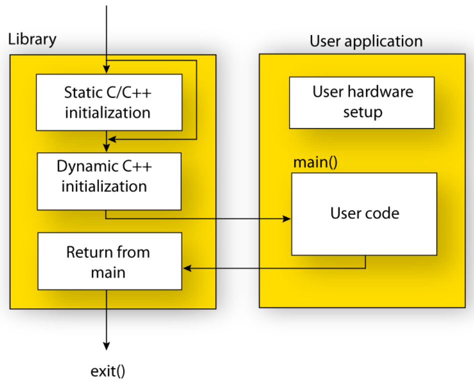


Figure 2: C/C++ initialization phase

- Static and global variables are initialized. That is, zero-initialized variables are cleared and the values of other initialized variables are copied from ROM to RAM memory. This step is skipped if `__low_level_init` returns zero. For more details, see *Initialized data*, page 37

- Static C++ objects are constructed
- The `main` function is called, which starts the application.

SYSTEM TERMINATION

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

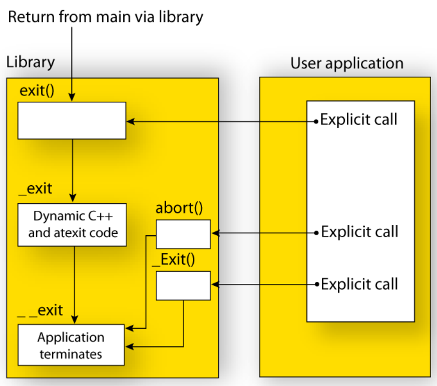


Figure 3: System termination phase

An application can terminate normally in two different ways:

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

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 these operations:

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

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

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

C-SPY interface to system termination

If your project is linked with the XLINK options **With runtime control modules** or **With I/O emulation modules**, the normal __exit and abort functions are replaced with special ones. C-SPY will then recognize when those functions are called and can take appropriate actions to simulate program termination. For more information, see *C-SPY runtime interface*, page 71.

Customizing system initialization

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

You can do this by providing a customized version of the routine __low_level_init, which is called from cstartup.s34 before the data segments are initialized.

The code for handling system startup is located in the source files cstartup.s34 and low_level_init.c, located in the m16c\src\lib directory.

Note: Normally, you do not need to customize cexit.s34.

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 57.

Note: Regardless of whether you modify the routine __low_level_init or the file cstartup.s34, you do not have to rebuild the library.

__LOW_LEVEL_INIT

There is a skeleton low-level initialization file supplied with the product: the C source file, low_level_init.c. The only limitation using a 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.S34

As noted earlier, you should not modify the file `cstartup.s34` if a customized version of `__low_level_init` is enough for your needs. However, if you do need to modify the file `cstartup.s34`, 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 55.

Note that you must make sure that the linker uses the same start label as used in your version of `cstartup.s34`. For information about how to change the start label used by the linker, read about the `-s` option in the *IAR Linker and Library Tools Reference Guide*.

Standard streams for input and output

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

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

IMPLEMENTING LOW-LEVEL CHARACTER INPUT AND OUTPUT

To implement low-level functionality of the `stdin` and `stdout` streams, you must write the functions `__read` and `__write`, respectively. You can find template source code for these functions in the `m16c\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 57. Note that customizing the low-level routines for input and output does not require you to rebuild the library.

Note: If you write your own variants of `__read` or `__write`, special considerations for the C-SPY runtime interface are needed, see *C-SPY runtime interface*, page 71.

Example of using __write

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

```
#include <stddef.h>

__no_init volatile unsigned char lcdIO @ 8;
```

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

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

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

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

  return nChars;
}
```

Note: A call to `__write` where `buf` has the value `NULL` is a command to flush the handle.

Example of using __read

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

```
#include <stddef.h>

__no_init volatile unsigned char kbIO @ 8;

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

  /* Check for stdin
     (only necessary if FILE descriptors are enabled) */
  if (handle != 0)
  {
    return -1;
  }

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

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

  return nChars;
}
```

For information about the `@` operator, see *Controlling data and function placement in memory*, page 127.

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 53.

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

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

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

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

Table 16: Descriptions of printf configuration symbols

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

Table 16: Descriptions of printf configuration symbols (Continued)

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

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

Table 17: Descriptions of scanf configuration symbols

CUSTOMIZING FORMATTING CAPABILITIES

To customize the formatting capabilities, you must set up a library project, see *Building and using a customized library*, page 57. Define the configuration symbols according to your application requirements.

File input and output

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

Note that file I/O capability in the library is only supported by libraries with full library configuration, see *Library configurations*, page 47. In other words, file I/O is supported when the configuration symbol `__DLIB_FILE_DESCRIPTOR` is enabled. If not enabled, functions taking a `FILE *` argument cannot be used.

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

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

Table 18: Low-level I/O files

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

Note: If you link your library with I/O debugging support, C-SPY variants of the low-level I/O functions are linked for interaction with C-SPY. For more information, see *Debug support in the runtime library*, page 48.

Locale

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

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

The DLIB library can be used in two main modes:

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

LOCALE SUPPORT IN PREBUILT LIBRARIES

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

- All prebuilt libraries support the C locale only

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

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

CUSTOMIZING THE LOCALE SUPPORT

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

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

Locale configuration symbols

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

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

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

If you want to customize the locale support, you simply define the locale configuration symbols required by your application. For more information, see *Building and using a customized library*, page 57.

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

Building a library without support for locale interface

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

Building a library with support for locale interface

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

CHANGING LOCALES AT RUNTIME

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

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

`lang_REGION`

or

`lang_REGION.encoding`

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

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

Example

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

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

Environment interaction

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

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

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

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

```
key=value\0
```

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

For example:

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

If you need a more sophisticated environment variable handling, you should implement your own `getenv`, and possibly `putenv` function. This does not require that you rebuild the library. You can find source templates in the files `getenv.c` and `environ.c` in the `m16c\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 55.

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

If you decide to rebuild the library, you can find source templates in the library project template. For further information, see *Building and using a customized library*, page 57.

Note: If you link your application with support for I/O debugging, the functions `getenv` and `system` are replaced by C-SPY variants. For further information, see *Debug support in the runtime library*, page 48.

Signal and raise

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

This does not require that you rebuild the library. You can find source templates in the files `signal.c` and `raise.c` in the `m16c\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 55.

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 57.

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 `m16c\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 55.

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 57.

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

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

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 must rebuild the library, see *Building and using a customized library*, page 57. Enable the configuration symbol `_DLIB_STRTOD_HEX_FLOAT` in the library configuration file.

Assert

If you linked your application with support for runtime debugging, C-SPY will be notified about failed asserts. If this is not the behavior you require, you must add the source file `xreportassert.c` to your application project. Alternatively, you can rebuild the library. The `__ReportAssert` function generates the assert notification. You can find template code in the `m16c\src\lib` directory. For further information, see *Building and using a customized library*, page 57. To turn off assertions, you must define the symbol `NDEBUG`.

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

Heaps

The runtime environment supports heaps in these memory types:

Memory type	Segment name	Memory attribute	Used by default in data model
Data16	`DATA16_HEAP`	`__data16`	Near
Far	`FAR_HEAP`	`__far`	Far
Data20	`DATA20_HEAP`	`__data20`	Huge

Table 19: Heaps and memory types

See *The heap*, page 40 for information about how to set the size for each heap. To use a specific heap, the prefix in the table is the memory attribute to use in front of `malloc`, `free`, `calloc`, and `realloc`, for example `__data16_malloc`. 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 115.

C-SPY runtime interface

To include support for runtime and I/O debugging, you must link your application with the XLINK options **With runtime control modules** or **With I/O emulation modules**, see *Debug support in the runtime library*, page 48.

In this case, C-SPY variants of these library functions are linked to the application:

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

Table 20: Functions with special meanings when linked with debug info

Function	Description
`time`	Returns the time on the host computer
`__write`	`stdin`, `stdout`, and `stderr` will be directed to the Terminal I/O window, all other files will write to the associated host file

Table 20: Functions with special meanings when linked with debug info (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 are implemented. Or, if you need to debug constructions in your application that use `stdin` and `stdout` without the actual hardware device for input and output being available. Another debugging purpose can be to produce debug 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 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 48. This means that when the functions `__read` or `__write` are called to perform I/O operations on the streams `stdin`, `stdout`, or `stderr`, data will be sent to or read from the C-SPY Terminal I/O window.

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

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

Speeding up terminal output

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

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

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

```
-e__write_buffered=__write
```

Checking module consistency

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

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

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

RUNTIME MODEL ATTRIBUTES

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

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

Example

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

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

Table 21: Example of runtime model attributes

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

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

USING RUNTIME MODEL ATTRIBUTES

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

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

For detailed syntax information, see *rtmodel*, page 228.

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

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

For detailed syntax information, see the *M16C/R8C 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 compiler. These can be included in assembler code or in mixed C/C++ and assembler code.

Runtime model attribute	Value	Description
`__rt_version`	*n*	This runtime key is always present in all modules generated by the compiler. If a major change in the runtime characteristics occurs, the value of this key changes.
`__processor`	`M16C`	Note that this runtime attribute is always `M16C`, also when the compiler is used in R8C mode (that is, the option `--cpu=R8C` is used).
`__data_model`	`near`, `far` or `huge`	Corresponds to the data model used in the project.
`__variable_data`	`near`, `far`, or `huge`	Corresponds to where variable data is placed, in near, far, or huge memory.
`__constant_data`	`near`, `far`, or `huge`	Corresponds to where constant data is placed, in near, far, or huge memory.
`__data_alignment`	`1` or `2`	Corresponds to the data alignment used in the project.
`__64bit_doubles`	`Enabled` or `Disabled`	Enabled when 64-bit doubles are used, or disabled when 32-bit doubles are used.
`__calling_convention`	`Simple` or `Normal`	Corresponds to the use of calling convention, simple or normal

Table 22: Predefined runtime model attributes

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

If you are using assembler routines in the C or C++ code, refer to the chapter *Assembler directives* in the *M16C/R8C IAR Assembler Reference Guide.*

Examples

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

```
            module          myCounter
            public          myCounter
            section         CODE:CODE
            rtmodel         "__reg_r4", "counter"
myCounter:  add             r4, r4, #1
            mov             pc, lr
            end
```

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

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

USER-DEFINED RUNTIME MODEL ATTRIBUTES

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

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

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

The CLIB runtime environment

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

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

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

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

For information about migrating from CLIB to DLIB, see the *M16C/R8C IAR Embedded Workbench® Migration Guide*.

Prebuilt libraries

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

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

- The runtime library—CLIB or DLIB
- The CPU core—always M16C, regardless of selected CPU core
- Data model

- Variable data
- Constant data
- Size of the `double` floating-point type
- Data alignment
- Copies constants to near
- Calling convention—normal or simple.

These prebuilt libraries are available:

Library object file	Data model	Variables area	Constants area	Size of doubles	Data alignment	Copies constants to near	Calling convention
`clm16cfffdbc.r34`	Far	Far	Far	64 bits	1 byte	no	normal
`clm16cfffdbcs.r34`	Far	Far	Far	64 bits	1 byte	no	simple
`clm16cfffdwc.r34`	Far	Far	Far	64 bits	2 bytes	no	normal
`clm16cfffdwcs.r34`	Far	Far	Far	64 bits	2 bytes	no	simple
`clm16cffffbc.r34`	Far	Far	Far	32 bits	1 byte	no	normal
`clm16cffffbcs.r34`	Far	Far	Far	32 bits	1 byte	no	simple
`clm16cffffwc.r34`	Far	Far	Far	32 bits	2 bytes	no	normal
`clm16cffffwcs.r34`	Far	Far	Far	32 bits	2 bytes	no	simple
`clm16cfnfdbc.r34`	Far	Near	Far	64 bits	1 byte	no	normal
`clm16cfnfdbcs.r34`	Far	Near	Far	64 bits	1 byte	no	simple
`clm16cfnfdwc.r34`	Far	Near	Far	64 bits	2 bytes	no	normal
`clm16cfnfdwcs.r34`	Far	Near	Far	64 bits	2 bytes	no	simple
`clm16cfnffbc.r34`	Far	Near	Far	32 bits	1 byte	no	normal
`clm16cfnffbcs.r34`	Far	Near	Far	32 bits	1 byte	no	simple
`clm16cfnffwc.r34`	Far	Near	Far	32 bits	2 bytes	no	normal
`clm16cfnffwcs.r34`	Far	Near	Far	32 bits	2 bytes	no	simple
`clm16chhhdbc.r34`	Huge	Huge	Huge	64 bits	1 byte	no	normal
`clm16chhhdbcs.r34`	Huge	Huge	Huge	64 bits	1 byte	no	simple
`clm16chhhdwc.r34`	Huge	Huge	Huge	64 bits	2 bytes	no	normal
`clm16chhhdwcs.r34`	Huge	Huge	Huge	64 bits	2 bytes	no	simple
`clm16chhhfbc.r34`	Huge	Huge	Huge	32 bits	1 byte	no	normal
`clm16chhhfbcs.r34`	Huge	Huge	Huge	32 bits	1 byte	no	simple
`clm16chhhfwc.r34`	Huge	Huge	Huge	32 bits	2 bytes	no	normal

Table 23: Runtime libraries

Library object file	Data model	Variables area	Constants area	Size of doubles	Data alignment	Copies constants to near	Calling convention
`clm16chhhfwcs.r34`	Huge	Huge	Huge	32 bits	2 bytes	no	simple
`clm16chnhdbc.r34`	Huge	Near	Huge	64 bits	1 byte	no	normal
`clm16chnhdwc.r34`	Huge	Near	Huge	64 bits	1 byte	no	normal
`clm16chnhfbc.r34`	Huge	Near	Huge	32 bits	1 byte	no	normal
`clm16chnhfwc.r34`	Huge	Near	Huge	32 bits	2 bytes	no	normal
`clm16cnnfdbc.r34`	Near	Near	Far	64 bits	1 byte	no	normal
`clm16cnnfdwc.r34`	Near	Near	Far	64 bits	2 bytes	no	normal
`clm16cnnffbc.r34`	Near	Near	Far	32 bits	1 byte	no	normal
`clm16cnnffwc.r34`	Near	Near	Far	32 bits	2 bytes	no	normal
`clm16cnnhdbc.r34`	Near	Near	Huge	64 bits	1 byte	no	normal
`clm16cnnhdwc.r34`	Near	Near	Huge	64 bits	1 byte	no	normal
`clm16cnnhfbc.r34`	Near	Near	Huge	32 bits	1 byte	no	normal
`clm16cnnhfwc.r34`	Near	Near	Huge	32 bits	2 bytes	no	normal
`clm16cnnndbc.r34`	Near	Near	Near	64 bits	1 byte	no	normal
`clm16cnnndbcs.r34`	Near	Near	Near	64 bits	1 byte	no	simple
`clm16cnnndbw.r34`	Near	Near	Near	64 bits	1 byte	yes	normal
`clm16cnnndbws.r34`	Near	Near	Near	64 bits	1 byte	yes	simple
`clm16cnnndwc.r34`	Near	Near	Near	64 bits	2 bytes	no	normal
`clm16cnnndwcs.r34`	Near	Near	Near	64 bits	2 bytes	no	simple
`clm16cnnndww.r34`	Near	Near	Near	64 bits	2 bytes	yes	normal
`clm16cnnndwws.r34`	Near	Near	Near	64 bits	2 bytes	yes	simple
`clm16cnnnfbc.r34`	Near	Near	Near	32 bits	1 byte	no	normal
`clm16cnnnfbcs.r34`	Near	Near	Near	32 bits	1 byte	no	simple
`clm16cnnnfbw.r34`	Near	Near	Near	32 bits	1 byte	yes	normal
`clm16cnnnfbws.r34`	Near	Near	Near	32 bits	1 byte	yes	simple
`clm16cnnnfwc.r34`	Near	Near	Near	32 bits	2 bytes	no	normal
`clm16cnnnfwcs.r34`	Near	Near	Near	32 bits	2 bytes	no	simple
`clm16cnnnfww.r34`	Near	Near	Near	32 bits	2 bytes	yes	normal
`clm16cnnnfwws.r34`	Near	Near	Near	32 bits	2 bytes	yes	simple

Table 23: Runtime libraries (Continued)

The names of the libraries are constructed in this way:

<library><cpu><data_model><variables><constants><doubles>
<data_alignment><constants_to_near><calling_convention>`.r34`

where

- *<library>* is `cl` for the IAR CLIB runtime environment
- *<cpu>* is always `m16c` (regardless of M16C or R8C)
- *<data_model>* is one of `n`, `f` or `h` for near, far and huge data model, respectively
- *<variables>* is one of `n`, `f`, or `h`, for placing data in near, far, and huge memory, respectively
- *<constants>* is one of `n`, `f`, or `h`, for placing constant data in near, far, and huge memory, respectively
- *<doubles>* is `f` or `d`, depending on whether doubles are 32 (`f`) or 64 (`d`) bits
- *<data_alignment>* is `w` or `b`, depending on whether the data alignment is 2 bytes (`w`) or 1 byte (`b`)
- *<constants_to_near>* is either `w`, which means that constants are copied to near memory, or `c`, which means that they are not
- *<calling_convention>* is `s` if the library uses the simple calling convention, otherwise this position in the library name is empty.

Examples

- `clm16cffffwc.r34` is an IAR CLIB library that uses the far data model, places both variable and constant data in far memory and uses 32-bit doubles. It is 2-byte-aligned, and it does not copy constants to near memory. It uses the normal calling convention.
- `clm16cnnnfwc.r34` is an IAR CLIB library that uses the near data model, places both variable and constant data in near memory and uses 32-bit doubles. It is 2-byte-aligned, and it does not copy constants to near memory. It uses the normal calling convention.
- `clm16cnnndbcs.r34` is an IAR CLIB library that uses the near data model, places both variable and constant data in near memory, and uses 64-bit doubles. It is 1-byte-aligned, and it does not copy constants to near memory. It uses the simple calling convention.

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

If you build your application from the command line, you must specify these items to get the required runtime library:

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

  ```
  clm16cffffwc.r34
  ```
- Specify the include paths for the compiler and assembler:

  ```
  -I m16c\inc\clib
  ```

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

You can find the library object files in the subdirectory `m16c\lib`.

Input and output

You can customize:

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

CHARACTER-BASED I/O

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

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

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

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

```
__no_init volatile unsigned char devIO @ 8;

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

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

For information about how to include your own modified version of `putchar` and `getchar` in your project build process, see *Overriding library modules*, page 55.

FORMATTERS USED BY PRINTF AND SPRINTF

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

_medium_write

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

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

`_medium_write` is considerably smaller than `_formatted_write`.

_small_write

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

Specifying the printf formatter in the IDE

1 Choose **Project>Options** and select the **General Options** category. Click the **Library options** tab.

2 Select the appropriate **Printf formatter** option, which can be either **Small**, **Medium**, or **Large**.

Specifying the printf formatter from the command line

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

```
-e_small_write=_formatted_write
```

or

```
-e_medium_write=_formatted_write
```

To use the full version, remove the line.

Customizing printf

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

For such applications, a much reduced version of the `printf` function (without `sprintf`) is supplied in source form in the file `intwri.c`. This file can be modified to meet your requirements, and the compiled module inserted into the library in place of the original file; see *Overriding library modules*, page 55.

FORMATTERS USED BY SCANF AND SSCANF

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

_medium_read

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

Specifying the scanf formatter in the IDE

1 Choose **Project>Options** and select the **General Options** category. Click the **Library options** tab.

2 Select the appropriate **Scanf formatter** option, which can be either **Medium** or **Large**.

Specifying the read formatter from the command line

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

```
-e_medium_read=_formatted_read
```

To use the full version, remove the line.

System startup and termination

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

The code for handling startup and termination is located in the source files `cstartup.s34` and `low_level_init.c` located in the `m16c\src\lib` directory.

SYSTEM STARTUP

When an application is initialized, several steps are performed:

- The custom function `__low_level_init` is called if you defined it, giving the application a chance to perform early initializations
- Static variables are initialized; this includes clearing zero-initialized memory and copying the ROM image of the RAM memory of the remaining initialized variables
- The `main` function is called, which starts the application.

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

SYSTEM TERMINATION

An application can terminate normally in two different ways:

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

Because the ISO/ANSI C standard states that the two methods should be equivalent, the `cstartup` code calls the `exit` function if `main` returns. The parameter passed to the `exit` function is the return value of `main`. The default `exit` function is written in assembler.

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

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

Overriding default library modules

The IAR CLIB Library contains modules which you probably need to override with your own customized modules, for example for character-based I/O, without rebuilding the entire library. For information about how to override default library modules, see *Overriding library modules*, page 55, in the chapter *The DLIB runtime environment*.

Customizing system initialization

For information about how to customize system initialization, see *Customizing system initialization*, page 61.

C-SPY runtime interface

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

THE DEBUGGER TERMINAL I/O WINDOW

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

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

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

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

TERMINATION

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

Checking module consistency

For information about how to check module consistency, see *Checking module consistency*, page 73.

Assembler language interface

When you develop an application for an embedded system, there might be situations where you will find it necessary to write parts of the code in assembler, for example when using mechanisms in the M16C/R8C Series CPU core that require precise timing and special instruction sequences.

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

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

Mixing C and assembler

The IAR C/C++ Compiler for M16C/R8C provides several ways to access low-level resources:

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

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

INTRINSIC FUNCTIONS

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

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

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

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

MIXING C AND ASSEMBLER MODULES

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

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

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

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

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

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

The first issue is discussed in the section *Calling assembler routines from C*, page 90. The following two are covered in the section *Calling convention*, page 93.

The section on memory access methods, page 101, 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 104.

The recommended method for mixing C or C++ and assembler modules is described in *Calling assembler routines from C*, page 90, and *Calling assembler routines from C++*, page 92, respectively.

INLINE ASSEMBLER

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

```
static int sFlag;

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

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

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

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

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

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

Calling assembler routines from C

An assembler routine that will be called from C must:

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

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

  or

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

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

The compiler supports two different calling conventions, see *Calling convention*, page 93.

CREATING SKELETON CODE

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

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

```
extern int gInt;
extern char gChar;

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

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

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

COMPILING THE CODE

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

Use these options to compile the skeleton code:

```
iccm16c skeleton.c -lA .
```

The `-lA` option creates an assembler language output file including C or C++ source lines as assembler comments. The `.` (period) specifies that the assembler file should be named in the same way as the C or C++ module (`skeleton`), but with the filename extension `s34`. Also remember to specify the data model you are using, a low level of optimization, and `-e` for enabling language extensions.

The result is the assembler source output file `skeleton.s34`.

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

The output file

The output file contains the following important information:

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

The `CFI` directives describe the call frame information needed by the Call Stack window in the debugger. For more information, see *Call frame information*, page 104.

Calling assembler routines from C++

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

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

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

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

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

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

```
class MyClass;

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

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

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

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

Calling convention

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

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

The compiler provides two calling conventions—one normal, which is used by default, and one simple. This section describes the calling conventions used by the compiler. These items are examined:

- Choosing a calling convention
- Function declarations
- C and C++ linkage
- Preserved versus scratch registers

- Function entrance
- Function exit
- Return address handling.

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

CHOOSING A CALLING CONVENTION

You can choose between two calling conventions:

- The *Simple* calling convention offers a simple assembler interface. It is compatible with previous versions of the compiler. This calling convention is recommended for use with assembler code as it will remain unchanged over time. For information about the keyword `__simple`, see *__simple*, page 214.
- The *Normal* calling convention is the default and is used in all prebuilt DLIB libraries. It is more efficient than the Simple calling convention, but also more complex to understand and subject to change in later versions of the compiler.

The Normal calling convention is used by default. To specify the calling convention, use the `--calling_convention={simple|normal}` command line option.

In the IDE, the calling convention option is located on the **Project>Options>General Options>Target** page.

FUNCTION DECLARATIONS

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

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

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

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

USING C LINKAGE IN C++ SOURCE CODE

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

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

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

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

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

int F(int);

#ifdef __cplusplus
}
#endif
```

PRESERVED VERSUS SCRATCH REGISTERS

The general M16C/R8C Series CPU registers are divided into three separate sets, which are described in this section.

Scratch registers

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

In the normal calling convention, the registers `R0`, `R2`, and `A0` are scratch registers. In the simple calling convention, there are no scratch registers.

Preserved registers

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

In the normal calling convention, the registers `FB`, `R1`, `R3`, `A1`, and `SB` are preserved registers. In the simple calling convention, all registers are preserved.

Special registers

For some registers, you must consider certain prerequisites:

- The stack pointer register 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.

FUNCTION ENTRANCE

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

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

Hidden parameters

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

- If the function returns a structure or a `double`, the memory location where the structure will be stored is passed as a hidden first parameter, followed by a `this` pointer, if there is one. Then come the visible parameters.
- If the function is a non-static C++ member function, then the `this` pointer is passed as the first parameter (but placed after the return structure pointer, if there is one). The reason for the requirement that the member function must be non-static is that static member methods do not have a `this` pointer.

Register parameters

One and two-byte scalar parameters—for example, `char`, `short`, and pointers to data16 memory—require one register. Larger scalar types and 32-bit floats require two.

The assignment of registers to parameters is a straightforward process. Traversing the parameters in strict order from left to right, the first parameter is assigned to an available register or register pair. Should there be no suitable register available, the parameter is passed on the stack. In this case, and if the simple calling convention is used, any remaining parameters will also be passed on the stack.

The registers available for passing parameters differ for the normal and simple calling convention.

The normal calling convention

The registers `A0`, `R0`, and `R2` are used as much as possible to pass parameters. This enables at most 6 bytes of parameters to be passed in registers. `R0` can be used as `R0L` and `R0H` to pass two 8-bit parameters.

The compiler will scan the parameters from left to right and assign them to the first available register in a priority order for each type of parameter as follows:

Parameters	**Passed in registers**
8-bit values	`R0L` if possible, otherwise `R0H`
16-bit non-pointer parameters	`R0` if no byte-size register candidates are found within the first 6 bytes of parameters; otherwise in `R2` if it is available or else in `A0`
16-bit pointer parameters	`A0` if possible, otherwise like 16-bit non-pointer parameters
32-bit values, including 20-bit pointers	`R2R0` if it is available

Table 24: Registers used for passing parameters in the normal calling convention

Note: Parameters of a struct or class type will never be passed in registers.

The simple calling convention

Only the first paramater may be passed in a register, either in `R0L`, `R0`, or `R2R0`, depending on the size of the first parameter.

Stack parameters and layout

Stack parameters are stored in the main memory, starting at the location pointed to by the stack pointer. Below the stack pointer (toward low memory) there is free space that the called function can use. The first stack parameter is stored at the location pointed to by the stack pointer. The next one is stored at the next location on the stack.

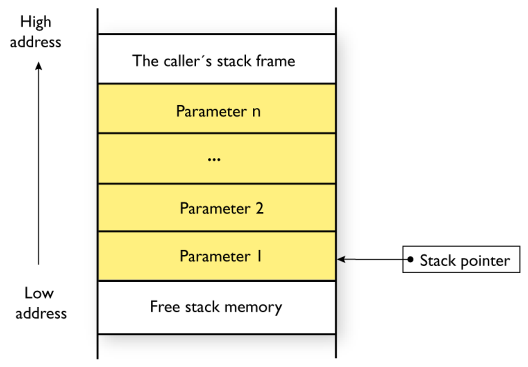


Figure 4: Stack image after the function call

FUNCTION EXIT

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

The return value of a function, if any, can be scalar (such as integers and pointers), floating-point, or a structure. In the case of structures and classes, the return value is a pointer, see *Hidden parameters*, page 96.

Registers used for returning values

The registers available for returning values are `R0` and `R2R0`.

Return values	Passed in registers
8-bit scalar values	`R0L`
16-bit scalar values	`R0`
32-bit values including 20-bit pointers	`R2R0`
64-bit values	`R0` for the near data model and `R2R0` for the other data models

Table 25: Registers used for returning values

Stack layout at function exit

It is the responsibility of the caller to clean the stack after the called function returns.

Return address handling

A function written in assembler language should, when finished, return to the caller.

Typically, a function returns by using the `rts` instruction.

EXAMPLES

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

Example 1

Assume this function declaration:

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

This function takes one parameter in the register `R0L`, and the return value is passed back to its caller in the same register.

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

```
        PUBLIC add1

        RSEG CODE:CODE:REORDER:NOROOT(0)
add1:
        ADD.B   #0x1, R0L
        RTS

        END
```

Example 2

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

```
struct MyStruct
{
  int mA;
};

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

Following the *normal* calling convention, the calling function must reserve 2 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 `R0`. Following the *simple* calling convention, four bytes are used on the stack, of which the top two are the first parameter.

Example 3

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

```
struct MyStruct
{
  int mA;
};

struct MyStruct MyFunction(int x);
```

It is the responsibility of the calling function to allocate a memory location for the return value and pass a pointer to it as a hidden first parameter. The pointer to the location where the return value should be stored is passed in `A0` in the near data model, and in `R2R0` in the other models. The caller assumes that these registers remain untouched. When using the normal calling convention, the parameter `x` is passed in `R0` in the near data model, and in `A0` in the other data models.

Assume that the function instead was declared to return a pointer to the structure:

```
struct MyStruct *MyFunction(int x);
```

In this case, the return value is a scalar, so there is no hidden parameter. The parameter `x` is passed in `R0`, and the return value is returned in `R0` or (`R2`,`R0`), depending on the data model.

FUNCTION DIRECTIVES

Note: This type of directive is primarily intended to support static overlay, a feature which is useful in some smaller microcontrollers. The IAR C/C++ Compiler for M16C/R8C does not use static overlay, because it has no use for it.

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

For reference information about the function directives, see the *M16C/R8C IAR Assembler Reference Guide*.

Calling functions

Functions can be called in two fundamentally different ways—directly or via a function pointer. In this section we will discuss how both types of calls will be performed.

ASSEMBLER INSTRUCTIONS USED FOR CALLING FUNCTIONS

This section presents the assembler instructions that can be used for calling and returning from functions on the M16C/R8C Series CPU core.

The normal function calling instruction is the `jsr` instruction:

```
            jsr.a label
```

The location that the called function should return to (that is, the location immediately after this instruction) is stored on the stack.

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

```
            jsri.a funcptr
```

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 M16C/R8C Series instruction set, in particular the different addressing modes used by the instructions that can access memory.

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

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

These three examples can be illustrated by this C program:

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

struct MyStruct
{
  long mA;
  char mB;
};

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

THE DATA13 MEMORY ACCESS METHOD

Data13 memory is located in the first 8 Kbytes of memory. The data13 memory access method is very similar to the data16 memory access method. In fact, there will be no difference in code generated for our previous example when using data13 memory. The data13 memory access method is strictly a bit access optimization: the M16C/R8C Series has special bit addressing modes that makes accessing a single bit more efficient if it is located in the first 8Kbytes of memory.

Examples

```
INT3IC_bit.POL = INT5IC_bit.POL;
```

Using the data13 access method:

`BTST 4,0x048`	Load bit to C flag
`BMC  4,0x044`	Store C flag

Using the data16 access method:

`MOV.B 0x048,R0L`
`BTST 4,R0`
	Load bit to C flag

`MOV.B 0x044,R0L`
`BMC 4,R0`	

`MOV.B R0L`	Store C flag

THE DATA16 MEMORY ACCESS METHOD

Data16 memory is located in the first 64 Kbytes of memory. This is the only memory type that can be accessed using 16-bit pointers and using a 16-bit index type. The advantages are that a single 16-bit register can be used, instead of a register pair and that most instructions can access this memory directly.

Examples

These examples access data16 memory in different ways:

`MOV.B x,R0L`	Access the global variable `x`
`ADD.B y[A0],R0L`	Access an entry in the global array `y`
`ADD.B 4[A0],R0L`	Accessing a member of a struct

THE FAR MEMORY ACCESS METHOD

The far memory access method is a compromise between the data16 and data20 access methods. This method can access the entire memory range, assuming that no data objects straddle a 64-Kbyte boundary. This means that although you must use register pairs for pointers, single 16-bit registers can be used for indexing. As in the case of the data20 access method, special instructions must be used for loading from and storing to far memory, but pointer arithmetics will not generate any extra instructions.

Examples

These examples access far memory in different ways:

`LDE.B x,R0L`	Access the global variable `x`
`MOV.W #LWRD(y),A0`	Access an entry in the global array
`MOV.W #HWRD(y),A1`	
`ADD.W R0,A0`	
`LDE.B [A1A0], R0L`	
`ADD.W #4,A0`	
`LDE.B [A1A0],R0L`	Accessing a member of a struct

THE DATA20 MEMORY ACCESS METHOD

The data20 memory access method can access the entire memory range. The drawbacks of this access method are that register pairs must be used for both pointers and indexes, special instructions must be used for loading from and storing to data20 memory, and that pointer arithmetics will generate extra instructions. This can result in larger and slower code when comparing to code accessing other types of data.

The pointer size is 20 bits and the index type has a size of 20 bits.

Examples

These examples access data20 memory in different ways:

`LDE.B x,R0L`	Access the global variable `x`
`MOV.W #LWRD(y),A0`	Access an array
`MOV.W #HWRD(y),A1`	
`ADD.W R0,A0`	
`ADC.W R2,A1`	
`LDE.B [A1A0],R0L`	
`ADD.W #4,A0`	Accessing a member of a struct
`ADCF A1`	
`LDE.B [A1A0],R0L`	

Call frame information

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

If you want the call stack to be available when debugging a routine written in assembler language, you must supply equivalent debug information in your assembler source using the assembler directive `CFI`. This directive is described in detail in the *M16C/R8C IAR Assembler Reference Guide*.

CFI DIRECTIVES

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

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

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

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

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

Resource	Description
`CFA`	The call frame address
`R0L`, `R0H`, `R1L`, `R1H`, `R2`, `R3`, `A0`, `A1`	Regular registers
`SP`	The stack pointer
`FB`	The frame base register
`SB`	The static base register
`?RET`	The return address register
`?RHI`	The 8 most significant bits of the return address
`?RLO`	The 16 least significant bits of the return address

Table 26: Call frame information resources defined in a names block

CREATING ASSEMBLER SOURCE WITH CFI SUPPORT

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

1 Start with suitable C source code, for example:

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

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

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

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

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

```
        NAME cfiexample

        RTMODEL "__64bit_doubles", "Disabled"
        RTMODEL "__calling_convention", "Normal"
        RTMODEL "__constant_data", "near"
        RTMODEL "__data_alignment", "2"
        RTMODEL "__data_model", "near"
        RTMODEL "__processor", "M16C"
        RTMODEL "__rt_version", "1"
        RTMODEL "__variable_data", "near"

        RSEG CSTACK:NEARDATA:REORDER:NOROOT(0)

        PUBLIC cfiExample
        FUNCTION cfiExample,021203H
        ARGFRAME CSTACK, 0, STACK

        CFI Names cfiNames0
        CFI StackFrame CFA SP NEARDATA
        CFI Resource R0L:8, R0H:8, R1L:8, R1H:8, R2:16, R3:16,
A0:16, A1:16
        CFI Resource FB:16, SB:16, SP:16
        CFI VirtualResource ?RET:24, ?RHI:8, ?RLO:16
        CFI ResourceParts ?RET ?RHI, ?RLO
        CFI EndNames cfiNames0

        CFI Common cfiCommon0 Using cfiNames0
        CFI CodeAlign 1
        CFI DataAlign 1
        CFI ReturnAddress ?RET NEARDATA
        CFI CFA SP+3
        CFI R0L Undefined
        CFI R0H Undefined
        CFI R1L SameValue
        CFI R1H SameValue
        CFI R2 Undefined
        CFI R3 SameValue
        CFI A0 Undefined
        CFI A1 SameValue
        CFI FB SameValue
        CFI SB SameValue
        CFI ?RET Concat
        CFI ?RHI Frame(CFA, -1)
        CFI ?RLO Frame(CFA, -3)
        CFI EndCommon cfiCommon0
        EXTERN F
        FUNCTION F,0202H
        ARGFRAME CSTACK, 0, STACK


        RSEG CODE:CODE:REORDER:NOROOT(0)
cfiExample:
        CFI Block cfiBlock0 Using cfiCommon0
        CFI Function cfiExample
        FUNCALL cfiExample, F
        LOCFRAME CSTACK, 0, STACK
        ARGFRAME CSTACK, 0, STACK
        ADD.B   #-0x1, SP
        CFI CFA SP+4
        PUSHM   R1
        CFI R1H Frame(CFA, -5)
        CFI R1L Frame(CFA, -6)
        CFI CFA SP+6
        MOV.W   R0, R1
        MOV.W   R1, R0
        JSR.A   F
        ADD.W   R1, R0
        POPM    R1
        CFI R1L SameValue
        CFI R1H SameValue
        CFI CFA SP+4
        ADD.B   #0x1, SP
        CFI CFA SP+3
        RTS
        CFI EndBlock cfiBlock0

        END
```

Note: The header file `cfi.m34` contains the macros `XCFI_NAMES` and `XCFI_COMMON`, which declare a typical names block and a typical common block. These two macros declare several resources, both concrete and virtual.

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 their argument lists are sufficiently different
- Type-safe memory management using the operators `new` and `delete`
- Inline functions, which are indicated as particularly suitable for inline expansion.

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

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

- Templates
- Multiple and virtual inheritance
- Exception handling

- Runtime type information
- New cast syntax (the operators `dynamic_cast`, `static_cast`, `reinterpret_cast`, and `const_cast`)
- Namespaces
- The `mutable` attribute.

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

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

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

EXTENDED EMBEDDED C++

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

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

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

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

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

ENABLING C++ SUPPORT

In the compiler, the default language is C. To be able to compile files written in Embedded C++, you must use the `--ec++` compiler option. See *--ec++*, page 166. 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 166.

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

Feature descriptions

When you write C++ source code for the IAR C/C++ Compiler for M16C/R8C, you must be aware of some benefits and some possible quirks when mixing C++ features—such as classes, and class members—with IAR language extensions, such as IAR-specific attributes.

CLASSES

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

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

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

For further information about attributes, see *Type qualifiers*, page 191.

Example

```
class MyClass
{
public:
  // Locate a static variable in __data16 memory at address 60
  static __data16 __no_init int mI @ 60;

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

  // Locate a function in __tiny_func memory
  __tiny_Func void G();
  // Locate a virtual function in __tiny_func memory
  virtual __tiny_func void H();

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

The this pointer

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

Class memory

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

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

Example

```
class __data20 C
{
public:
  void MyF();        // Has a this pointer of type C __data20 *
  void MyF() const; // Has a this pointer of type
                     // C __data20 const *
  C();               // Has a this pointer pointing into data20
                     // memory
  C(C const &);      // Takes a parameter of type C __data20
                     // const & (also true of generated copy
                     // constructor)
  int mI;
};
```

```
C Ca;                // Resides in data20 memory instead of the
                     // default memory
C __data16 Cb;       // Resides in data16 memory, the 'this'
                     // pointer still points into data20 memory

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

C *Cp1;              // Creates a pointer to data20 memory
C __data16 *Cp2;     // Creates a pointer to data16 memory
```

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 __data20 C;
```

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

A built-in operator returns the class memory type associated with a class, `__memory_of(`*`class`*`)`. For instance, `__memory_of(C)` returns `__data20`.

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 __data20 D : public C
{ // OK, same class memory
public:
  void MyG();
  int mJ;
};

class __data16 E : public C
{ // OK, data16 memory is inside data20
public:
  void MyG() // Has a this pointer pointing into data16 memory
  {
    MyF();    // Gets a this pointer into data20 memory
  }
  int mJ;
};
```

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

A `new` expression on the class will allocate memory in the heap associated with the class memory. A `delete` expression will naturally deallocate the memory back to the same heap. To override the default `new` and `delete` operator for a class, declare

```
void *operator new(size_t);
void operator delete(void *);
```

as member functions, just like in ordinary C++.

For more information about memory types, see *Memory types*, page 13.

FUNCTION TYPES

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

Example

```
extern "C"
{
  typedef void (*FpC)(void);    // A C function typedef
}

typedef void (*FpCpp)(void);    // A C++ function typedef

FpC F1;
FpCpp F2;
void MyF(FpC);

void MyG()
{
  MyF(F1);                      // Always works
  MyF(F2);                      // FpCpp is compatible with FpC
}
```

NEW AND DELETE OPERATORS

There are operators for `new` and `delete` for each memory that can have a heap, that is, data16, data20, and far memory.

```
// Assumes that there is a heap in both __data16 and __data20
memory
void __data20 *operator new __data20(__data20_size_t);
void __data16  *operator new __data16 (__data16_size_t);
void operator delete(void __data20 *);
void operator delete(void __data16  *);

// And correspondingly for array new and delete operators
void __data20 *operator new[] __data20(__data20_size_t);
void __data16  *operator new[] __data16 (__data16_size_t);
void operator delete[](void __data20 *);
void operator delete[](void __data16  *);
```

Use this syntax if you want to override both global and class-specific `operator new` and `operator delete` for any data memory.

Note that there is a special syntax to name the `operator new` functions for each memory, while the naming for the `operator delete` functions relies on normal overloading.

New and delete expressions

A `new` expression calls the `operator new` function for the memory of the type given. If a `class`, `struct`, or `union` type with a class memory is used, the class memory will determine the `operator new` function called. For example,

```
void MyF()
{
  // Calls operator new __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 __data20(__data20_size_t)
  class __data20 S
  {
  };
  S *s = new S;
  // Calls operator delete(void __data16 *)
  delete p;
  // Calls operator delete(void __data20  *)
  delete s;

  int __data20 *t = new __data16 int;
  delete t; // Error: Causes a corrupt heap
}
```

Note that the pointer used in a `delete` expression must have the correct type, that is, the same type as that returned by the `new` expression. If you use a pointer to the wrong memory, the result might be a corrupt heap. For example,

TEMPLATES

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

Templates and data memory attributes

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

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

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

Example

```
// We assume that __data20 is the memory type of the default
// pointer.
template<typename> class Z {};
template<typename T> class Z<T *> {};

Z<int __data16 *> Zn;   // T = int __data16
Z<int __data20 *> Zf;   // T = int
Z<int          *> Zd;   // T = int
Z<int __far *> Zh;      // T = int __far
```

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

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

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

Example

```
// We assume that __data20 is the memory type of the default
// pointer.
template<typename T> void fun(T *);

void MyF()
{
  fun((int __data16 *) 0);  // T = int. The result is different
                            // than the analogous situation with
                            // class template specializations.
  fun((int         *) 0);      // T = int
  fun((int __data20  *) 0); // T = int
  fun((int __far *) 0);     // T = int __far
}
```

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

Example

```
// We assume that __data20 is the memory type of the default
// pointer.
#pragma basic_template_matching
template<typename T> void fun(T *);

void MyF()
{
  fun((int __data16 *) 0); // T = int __data16
}
```

Non-type template parameters

It is allowed to have a reference to a memory type as a template parameter, even if pointers to that memory type are not allowed.

Example

```
extern int __data16SFR X;

template<__data16SFR int &y>
void Foo()
{
  y = 17;
}

void Bar()
{
  Foo<X>();
}
```

The standard template library

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

The containers in the STL, like `vector` and `map`, are memory attribute aware. This means that a container can be declared to reside in a specific memory type which has the following consequences:

- The container itself will reside in the chosen memory
- Allocations of elements in the container will use a heap for the chosen memory
- All references inside it use pointers to the chosen memory.

Example

```
#include <vector>

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 __far> __data16 Y;      // Y placed in data16 memory,
                                   // heap allocation from far,
                                   // uses pointers to far memory
```

Note also that `map<key, T>`, `multimap<key, T>`, `hash_map<key, T>`, and `hash_multimap<key, T>` all use the memory of `T`. This means that the `value_type` of these collections will be `pair<key, const T>` *`mem`* where *`mem`* is the memory type of `T`. Supplying a key with a memory type is not useful.

Example

Note that two containers that only differ by the data memory attribute they use cannot be assigned to each other. Instead, the templated assign member method must be used.

```
#include <vector>

vector<int __data16> X;
vector<int __far> Y;

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

Note: To be able to watch STL containers with many elements in a comprehensive way, the **STL container expansion** option—available by choosing **Tools>Options>Debugger**—is set to display only a few items at first.

VARIANTS OF CASTS

In Extended EC++ these 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
```

You must make sure that identifiers in your application do not interfere with identifiers in the runtime library.

POINTER TO MEMBER FUNCTIONS

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

Example

```
class X
{
public:
  __tiny_func void F();
};

void (__tiny_func X::*PMF)(void) = &X::F;
```

USING INTERRUPTS AND EC++ DESTRUCTORS

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

To avoid this, make sure that interrupts are disabled when returning from `main` or when calling `exit` or `abort`.

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 enable IAR language extensions, the following C++ language extensions are available in the compiler:

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

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

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

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

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

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

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

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

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

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

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

```
bool X;

char *P1 = X ? "abc" : "def";        //Possible when using IAR
                                     //language extensions
char const *P2 = X ? "abc" : "def"; //According to standard
```

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

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

Efficient coding for embedded applications

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

The topics discussed are:

- Selecting data types
- Controlling data and function placement in memory
- Controlling compiler optimizations
- Writing efficient code.

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

Selecting data types

For efficient treatment of data, you should consider the data types used and the most efficient placement of the variables.

USING EFFICIENT DATA TYPES

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

- Use small and unsigned data types, (`unsigned char` and `unsigned short`) unless your application really requires signed values.
- 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 data16 this is `int`, for far this is `int`, and for data20 it is `long`.

- Using floating-point types is very inefficient, both in terms of code size and execution speed.
- Declaring a pointer parameter to point to `const` data might open for better optimizations in the calling function.
- Try to avoid 64-bit data types, such as `double` and `long long`.

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

FLOATING-POINT TYPES

Using floating-point types on a microprocessor without a math coprocessor is very inefficient, both in terms of code size and execution speed. Thus, you should consider replacing code that uses floating-point operations with code that uses integers, because these are more efficient.

The compiler supports two floating-point formats—32 and 64 bits. The 32-bit floating-point type `float` is more efficient in terms of code size and execution speed. However, the 64-bit format `double` supports higher precision and larger numbers.

In the compiler, the floating-point type `float` always uses the 32-bit format. The format used by the `double` floating-point type depends on the setting of the `--64bit_doubles` compiler option.

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

By default, a *floating-point constant* in the source code is treated as being of the type `double`. This can cause innocent-looking expressions to be evaluated in double precision. In the example below `a` is converted from a `float` to a `double`, the `double` constant `1.0` is added and the result is converted back to a `float`:

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

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

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

For more information about floating-point types, see *Floating-point types*, page 186.

USING DIFFERENT POINTER TYPES

Neither ANSI C or ANSI C++ supports the concept of different pointer sizes. This means that whenever you use keywords or data models that construct pointers of different sizes, your code is non-compliant.

For the IAR C/C++ Compiler for M16C/R8C, the most common use of different pointer sizes is when you want to place constants in far or data20 memory, where M16C devices have ROM, and variables in data16 memory, where M16C devices have RAM.

When you do this, pointers to const declared objects will be 3-byte pointers and pointers to non-const objects will be 2-byte pointers and casting between these pointer types might not be possible.

For C, the most common problem is string literals, which are represented as `const char *`, but it is allowed to assign this to a `char *` variable.

Example

```
char * str = "This will not work";
```

For C++, the problem is larger because the use of const declared objects are much more widespread. The compiler does not allow combinations of options that use different sizes for default pointers. That is, if you use `--data_model=far`, you also need to use `--variable_data=far`.

In rare cases however, this is not enough. For some C++ constructs, the compiler might be forced to create temporary objects on the stack, which will always be in data16 memory and any pointers to these objects will be 2-byte pointers. This can cause errors when using the far or data20 data models.

ALIGNMENT OF ELEMENTS IN A STRUCTURE

The M16C/R8C Series of CPU cores can access data on odd and even addresses. However, when you access a 16-bit object on an odd address, there is a performance penalty. For this reason, the compiler can align data on even addresses (for objects larger than one byte) by using the command line option `--align_data=2`. In this case all elements in a structure will also be aligned on even addresses. This means that the compiler might need to insert *pad bytes* to keep the alignment correct. Also, to keep the stack aligned, extra code at function entry and function exit might be needed.

If you want to change the alignment on individual structures, you can override the command line option by using `#pragma data_alignment` and `#pragma pack`. For further details about the `#pragma data_alignment` directive and the `#pragma pack` directive, see *data_alignment*, page 220 and *pack*, page 226.

For information about alignment requirements, see *Alignment*, page 183.

ANONYMOUS STRUCTS AND UNIONS

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

Anonymous structures are part of the C++ language; however, they are not part of the C standard. In the IAR C/C++ Compiler for M16C/R8C they can be used in C if language extensions are enabled.

In the IDE, language extensions are enabled by default.

Use the `-e` compiler option to enable language extensions. See *-e*, page 165, for additional information.

Example

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

```
struct S
{
  char mTag;
  union
  {
    long mL;
    float mF;
  };
} St;

void F(void)
{
  St.mL = 5;
}
```

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

```
__no_init volatile
union
{
  unsigned char IOPORT;
  struct
  {
    unsigned char Way: 1;
    unsigned char Out: 1;
  };
} @ 8;
```

```
/* Here the variables are used*/

void Test(void)
{
  IOPORT = 0;
  Way = 1;
  Out = 1;
}
```

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

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

Controlling data and function placement in memory

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

- Data models

  Use the different compiler options for data models to take advantage of the different addressing modes available for the CPU core and thereby also place functions and data objects in different parts of memory. To read more about data models, see *Data models*, page 12.

- Memory attributes

  Use memory attributes to override the default addressing mode and placement of data objects. To read more about memory attributes for data, see *Using data memory attributes*, page 15.

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

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

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

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

  Use the `--code_segment` option to place functions in named segments, which is useful, for example, if you want to direct them to different fast or slow memories. To read more about the `--code_segment` option, see *--code_segment*, page 159.

At compile time, data and functions are placed in different segments as described in *Data segments*, page 35, and *Code segments*, page 42, 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 32.

DATA PLACEMENT AT AN ABSOLUTE LOCATION

The `@` operator, alternatively the `#pragma location` directive, can be used for placing global and static variables at absolute addresses. The variables must be declared using one of these combinations of keywords:

- `__no_init`
- `__no_init` and `const` (without initializers)
- `const` (with initializers).

To place a variable at an absolute address, the argument to the `@` operator and the `#pragma location` directive should be a literal number, representing the actual address. The absolute location must fulfill the alignment requirement for the variable that should be located.

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

Examples

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

```
__no_init volatile char alpha @ 0x3D0;/* OK */
```

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

```
#pragma location=0x3D2
__no_init const int beta;                /* OK */

const int gamma @ 0x3D4 = 3;             /* OK */
```

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

These examples show incorrect usage:

```
int delta @ 0x3D6;                   /* Error, neither */
                                     /* "__no_init" nor "const".*/

__no_init int epsilon @ 0x3D7;       /* Error, misaligned. */
```

C++ considerations

In C++, module scoped `const` variables are static (module local), whereas in C they are global. This means that each module that declares a certain `const` variable will contain a separate variable with this name. If you link an application with several such modules all containing (via a header file), for instance, the declaration:

```
volatile const __no_init int x @ 0x100;        /* Bad in C++ */
```

the linker will report that more than one variable is located at address `0x100`.

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

```
/* The extern keyword makes x public. */
extern volatile const __no_init int x @ 0x100;
```

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

DATA AND FUNCTION PLACEMENT IN SEGMENTS

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

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

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

If you use your own segments, in addition to the predefined segments, the segments must also be defined in the linker 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.

The location of the segments can be controlled from the linker command file.

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

Examples of placing variables in named segments

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

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

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

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

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

```
__data20 __no_init int alpha @ "NOINIT";/* Placed in data20*/
```

This example shows incorrect usage:

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

Examples of placing functions in named segments

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

void g(void) @ "FUNCTIONS"
{
}

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

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

```
__data20 void f(void) @ "FUNCTIONS";
```

Controlling compiler optimizations

The compiler performs many transformations on your application to generate the best possible code. Examples of such transformations are storing values in registers instead of memory, removing superfluous code, reordering computations in a more efficient order, and replacing arithmetic operations by cheaper operations.

The linker should also be considered an integral part of the compilation system, because some optimizations are performed by the linker. For instance, all unused functions and variables are removed and not included in the final output.

SCOPE FOR PERFORMED OPTIMIZATIONS

You can decide whether optimizations should be performed on your whole application or on individual files. By default, the same types of optimizations are used for an entire project, but you should consider using different optimization settings for individual files. For example, put code that must execute very quickly into a separate file and compile it for minimal execution time, and the rest of the code for minimal code size. This will give a small program, which is still fast enough where it matters.

You can also exclude individual functions from the performed optimizations. The `#pragma optimize` directive allows you to either lower the optimization level, or specify another type of optimization to be performed. Refer to *optimize*, page 225, for information about the pragma directive.

Multi-file compilation units

In addition to applying different optimizations to different source files or even functions, you can also decide what a compilation unit consists of—one or several source code files.

By default, a compilation unit consists of one source file, but you can also use multi-file compilation to make several source files in a compilation unit. The advantage is that interprocedural optimizations such as inlining, cross call, and cross jump have more source code to work on. Ideally, the whole application should be compiled as one compilation unit. However, for large applications this is not practical because of resource restrictions on the host computer. For more information, see *--mfc*, page 169.

If the whole application is compiled as one compilation unit, it is very useful to make the compiler also discard unused public functions and variables before the interprocedural optimizations are performed. Doing this limits the scope of the optimizations to functions and variables that are actually used. For more information, see *--discard_unused_publics*, page 164.

OPTIMIZATION LEVELS

The compiler supports different levels of optimizations. This table lists the optimizations that are performed on each level:

Optimization level	Description
None (Best debug support)	Variables live through their entire scope
Low	Same as above but variables only live for as long as they are needed, not necessarily through their entire scope Dead code elimination Redundant label elimination Redundant branch elimination
Medium	Same as above Live-dead analysis and optimization Code hoisting Register content analysis and optimization Common subexpression elimination
High (Balanced)	Same as above Peephole optimization Cross jumping (when optimizing for size) Cross call (when optimizing for size) Loop unrolling (when optimizing for speed) Function inlining Code motion Type-based alias analysis

Table 27: Compiler optimization levels

Note: Some of the performed optimizations can be individually enabled or disabled. For more information about these, see *Fine-tuning enabled transformations*, page 133.

A high level of optimization might result in increased compile time, and will most likely also make debugging more difficult, because it is less clear how the generated code relates to the source code. For example, at the low, medium, and high optimization levels, variables do not live through their entire scope, which means processor registers used for storing variables can be reused immediately after they were last used. Due to this, the C-SPY Watch window might not be able to display the value of the variable throughout its scope. At any time, if you experience difficulties when debugging your code, try lowering the optimization level.

SPEED VERSUS SIZE

At the high optimization level, the compiler balances between size and speed optimizations. However, it is possible to fine-tune the optimizations explicitly for either size or speed. They only differ in what thresholds that are used; speed will trade size for speed, whereas size will trade speed for size. Note that one optimization sometimes enables other optimizations to be performed, and an application might in some cases become smaller even when optimizing for speed rather than size.

FINE-TUNING ENABLED TRANSFORMATIONS

At each optimization level you can disable some of the transformations individually. To disable a transformation, use either the appropriate option, for instance the command line option `--no_inline`, alternatively its equivalent in the IDE **Function inlining**, or the `#pragma optimize` directive. These transformations can be disabled individually:

- Common subexpression elimination
- Loop unrolling
- Function inlining
- Code motion
- Type-based alias analysis
- Cross call.

Common subexpression elimination

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

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

To read more about the command line option, see *--no_cse*, page 171.

Loop unrolling

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

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

The compiler heuristically decides which loops to unroll. Different heuristics are used when optimizing for speed, size, or when balancing between size and speed.

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

To read more about the command line option, see *--no_unroll*, page 174.

Function inlining

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

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

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

To read more about the command line option, see *--no_inline*, page 172.

Code motion

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

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

Type-based alias analysis

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

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

Type-based alias analysis is performed at optimization level **High**. For 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 173.

Example

```
short F(short *p1, long *p2)
{
  *p2 = 0;
  *p1 = 1;
  return *p2;
}
```

With type-based alias analysis, it is assumed that a write access to the `short` pointed to by `p1` cannot affect the `long` value that `p2` points to. Thus, it is known at compile time that this function returns 0. However, in non-standard-conforming C or C++ code these pointers could overlap each other by being part of the same union. If you use explicit casts, you can also force pointers of different pointer types to point to the same memory location.

Cross call

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

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 can modify non-local variables. When the life spans for local variables end, the previously occupied memory can then be reused. Globally declared variables will occupy data memory during the whole program execution.

- Avoid taking the address of local variables using the `&` operator. This is inefficient for two main reasons. First, the variable must be placed in memory, and thus cannot be placed in a processor register. This results in larger and slower code. Second, the optimizer can no longer assume that the local variable is unaffected over function calls.
- Module-local variables—variables that are declared static—are preferred over global variables. Also avoid taking the address of frequently accessed static variables.
- The compiler is capable of inlining functions. 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 might 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 172.
- 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 87.

MEMORY TYPES

The compiler provides a range of different memory types.

For most applications it is sufficient to use the data model feature to specify the default data type. However, for some applications it might be necessary to specify other memory types in certain cases, for example:

- An application where some global variables are accessed in a large number of locations. In this case they can be declared to be placed in data16 memory (or data13, if it is a bit in a bitfield).
- An application where all data, with the exception of one large chunk of data, fits into the region of one of the smaller memory types.
- Data that must be placed at a specific memory location.

Constants and variables in different parts of memory

Placing constants and variables in different parts of memory, for example constants in far or data20 memory and variables in data16 memory is not ISO/ANSI standard compliant. These combinations are efficient to use for the IAR C/C++ Compiler for

M16C/R8C, and code written using them can be migrated to other ISO/ANSI C/C++ compilers.

However, some code written for other ISO/ANSI C/C++ compilers cannot be migrated to the IAR C/C++ Compiler for M16C/R8C because:

- Default pointers cannot point to constant objects. (Only pointers that refer to a `const` type object are capable of referring to constant objects). A special case of this is the `varargs` argument; pointers here are handled as non-`const` objects even though type-checking is limited.
- String literals are handled as `const` objects, which means that any function that has a parameter of type `char *` cannot be called directly with a string literal. There are two possible ways to solve this.

Solution 1: Define the parameter as `const char *` instead of `char *`:

```
/* Original function */
void my_func(char * str)
{
}
/* Rewritten function */
void my_func(const char * str)
{
}
/* Original function call can be used as is */
void main(void)
{
  my_func1("Hello World");
}
```

If it is not possible to change the definition of the parameter, it might be possible to use the second alternative instead.

Solution 2: Copy the string literal to a `char *` variable and call the function with the variable:

```
/* Original function can be used as is */
void my_func(char * str)
{
}
/* Copy string literal to char * variable, and call
   function with variable */
void main(void)
{
  char my_string[] = "Hello World";
  my_func(my_string);
}
```

The standard `printf` function allows the format string to be a string literal, but `varargs` arguments must still be located in non-constant memory:

```
void main(void)
{
  /* String copied to non-constant memory */
  char success_var[] = "success";

  /* String literal argument located in constant memory */
  printf("This will result in %s\n", "failure");

  /* Variable string argument located in non-constant
     memory */
  printf("This will result in %s\n", success_var);
```

Note that some of the standard C library functions have been altered to take `const char *` parameters and return `const char *` instead of `char *` whenever possible.

SAVING STACK SPACE AND RAM MEMORY

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

- If stack space is limited, avoid long call chains and recursive functions.
- Avoid using large non-scalar types, such as structures, as parameters or return type. To save stack space, you should instead pass them as pointers or, in C++, as references.

FUNCTION PROTOTYPES

It is possible to declare and define functions using one of two different styles:

- Prototyped
- Kernighan & Ritchie C (K&R C)

Both styles are included in the C standard; however, it is recommended to use the prototyped style, since it makes it easier for the compiler to find problems in the code. Using the prototyped style will also make it possible to generate more efficient code, since type promotion (implicit casting) is not needed. The K&R style is only supported for compatibility reasons.

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

Prototyped style

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

```
int Test(char, int); /* Declaration */

int Test(char ch, int i) /* Definition */
{
  return i + ch;
}
```

Kernighan & Ritchie style

In K&R style—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.

For example:

```
int Test();      /* Declaration */

int Test(ch, i) /* Definition */
char ch;
int i;
{
  return i + ch;
}
```

INTEGER TYPES AND BIT NEGATION

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

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

```
void F1(unsigned char c1)
{
  if (c1 == ~0x80)
    ;
}
```

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

PROTECTING SIMULTANEOUSLY ACCESSED VARIABLES

Variables that are accessed asynchronously, for example by interrupt routines or by code executing in separate threads, must be properly marked and have adequate protection. The only exception to this is a variable that is always *read-only*.

To mark a variable properly, use the `volatile` keyword. This informs the compiler, among other things, that the variable can be changed from other threads. The compiler will then avoid optimizing on the variable (for example, keeping track of the variable in registers), will not delay writes to it, and be careful accessing the variable only the number of times given in the source code. To read more about the `volatile` type qualifier, see *Declaring objects volatile*, page 191.

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

ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for several M16C/R8C Series devices are included in the IAR product installation. The header files are named `io`*`device`*`.h` and define the processor-specific special function registers (SFRs).

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

SFRs with bitfields are declared in the header file. This example is from `iom16c62.h`:

```
/* Timer B3,4,5 count start flag */
__data13 __no_init volatile union
{
  unsigned char TBSR;
  struct
  {
    unsigned char         : 3;
    unsigned char TB3S    : 1;
    unsigned char TB4S    : 1;
    unsigned char TB5S    : 1;
  } TBSR_bit;
} @ 0x340;
```

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:

```
#define tbsr       TBSR
#define tb3s       TBSR_bit.TB3S
#define tb4s       TBSR_bit.TB4S
#define tb5s       TBSR_bit.TB5S
```

You can also use the header files as templates when you create new header files for other M16C/R8C Series devices. For details about the @ operator, see *Located data*, page 41.

NON-INITIALIZED VARIABLES

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

The compiler supports the declaration of variables that will not be initialized, using the `__no_init` type modifier. They can be specified either as a keyword or using the `#pragma object_attribute` directive. The compiler places such variables in a separate segment, according to the specified memory keyword. See the chapter *Placing code and data* for more information.

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

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

For information about the `__no_init` keyword, see page 213. Note that to use this keyword, language extensions must be enabled; see *-e*, page 165. For information about the `#pragma object_attribute`, see page 225.

Part 2. Reference information

This part of the IAR C/C++ Compiler Reference Guide for M16C/R8C contains these chapters:

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

External interface details

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

Invocation syntax

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

COMPILER INVOCATION SYNTAX

The invocation syntax for the compiler is:

```
iccm16c [options] [sourcefile] [options]
```

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

```
iccm16c prog.c --debug
```

The source file can be a C or C++ file, typically with the filename extension `c` or `cpp`, respectively. If no filename extension is specified, the file to be compiled must have the extension `c`.

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

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

PASSING OPTIONS

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

- Directly from the command line

  Specify the options on the command line after the `iccm16c` command, either before or after the source filename; see *Invocation syntax*, page 145.

- Via environment variables

  The compiler automatically appends the value of the environment variables to every command line; see *Environment variables*, page 146.
- Via a text file, using the `-f` option; see *-f*, page 167.

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

ENVIRONMENT VARIABLES

These environment variables can be used with the compiler:

Environment variable	Description
C_INCLUDE	Specifies directories to search for include files; for example: `C_INCLUDE=c:\program files\iar systems\embedded workbench 5.`*n*`\m16c\inc;c:\headers`
QCCM16C	Specifies command line options; for example: `QCCM16C=-lA asm.lst`

Table 28: Compiler environment variables

Include file search procedure

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

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

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

  it searches these directories for the file to include:

  1 The directories specified with the `-I` option, in the order that they were specified, see *-I*, page 167.

  2 The directories specified using the `C_INCLUDE` environment variable, if any, see *Environment variables*, page 146.
- If the compiler encounters the name of an `#include` file in double quotes, for example:

  ```
  #include "vars.h"
  ```

  it searches the directory of the source file in which the `#include` statement occurs, and then performs the same sequence as for angle-bracketed filenames.

If there are nested `#include` files, the compiler starts searching the directory of the file that was last included, iterating upwards for each included file, searching the source file directory last. For example:

```
src.c in directory dir\src
  #include "src.h"
  ...
src.h in directory dir\include
  #include "config.h"
  ...
```

When `dir\exe` is the current directory, use this command for compilation:

```
iccm16c ..\src\src.c -I..\include -I..\debugconfig
```

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

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

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

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

Compiler output

The compiler can produce the following output:

- A linkable object file

  The object files produced by the compiler use a proprietary format called UBROF, which stands for Universal Binary Relocatable Object Format. By default, the object file has the filename extension `r34`.
- Optional list files

  Various kinds of list files can be specified using the compiler option `-l`, see *-l*, page 168. By default, these files will have the filename extension `lst`.
- Optional preprocessor output files

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

- Diagnostic messages

  Diagnostic messages are directed to the standard error stream and displayed on the screen, and printed in an optional list file. To read more about diagnostic messages, see *Diagnostics*, page 148.
- 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 148.
- Size information

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

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

Error return codes

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

These command line error codes are supported:

Code	Description
0	Compilation successful, but there might have been warnings.
1	Warnings were produced and the option `--warnings_affect_exit_code` was used.
2	Errors occurred.
3	Fatal errors occurred, making the compiler abort.
4	Internal errors occurred, making the compiler abort.

Table 29: Error return codes

Diagnostics

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

MESSAGE FORMAT

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

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

with these elements:

filename	The name of the source file in which the issue was encountered
linenumber	The line number at which the compiler detected the issue
level	The level of seriousness of the issue
tag	A unique tag that identifies the diagnostic message
message	An explanation, possibly several lines long

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

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

SEVERITY LEVELS

The diagnostic messages are divided into different levels of severity:

Remark

A diagnostic message that is produced when the compiler finds a source code construction that can possibly lead to erroneous behavior in the generated code. Remarks are by default not issued, but can be enabled, see *--remarks*, page 178.

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 174.

Error

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

Fatal error

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

SETTING THE SEVERITY LEVEL

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

See *Summary of compiler options*, page 153, 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 was a serious and unexpected failure due to a fault in the compiler. It is produced using this form:

```
Internal error: message
```

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

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

Compiler options

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

Options syntax

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

Refer to the *IAR Embedded Workbench® IDE User Guide* for information about the compiler options available in the IDE and how to set them.

TYPES OF OPTIONS

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

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

For information about the different methods for passing options, see *Passing options*, page 145.

RULES FOR SPECIFYING PARAMETERS

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

Rules for optional parameters

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

`-O` or `-Oh`

For options with a long name and an optional parameter, any parameter should be specified with a preceding equal sign (=), for example:

```
--diag_suppress=n
```

Rules for mandatory parameters

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

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

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

```
--diagnostics_tables=MyDiagnostics.lst
```

or

```
--diagnostics_tables MyDiagnostics.lst
```

Rules for options with both optional and mandatory parameters

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

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

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

```
-lA MyList.lst
```

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

```
--preprocess=n PreprocOutput.lst
```

Rules for specifying a filename or directory as parameters

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

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

  ```
  iccm16c prog.c -l ..\listings\List.lst
  ```

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

  ```
  iccm16c prog.c -l ..\listings\
  ```

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

  ```
  iccm16c prog.c -l .
  ```
- `/` can be used instead of `\` as the directory delimiter.
- By specifying `-`, input files and output files can be redirected to the standard input and output stream, respectively. For example:

  ```
  iccm16c prog.c -l -
  ```

Additional rules

These rules also apply:

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

  ```
  iccm16c prog.c -l ---r
  ```
- For options that accept multiple arguments of the same type, the arguments can be provided as a comma-separated list (without a space), for example:

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

  Alternatively, the option can be repeated for each argument, for example:

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

Summary of compiler options

This table summarizes the compiler command line options:

Command line option	Description
`-2`	An alias for `--64bit_doubles`, available for backward compatibility.
`--64bit_doubles`	Sets the the size of doubles to 64 bits

Table 30: Compiler options summary

Command line option	Description
`--align_data`	Specifies byte alignment for data
`--align_func`	Specifies byte alignment for functions
`--calling_convention`	Specifies the calling convention
`--char_is_signed`	Treats `char` as signed
`--code_segment`	Places executable code in a specified segment
`--constant_data`	Overrides the default placement of constants
`--cpu`	Specifies a specific CPU core
`-D`	Defines preprocessor symbols
`--data_model`	Specifies the data model
`--debug`	Generates debug information
`--dependencies`	Lists file dependencies
`--diag_error`	Treats these as errors
`--diag_remark`	Treats these as remarks
`--diag_suppress`	Suppresses these diagnostics
`--diag_warning`	Treats these as warnings
`--diagnostics_tables`	Lists all diagnostic messages
`--discard_unused_publics`	Discards unused public symbols
`--dlib_config`	Determines the library configuration file
`-e`	Enables language extensions
`--ec++`	Enables Embedded C++ syntax
`--eec++`	Enables Extended Embedded C++ syntax
`--enable_multibytes`	Enables support for multibyte characters in source files
`--error_limit`	Specifies the allowed number of errors before compilation stops
`-f`	Extends the command line
`--header_context`	Lists all referred source files and header files
`-I`	Specifies include file path
`-l`	Creates a list file
`--library_module`	Creates a library module
`--low_consts`	Copies constants to near RAM
`--mfc`	Enables multi-file compilation

Table 30: Compiler options summary (Continued)

Command line option	Description
`--migration_preprocessor_extensions`	Extends the preprocessor
`--misrac`	Enables error messages specific to MISRA C:1998. This option is a synonym of `--misrac1998` and is only available for backward compatibility.
`--misrac1998`	Enables error messages specific to MISRA-C:1998. See the *IAR Embedded Workbench® MISRA C:1998 Reference Guide*.
`--misrac2004`	Enables error messages specific to MISRA-C:2004. See the *IAR Embedded Workbench® MISRA C:2004 Reference Guide*.
`--misrac_verbose`	Enables verbose logging of MISRA C checking. See the *IAR Embedded Workbench® MISRA C:1998 Reference Guide* or the *IAR Embedded Workbench® MISRA C:2004 Reference Guide*.
`--module_name`	Sets the object module name
`--no_code_motion`	Disables code motion optimization
`--no_cross_call`	Disables cross-call optimization
`--no_cse`	Disables common subexpression elimination
`--no_inline`	Disables function inlining
`--no_path_in_file_macros`	Removes the path from the return value of the symbols `__FILE__` and `__BASE_FILE__`
`--no_tbaa`	Disables type-based alias analysis
`--no_typedefs_in_diagnostics`	Disables the use of typedef names in diagnostics
`--no_unroll`	Disables loop unrolling
`--no_warnings`	Disables all warnings
`--no_wrap_diagnostics`	Disables wrapping of diagnostic messages
`-O`	Sets the optimization level
`-o`	Sets the object filename
`--omit_types`	Excludes type information
`--only_stdout`	Uses standard output only
`--output`	Sets the object filename
`--predef_macros`	Lists the predefined symbols.

Table 30: Compiler options summary (Continued)

Command line option	Description
`--preinclude`	Includes an include file before reading the source file
`--preprocess`	Generates preprocessor output
`--public_equ`	Defines a global named assembler label
`-R`	An alias for `--code_segment`, available for backward compatibility.
`-r`	Generates debug information
`--remarks`	Enables remarks
`--require_prototypes`	Verifies that functions are declared before they are defined
`-s`	Optimizes for speed, available for backward compatibility
`--silent`	Sets silent operation
`--strict_ansi`	Checks for strict compliance with ISO/ANSI C
`-u`	An alias for `--align_data`, available for backward compatibility.
`--use_DIV`	Uses `div` and `divu`
`--variable_data`	Specifies an explicit location for variables
`--warnings_affect_exit_code`	Warnings affects exit code
`--warnings_are_errors`	Warnings are treated as errors
`-y`	An alias for `--low_consts`, available for backward compatibility.
`-z`	Optimizes for size, available for backward compatibility

Table 30: Compiler options summary (Continued)

Descriptions of options

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

Note that if you use the options page **Extra Options** to specify specific command line options, the IDE does not perform an instant check for consistency problems like conflicting options, duplication of options, or use of irrelevant options.

--64bit_doubles

Syntax `--64bit_doubles`

Description By default, the compiler uses 32-bit doubles. Use this option to set the size of doubles to 64 bits instead.

See also *Floating-point types*, page 186.

Project>Options>General Options>Target>Floating-point

--align_data

Syntax `--align_data={1|2}`

Parameters

`1`	Specifies byte alignment
`2` (default)	Specifies word alignment

Description By default, the compiler uses word alignment for data objects. Using word alignment implies efficient memory accesses, but at the expense of wasted data memory for the padded bytes. In some cases the code size is increased when using word alignment. If space is a critical matter, choose byte alignment instead.

Project>Options>General Options>Target>Byte align objects

--align_func

Syntax `--align_func={1|2}`

Parameters

`1` (default)	Specifies byte alignment
`2`	Specifies word alignment

Description By default, the compiler uses byte alignment for function entries. Use `--align_func=2` to specify word alignment and force the compiler to align all function entries to even addresses. Using `--align_func=1` can save 1 byte of code space, but execution speed might vary depending on whether the start address is odd or even.

The M16C/R8C Series of CPU cores accesses code faster on even addresses, which means that a function might have a different execution performance if it starts on an odd or an even address. If the execution speed of the function should not vary depending on the size of other functions, you should use `--align_func=2`. However, the compiler will add a pad byte on all functions with an odd size.

Project>Options>General Options>Target>Word align function entries

--calling_convention

Syntax

`--calling_convention={simple|normal}`

Parameters

`simple`	Specifies the simple calling convention
`normal` (default)	Specifies the normal calling convention

Description

Use this option to override the default calling convention.

Note: The `--calling_convention` option must be specified in the same way in all modules.

See also

Calling convention, page 93.

Project>Options>General Options>Target>Calling convention

--char_is_signed

Syntax

`--char_is_signed`

Description

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

Note: The runtime library is compiled without the `--char_is_signed` option. If you use this option, you might get type mismatch warnings from the linker, because the library uses `unsigned char`.

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

--code_segment

Syntax

```
--code_segment=name
```

Parameters

name	Specifies the segment where code is to be placed

Description

Normally, the compiler places executable code in the segment named `CODE`. If you want to specify an explicit location for the code, use this option to specify a code segment name, which you can then assign to a fixed address in memory by modifying the linker command file.

Note: The segment name is case-sensitive.

Project>Options>C/C++ Compiler>Output>Code segment

--constant_data

Syntax

```
--constant_data={near|far|huge}
```

Parameters

`near` (default)	Places constants in near memory
`far`	Places constants in far memory
`huge`	Places constants in huge memory

Description

The default placement of constant data is determined by the used data model. Use this option to override the default placement for constants.

See also

Data models, page 12, *--variable_data*, page 180, and *Constants and variables in different parts of memory*, page 136..

Project>Options>General Options>Target>Constants in:

--cpu

Syntax

```
--cpu=core
```

Parameters

core	Specifies a specific CPU core; choose between `M16C` (default) and `R8C`.

Description

The compiler supports different CPU cores. Use this option to select for which CPU core the code will be generated.

Project>Options>General Options>Target>Device

-D

Syntax

`-D symbol[=value]`

Parameters

`symbol`	The name of the preprocessor symbol
`value`	The value of the preprocessor symbol

Description

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

The option `-D` has the same effect as a `#define` statement at the top of the source file:

```
-Dsymbol
```

is equivalent to:

```
#define symbol 1
```

To get the equivalence of:

```
#define FOO
```

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

```
-DFOO=
```

Project>Options>C/C++ Compiler>Preprocessor>Defined symbols

--data_model

Syntax

`--data_model={near|far|huge}`

Parameters

`near` (default)	Specifies the near data model
`far`	Specifies the far data model
`huge`	Specifies the huge data model

Description

Use this option to select the data model for which the code will be generated. If you do not select a data model option, the compiler uses the default data model. Note that all modules of your application must use the same data model.

See also

Data models, page 12.

Project>Options>General Options>Target>Data model

--debug, -r

Syntax

```
--debug
-r
```

Description

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

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

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

--dependencies

Syntax

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

Parameters

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

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 152.

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.r34: c:\iar\product\include\stdio.h
foo.r34: 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:

```
%.r34 : %.c
      $(ICC) $(ICCFLAGS) $< --dependencies=m $*.d
```

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

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

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

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

This option is not available in the IDE.

--diag_error

Syntax `--diag_error=tag[,tag,...]`

Parameters

tag	The number of a diagnostic message, for example the message number `Pe117`

Description

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

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

--diag_remark

Syntax `--diag_remark=tag[,tag,...]`

Parameters

tag	The number of a diagnostic message, for example the message number `Pe177`

Description

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

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

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

--diag_suppress

Syntax `--diag_suppress=tag[,tag,...]`

Parameters

tag	The number of a diagnostic message, for example the message number `Pe117`

Description

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

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

--diag_warning

Syntax `--diag_warning=tag[,tag,...]`

Parameters

tag	The number of a diagnostic message, for example the message number `Pe826`

Description

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

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

--diagnostics_tables

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

Parameters For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 214.

Description Use this option to list all possible diagnostic messages in a named file. This can be convenient, for example, if you have used a pragma directive to suppress or change the severity level of any diagnostic messages, but forgot to document why.

This option cannot be given together with other options.

This option is not available in the IDE.

--discard_unused_publics

Syntax `--discard_unused_publics`

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

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

Note: Do not use this option only on parts of the application, as necessary symbols might be removed from the generated output.

See also *--mfc*, page 169 and *Multi-file compilation units*, page 131.

Project>Options>C/C++ Compiler>Discard unused publics

--dlib_config

Syntax `--dlib_config` *`filename`*

Parameters For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 152.

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

All prebuilt runtime libraries are delivered with corresponding configuration files. You can find the library object files and the library configuration files in the directory `m16c\lib\dlib`. For examples and a list of prebuilt runtime libraries, see *Using a prebuilt library*, page 48.

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 57.

Note: This option only applies to the IAR DLIB runtime environment.

To set related options, choose:

Project>Options>General Options>Library Configuration

-e

Syntax `-e`

Description In the command line version of the compiler, language extensions are disabled by default. If you use language extensions such as extended keywords and anonymous structs and unions in your source code, you must use this option to enable them.

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

See also The chapter *Compiler extensions.*

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

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

--ec++

Syntax	`--ec++`
Description	In the compiler, the default language is C. If you use Embedded C++, you must use this option to set the language the compiler uses to Embedded C++.

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

--eec++

Syntax	`--eec++`
Description	In the compiler, the default language is C. If you take advantage of Extended Embedded C++ features like namespaces or the standard template library in your source code, you must use this option to set the language the compiler uses to Extended Embedded C++.
See also	*Extended Embedded C++*, page 110.

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

--enable_multibytes

Syntax	`--enable_multibytes`
Description	By default, multibyte characters cannot be used in C or C++ source code. Use this option to make multibyte characters in the source code be interpreted according to the host computer's default setting for multibyte support. Multibyte characters are allowed in C and C++ style comments, in string literals, and in character constants. They are transferred untouched to the generated code.

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

--error_limit

Syntax	`--error_limit=`*`n`*
Parameters	

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

Description

Use the `--error_limit` option to specify the number of errors allowed before the compiler stops the compilation. By default, 100 errors are allowed.

This option is not available in the IDE.

-f

Syntax

`-f filename`

Parameters

For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 214.

Descriptions

Use this option to make the compiler read command line options from the named file, with the default filename extension `xcl`.

In the command file, you format the items exactly as if they were on the command line itself, except that you may use multiple lines, because the newline character acts just as a space or tab character.

Both C and C++ style comments are allowed in the file. Double quotes behave in the same way as in the Microsoft Windows command line environment.

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

--header_context

Syntax

`--header_context`

Description

Occasionally, to find the cause of a problem it is necessary to know which header file that was included from which source line. Use this option to list, for each diagnostic message, not only the source position of the problem, but also the entire include stack at that point.

This option is not available in the IDE.

-I

Syntax

`-I path`

Parameters

path	The search path for `#include` files

Description

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

See also

Include file search procedure, page 146.

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

-l

Syntax

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

Parameters

`a`	Assembler list file
`A`	Assembler list file with C or C++ source as comments
`b`	Basic assembler list file. This file has the same contents as a list file produced with `-la`, except that no extra compiler-generated information (runtime model attributes, call frame information, frame size information) is included *
`B`	Basic assembler list file. This file has the same contents as a list file produced with `-lA`, except that no extra compiler generated information (runtime model attributes, call frame information, frame size information) is included *
`c`	C or C++ list file
`C` (default)	C or C++ list file with assembler source as comments
`D`	C or C++ list file with assembler source as comments, but without instruction offsets and hexadecimal byte values
`N`	No diagnostics in file
`H`	Include source lines from header files in output. Without this option, only source lines from the primary source file are included

*** This makes the list file less useful as input to the assembler, but more useful for reading by a human.**

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 152.

Description

Use this option to generate an assembler or C/C++ listing to a file. Note that this option can be used one or more times on the command line.

To set related options, choose:

Project>Options>C/C++ Compiler>List

--library_module

Syntax

```
--library_module
```

Description

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

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

--low_consts

Syntax

```
--low_consts
```

Description

Use this option to copy constants to near RAM from ROM. All constants will be treated as initialized variables. This simplifies the use of the near data model for projects with no ROM in data16 memory—for instance projects with no external ROM, as all M16C cores have their internal ROM outside data16 memory.

Note: This option is only useful in the near data model.

Project>Options>General Options>Target>Writable constants

--mfc

Syntax

```
--mfc
```

Description

Use this option to enable *multi-file compilation*. This means that the compiler compiles one or several source files specified on the command line as one unit, which makes interprocedural optimizations such as inlining, cross call, and cross jump possible.

Note: The compiler will generate one object file per input source code file, where the first object file contains all relevant data and the other ones are empty. If you want only the first file to be produced, use the `-o` compiler option and specify a certain output file.

Example `iccm16c myfile1.c myfile2.c myfile3.c --mfc`

See also *--discard_unused_publics*, page 164, *-o, --output*, page 175, and *Multi-file compilation units*, page 131.

Project>Options>C/C++ Compiler>Multi-file compilation

--migration_preprocessor_extensions

Syntax `--migration_preprocessor_extensions`

Description If you need to migrate code from an earlier IAR Systems C or C/C++ compiler, you might 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 does not conform to the ISO/ANSI C standard, but it will also reject some code that *does* conform to the standard.

Important! Do not depend on these extensions in newly written code, because support for them might be removed in future compiler versions.

Project>Options>C/C++ Compiler>Language>Enable IAR migration preprocessor extensions

--module_name

Syntax `--module_name=`*`name`*

Parameters

`name`	An explicit object module name

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

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

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

--no_code_motion

Syntax `--no_code_motion`

Description Use this option to disable code motion optimizations. These optimizations, which are performed at the optimization levels Medium and High, normally reduce code size and execution time. However, the resulting code might be difficult to debug.

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

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Code motion

--no_cross_call

Syntax `--no_cross_call`

Description Use this option to disable the cross-call optimization. This optimization is performed at size optimization, level High. Note that, although the option can drastically reduce the code size, this option increases the execution time.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Cross call

--no_cse

Syntax `--no_cse`

Description Use this option to disable common subexpression elimination. At the optimization levels Medium and High, the compiler avoids calculating the same expression more than once. This optimization normally reduces both code size and execution time. However, the resulting code might be difficult to debug.

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

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

--no_inline

Syntax `--no_inline`

Description Use this option to disable function inlining. Function inlining means that a simple function, whose definition is known at compile time, is integrated into the body of its caller to eliminate the overhead of the call.

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

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

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

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Function inlining

--no_path_in_file_macros

Syntax `--no_path_in_file_macros`

Description Use this option to exclude the path from the return value of the predefined preprocessor symbols `__FILE__` and `__BASE_FILE__`.

See also *Descriptions of predefined preprocessor symbols*, page 242.

This option is not available in the IDE.

--no_tbaa

Syntax `--no_tbaa`

Description Use this option to disable type-based alias analysis. When this options is not used, the compiler is free to assume that objects are only accessed through the declared type or through `unsigned char`.

See also *Type-based alias analysis*, page 134.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Type-based alias analysis

--no_typedefs_in_diagnostics

Syntax `--no_typedefs_in_diagnostics`

Description Use this option to disable the use of typedef names in diagnostics. Normally, when a type is mentioned in a message from the compiler, most commonly in a diagnostic message of some kind, the typedef names that were used in the original declaration are used whenever they make the resulting text shorter.

Example

```
typedef int (*MyPtr)(char const *);
MyPtr p = "foo";
```

will give an error message like this:

```
  Error[Pe144]: a value of type "char *" cannot be used to
  initialize an entity of type "MyPtr"
```

If the `--no_typedefs_in_diagnostics` option is used, the error message will be like this:

```
  Error[Pe144]: a value of type "char *" cannot be used to
  initialize an entity of type "int (*)(char const *)"
```

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

--no_unroll

Syntax

```
--no_unroll
```

Description

Use this option to disable loop unrolling. The code body of a small loop, whose number of iterations can be determined at compile time, is duplicated to reduce the loop overhead.

For small loops, the overhead required to perform the looping can be large compared with the work performed in the loop body.

The loop unrolling optimization duplicates the body several times, reducing the loop overhead. The unrolled body also opens up for other optimization opportunities.

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

The compiler heuristically decides which loops to unroll. Different heuristics are used when optimizing for speed and size.

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

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Loop unrolling

--no_warnings

Syntax

```
--no_warnings
```

Description

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

This option is not available in the IDE.

--no_wrap_diagnostics

Syntax

```
--no_wrap_diagnostics
```

Description

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

This option is not available in the IDE.

-O

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

Parameters

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

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

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

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

See also *Controlling compiler optimizations*, page 131.

Project>Options>C/C++ Compiler>Optimizations

-o, --output

Syntax

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

Parameters For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 214.

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 `r34`. Use this option to explicitly specify a different output filename for the object code output.

This option is not available in the IDE.

--omit_types

Syntax `--omit_types`

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

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

--only_stdout

Syntax `--only_stdout`

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

This option is not available in the IDE.

--output, -o

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

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

Description By default, the object code output produced by the compiler is located in a file with the same name as the source file, but with the extension `r34`. Use this option to explicitly specify a different output filename for the object code output.

This option is not available in the IDE.

--predef_macros

Syntax `--predef_macros {filename|directory}`

Parameters For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 152.

Description Use this option to list the predefined symbols. When using this option, make sure to also use the same options as for the rest of your project.

If a filename is specified, the compiler stores the output in that file. If a directory is specified, the compiler stores the output in that directory, in a file with the `predef` filename extension.

This option is not available in the IDE.

--preinclude

Syntax `--preinclude includefile`

Parameters For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 152.

Description Use this option to make the compiler include the specified include file before it starts to read the source file. This is useful if you want to change something in the source code for the entire application, for instance if you want to define a new symbol.

Project>Options>C/C++ Compiler>Preprocessor>Preinclude file

--preprocess

Syntax `--preprocess[=[c][n][l]] {filename|directory}`

Parameters

`c`	Preserve comments
`n`	Preprocess only
`l`	Generate `#line` directives

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 152.

Description

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

Project>Options>C/C++ Compiler>Preprocessor>Preprocessor output to file

--public_equ

Syntax

`--public_equ symbol[=value]`

Parameters

symbol	The name of the assembler symbol to be defined
value	An optional value of the defined assembler symbol

Description

This option is equivalent to defining a label in assembler language using the `EQU` directive and exporting it using the `PUBLIC` directive. This option can be used more than once on the command line.

This option is not available in the IDE.

-r, --debug

Syntax

```
-r
--debug
```

Description

Use the `-r` or the `--debug` option to make the compiler include information in the object modules required by 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

Syntax

`--remarks`

Description

The least severe diagnostic messages are called remarks. A remark indicates a source code construct that may cause strange behavior in the generated code. By default, the compiler does not generate remarks. Use this option to make the compiler generate remarks.

See also

Severity levels, page 209.

Project>Options>C/C++ Compiler>Diagnostics>Enable remarks

--require_prototypes

Syntax

```
--require_prototypes
```

Description

Use this option to force the compiler to verify that all functions have proper prototypes. Using this option means that code containing any of the following will generate an error:

- A function call of a function with no declaration, or with a Kernighan & Ritchie C declaration
- A function definition of a public function with no previous prototype declaration
- An indirect function call through a function pointer with a type that does not include a prototype.

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

Project>Options>C/C++ Compiler>Language>Require prototypes

--silent

Syntax

```
--silent
```

Description

By default, the compiler issues introductory messages and a final statistics report. Use this option to make the compiler operate without sending these messages to the standard output stream (normally the screen).

This option does not affect the display of error and warning messages.

This option is not available in the IDE.

--strict_ansi

Syntax

```
--strict_ansi
```

Description

By default, the compiler accepts a relaxed superset of ISO/ANSI C/C++, see *Minor language extensions*, page 201. Use this option to ensure that the program conforms to the ISO/ANSI C/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

--use_DIV

Syntax

`--use_DIV`

Description

The `div` and `divu` instructions are efficient division instructions where the dividend is a 32-bit or 16-bit value and the divisor is a 16-bit or 8-bit value. However, the result of the instruction is undefined if it cannot be contained in a 16-bit value. This makes it impossible for the compiler to generate `div.w`, `divu.w`, `div.b`, and `divu.b` instructions while still being ANSI-compliant. If you know for certain that your divisions will never overflow, use this option to make the compiler generate `div.w`, `divu.w`, `div.b`, and `divu.b` instructions for divisions where either:

- the dividend is a 4-byte value, the divisor is a 2-byte value, and the result is a 2-byte value

or

- the dividend is a 2-byte value, the divisor is a 1-byte value, and the result is a 1-byte value.

Note: This is very unsafe unless you are confident that the division will never overflow. If the result is too large, it will be undetermined.

Project>Options>C/C++ Compiler>Optimizations>Use DIV and DIVU (non-ANSI)

--variable_data

Syntax

`--variable_data={near|far|huge}`

Parameters

`near` (default)	Places variables in near memory
`far`	Places variables in huge memory
`huge`	Places variables in huge memory

Description

By default, the compiler places variable data in near memory. Use this option to override the default placement of variables.

See also

--*constant_data*, page 159, *Constants and variables in different parts of memory*, page 136.

Project>Options>General Options>Target>Variables in:

--warnings_affect_exit_code

Syntax

`--warnings_affect_exit_code`

Description

By default, the exit code is not affected by warnings, because only errors produce a non-zero exit code. With this option, warnings will also generate a non-zero exit code.

This option is not available in the IDE.

--warnings_are_errors

Syntax

`--warnings_are_errors`

Description

Use this option to make the compiler treat all warnings as errors. If the compiler encounters an error, no object code is generated. Warnings that have been changed into remarks are not treated as errors.

Note: Any diagnostic messages that have been reclassified as warnings by the option `--diag_warning` or the `#pragma diag_warning` directive will also be treated as errors when `--warnings_are_errors` is used.

See also

--*diag_warning*, page 225.

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

Data representation

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

See the chapter *Efficient coding for embedded applications* for information about which data types and pointers provide the most efficient code for your application.

Alignment

Every C data object has an alignment that controls how the object can be stored in memory. Should an object have an alignment of, for example, 4, it must be stored on an address that is divisible by 4.

The reason for the concept of alignment is that some processors have hardware limitations for how the memory can be accessed.

Assume that a processor can read 4 bytes of memory using one instruction, but only when the memory read is placed on an address divisible by 4. Then, 4-byte objects, such as `long` integers, will have alignment 4.

Another processor might only be able to read 2 bytes at a time; in that environment, the alignment for a 4-byte `long` integer might be 2.

A structure type will have the same alignment as the structure member with the most strict alignment. To decrease the alignment requirements on the structure and its members, use `#pragma pack`.

All data types must have a size that is a multiple of their alignment. Otherwise, only the first element of an array would be guaranteed to be placed in accordance with the alignment requirements. This means that the compiler might add pad bytes at the end of the structure. For more information about pad bytes, see *Packed structure types*, page 190.

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

ALIGNMENT ON THE M16C/R8C SERIES OF CPU CORES

With an alignment of 2, all objects with a size of 2 bytes or more are stored at addresses divisible by 2. If you set the alignment to 1 by specifying the option `--align_data=1`, there is no such requirement.

Basic data types

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

INTEGER TYPES

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

Data type	Size	Range	Alignment
bool	8 bits	0 to 1	1
char	8 bits	0 to 255	1
signed char	8 bits	-128 to 127	1
unsigned char	8 bits	0 to 255	1
signed short	16 bits	-32768 to 32767	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 31: Integer types

Note: Integer types with a size of 16 bits or more have a default alignment of 2. This can be changed using the option `--align_data`.

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

Bool

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

The long long type

The `long long` data type is supported with these restrictions:

- The CLIB runtime library does not support the `long long` type
- A `long long` variable cannot be used in a switch statement.

The enum type

The compiler will use the smallest type required to hold `enum` constants, preferring `signed` rather than `unsigned`.

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

To make the compiler use a larger type than it would automatically use, define an `enum` constant with a large enough value. For example:

```
/* Disables usage of the char type for enum */
enum Cards{Spade1, Spade2,
           DontUseChar=257};
```

The char type

The `char` type is by default unsigned in the compiler, but the `--char_is_signed` compiler option allows you to make it signed. Note, however, that the library is compiled with the `char` type as unsigned.

The wchar_t type

The `wchar_t` data type is an integer type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locals.

The `wchar_t` data type is supported by default in the C++ language. To use the `wchar_t` type also in C source code, you must include the file `stddef.h` from the runtime library.

Note: The IAR CLIB Library has only rudimentary support for `wchar_t`.

Bitfields

In ISO/ANSI C, `int` and `unsigned int` can be used as the base type for integer bitfields. In the IAR C/C++ Compiler for M16C/R8C, 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.

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

Floating-point types

In the IAR C/C++ Compiler for M16C/R8C, floating-point values are represented in standard IEEE 754 format. The sizes for the different floating-point types are:

Type	Size if --64bit_doubles is not used	Size if --64bit_doubles is used
float	32 bits	32 bits
double	32 bits (default)	64 bits
long double	32 bits	64 bits

Table 32: Floating-point types

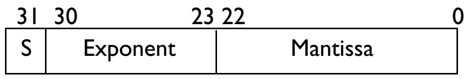
Note: The size of `double` and `long double` depends on the `--64bit_doubles` option, see *--64bit_doubles*, page 157. The type `long double` uses the same precision as `double`.

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:

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

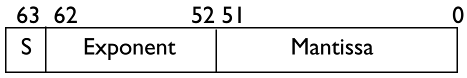
The range of the number is:

`±1.18E-38 to ±3.39E+38`

The precision of the float operators (`+`, `-`, `*`, and `/`) is approximately 7 decimal digits.

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:

`±2.23E-308 to ±1.79E+308`

The precision of the float operators (`+`, `-`, `*`, and `/`) is approximately 15 decimal digits.

Representation of special floating-point numbers

This list describes the representation of special floating-point numbers:

- Zero is represented by zero mantissa and exponent. The sign bit signifies positive or negative zero.
- Infinity is represented by setting the exponent to the highest value and the mantissa to zero. The sign bit signifies positive or negative infinity.
- Not a number (`NaN`) is represented by setting the exponent to the highest positive value and the mantissa to a non-zero value. The value of the sign bit is ignored.

Note: The IAR CLIB Library does not fully support the special cases of floating-point numbers, such as infinity, and `NaN`. A library function which gets one of these special cases of floating-point numbers as an argument might behave unexpectedly.

Pointer types

The compiler has two basic types of pointers: code pointers and data pointers.

SIZE

The size of code pointers is always 20 bits, with a storage size of 4 bytes, and they can address the entire memory. The internal representation of a code pointer is the actual address it refers to.

These data pointers are available:

Keyword	Pointer size	Storage in bytes	Description
`__data16`	16 bits	2	Can only point into 0-64 Kbytes
`__far`	20 bits 16 bits for index pointers	4	Element pointed at must be inside a 64-Kbyte page
`__data20`	20 bits 20 bits for index pointers	4	No restrictions

Table 33: Data pointers

CASTING

Casts between pointers have these characteristics:

- Casting a *value* of an integer type to a pointer of a smaller type is performed by truncation
- Casting a *value* of a signed integer type to a pointer of a larger type is performed in two steps. In the first step, the value is sign extended to `int` size. In the second step, the int value is zero extended to pointer size. In practice, both steps are performed when casting from a signed char to all pointer types except for data16 pointers.
- Casting a *pointer type* to a smaller integer type is performed by truncation
- Casting a *pointer type* to a larger integer type is performed by zero extension
- Casting a *data pointer* to a function pointer and vice versa is illegal
- Casting a *function pointer* to an integer type gives an undefined result
- Casting from a smaller pointer to a larger pointer is performed by zero extension.
- Casting from a larger pointer to a smaller pointer is illegal.

size_t

`size_t` is the unsigned integer type required to hold the maximum size of an object. In the IAR C/C++ Compiler for M16C/R8C, the size of `size_t` is 2 bytes for the near and far data models, and 4 bytes for 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 IAR C/C++ Compiler for M16C/R8C, the size of `ptrdiff_t` is 2 bytes for the near and far data models, and 4 bytes for 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 IAR C/C++ Compiler for M16C/R8C, the size of `intptr_t` is 2 bytes for the near data model, and 4 bytes for the far and huge data models.

uintptr_t

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

Note: The sizes of `size_t`, `ptrdiff_t`, `intptr_t`, and `uintptr_t` are not standard compliant for the near data model when constants are placed in far or data20 memory. See *--constant_data*, page 159, for a discussion about this.

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 have the same alignment as the member with the highest alignment requirement. The size of a `struct` is also adjusted to allow arrays of aligned structure objects.

GENERAL LAYOUT

Members of a `struct` are always allocated in the order specified in the declaration. Each member is placed in the `struct` according to the specified alignment (offsets).

Example

```
struct First
{
  char c;
  short s;
} s;
```

This diagram shows the layout in memory:

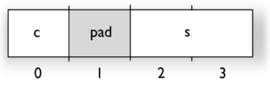


Figure 5: Structure layout

The alignment of the structure is 2 bytes, and a pad byte must be inserted to give `short s` the correct alignment.

PACKED STRUCTURE TYPES

The #pragma pack directive is used for relaxing the alignment requirements of the members of a structure. This changes the layout of the structure. The members are placed in the same order as when declared, but there might be less pad space between members.

Note that accessing an object that is not correctly aligned requires code that is both larger and slower. If such structure members are accessed many times, it is usually better to construct the correct values in a struct that is not packed, and access this struct instead.

Special care is also needed when creating and using pointers to misaligned members. For direct access to misaligned members in a packed struct, the compiler will emit the correct (but slower and larger) code when needed. However, when a misaligned member is accessed through a pointer to the member, the normal (smaller and faster) code is used. In the general case, this will not work.

Example

This example declares a packed structure:

```
#pragma pack(1)
struct S
{
  char c;
  short s;
};

#pragma pack()
```

In this example, the structure S has this memory layout:

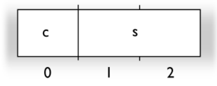


Figure 6: Packed structure layout

This example declares a new non-packed structure, S2, that contains the structure s declared in the previous example:

```
struct S2
{
  struct S s;
  long l;
};
```

`S2` has this memory layout

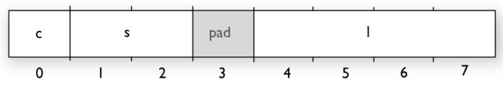


Figure 7: Packed structure layout

The structure `S` will use the memory layout, size, and alignment described in the previous example. The alignment of the member `l` is 2 (if `--align_data=2`), which means that alignment of the structure `S2` will become 2.

For more information, see *Alignment of elements in a structure*, page 125.

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 declared `volatile` as an access
- The unit for the access is either the entire object or, for accesses to an element in a composite object—such as an array, struct, class, or union—the element. For example:

```
char volatile a;
a = 5;   /* A write access */
a += 6;  /* First a read then a write access */
```

- An access to a bitfield is treated as an access to the underlaying type.

However, these rules are not detailed enough to handle the hardware-related requirements. The rules specific to the IAR C/C++ Compiler for M16C/R8C are described below.

Rules for accesses

In the IAR C/C++ Compiler for M16C/R8C, accesses to `volatile` declared objects are subject to these rules:

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

The compiler adheres to these rules for these combinations of memory types and data types:

Data type	Access rules
8 bits	Preserved, complete, same order, atomic.
16 bits	Preserved, complete, same order, atomic.
20 bits	Preserved, complete, same order.
32 bits	Preserved, complete, same order.
64 bits	Preserved, complete, same order.

Table 34: Volatile accesses

For all combinations of object types not listed, only the rule that states that all accesses are preserved 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` are allocated in ROM, unless the option `--low_consts` is used. The objects are allocated in RAM and initialized by the runtime system at startup.

In C++, objects that require runtime initialization cannot be placed in ROM.

Data types in C++

In C++, all plain C data types are represented in the same way as described earlier in this chapter. However, if any Embedded C++ features are used for a type, no assumptions can be made concerning the data representation. This means, for example, that it is not supported to write assembler code that accesses class members.

Compiler extensions

This chapter gives a brief overview of the 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 and a wide set of extensions, ranging from features specifically tailored for efficient programming in the embedded industry to the relaxation of some minor standards issues.

You can find the extensions available as:

- C/C++ language extensions

  For a summary of available language extensions, see *C language extensions*, page 196. 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. For several of them there is also a corresponding C/C++ language extension. For a list of available pragma directives, see the chapter *Pragma directives*.

- Preprocessor extensions

  The preprocessor of the compiler adheres to the ISO/ANSI standard. The compiler also makes several preprocessor-related extensions available to you. For more information, see the chapter *The preprocessor*.

- Intrinsic functions

  The intrinsic functions provide direct access to low-level processor operations and can be very useful in, for example, time-critical routines. The intrinsic functions compile into inline code, either as a single instruction or as a short sequence of instructions. To read more about using intrinsic functions, see *Mixing C and assembler*, page 87. 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. The library also provides some extensions, partly taken from the C99 standard. For more information, see *IAR DLIB Library*, page 249.

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

ENABLING LANGUAGE EXTENSIONS

In the IDE, language extensions are enabled by default.

For information about how to enable and disable language extensions from the command line, see the compiler options *-e*, page 165, and *--strict_ansi*, page 179.

C language extensions

This section gives a brief overview of the C language extensions available in the compiler. The compiler provides a wide set of extensions, so to help you to find the extensions required by your application, the extensions are 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 127, and *location*, page 224.

- Alignment

  Each data type has its own alignment, for more details, see *Alignment*, page 183. If you want to change the alignment, the `#pragma pack` and `#pragma data_alignment` directive are available. If you want to use the alignment of an object, use the `__ALIGNOF__()` operator.

  The `__ALIGNOF__` operator is used for accessing the alignment of an object. It takes one of two forms:

  - `__ALIGNOF__ (`*`type`*`)`
  - `__ALIGNOF__ (`*`expression`*`)`

  In the second form, the expression is not evaluated.

- Anonymous structs and unions

  C++ includes a feature named anonymous unions. The compiler allows a similar feature for both structs and unions in the C programming language. For more information, see *Anonymous structs and unions*, page 126.

- Bitfields and non-standard types

  In ISO/ANSI C, a bitfield must be of type `int` or `unsigned int`. Using IAR Systems language extensions, any integer type or enumeration can be used. The advantage is that the struct will sometimes be smaller. This matches G.5.8 in the appendix of the ISO standard, *ISO Portability Issues*. For more information, see *Floating-point types*, page 186.

- Dedicated segment operators `__segment_begin` and `__segment_end`

  The syntax for these operators is:

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

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

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

  In this example, the type of the `__segment_begin` operator is `void __data20 *`.

  ```
  #pragma segment="MYSEGMENT" __data20
  ...
  segment_start_address = __segment_begin("MYSEGMENT");
  ```

See also *segment*, page 229, and *location*, page 224.

USEFUL LANGUAGE EXTENSIONS

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

- Inline functions

  The `#pragma inline` directive, alternatively the `inline` keyword, advises the compiler that the function whose declaration follows immediately after the directive should be inlined. This is similar to the C++ keyword `inline`. For more information, see *inline*, page 223.
- 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)
  {
    /* Do something here. */
  }
  ```

  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 a data definition assembler directive, but not a comment. You can write several consecutive inline assembler instructions, for example:

```
asm ("Label:      nop\n"
     "            jsr 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 87.

Compound literals

To create compound literals you can use this syntax:

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

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

Note:

- A compound literal can be modified unless it is declared `const`
- 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` can be an incomplete array. This is useful for allocating a chunk of memory that contains both the structure and a fixed number of elements of the array. The number of elements can vary between allocations.

This feature is part of the C99 standard.

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

Example

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

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

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

The incomplete array will be aligned in the structure just like any other member of the structure. For more information about structure alignment, see *Structure types*, page 189.

Hexadecimal floating-point constants

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

Examples

`0x1p0` is `1`

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

Designated initializers in structures and arrays

Any initialization of either a structure (`struct` or `union`) or an array can have a designation. A designation consists of one or more designators followed by an initializer. A designator for a structure is specified as `.`*`elementname`* and for an array `[`*`constant index expression`*`]`. Using designated initializers is not supported in C++.

Examples

This definition shows a `struct` and its initialization using designators:

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

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

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

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

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

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

MINOR LANGUAGE EXTENSIONS

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

- Arrays of incomplete types

  An array can have an incomplete `struct`, `union`, or `enum` type as its element type. The types must be completed before the array is used (if it is), or by the end of the compilation unit (if it is not).

- Forward declaration of `enum` types

  The 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 can be cast to an integral type if the integral type is large enough to contain it. For more information about casting pointers, see *Casting*, page 188.
- 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 is issued. However, if constants are placed in far or data20 memory and variables are placed in data16, this will cause an error.
- Non-top level `const`

  Assignment of pointers is allowed in cases where the destination type has added type qualifiers that are not at the top level (for example, `int **` to `int const **`). Comparing and taking the difference of such pointers is also allowed.

- Non-`lvalue` arrays

  A non-`lvalue` array expression is converted to a pointer to the first element of the array when it is used.
- Comments at the end of preprocessor directives

  This extension, which makes it legal to place text after preprocessor directives, is enabled, unless strict ISO/ANSI mode is used. The purpose of this language extension is to support compilation of legacy code; we do *not* recommend that you write new code in this fashion.
- An extra comma at the end of `enum` lists

  Placing an extra comma is allowed at the end of an `enum` list. In strict ISO/ANSI mode, a warning is issued.
- A label preceding a `}`

  In ISO/ANSI C, a label must be followed by at least one statement. Therefore, it is illegal to place the label at the end of a block. The compiler issues a warning.

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. The compiler accepts this expression:

  ```
  struct str
  {
    int a;
  } x = 10;
  ```

- Declarations in other scopes

  External and static declarations in other scopes are visible. In the following example, the variable `y` can be used at the end of the function, even though it should only be visible in the body of the `if` statement. A warning is issued.

  ```
  int test(int x)
  {
    if (x)
    {
      extern int y;
      y = 1;
    }

    return y;
  }
  ```

- Expanding function names into strings with the function as context

  Use any of the symbols `__func__` or `__FUNCTION__` inside a function body to make the symbol expand into a string, with the function name as context. Use the symbol `__PRETTY_FUNCTION__` to also include the parameter types and return type. The result might, for example, look like this if you use the `__PRETTY_FUNCTION__` symbol:

  ```
  "void func(char)"
  ```

  These symbols are useful for assertions and other trace utilities and they require that language extensions are enabled, see *-e*, page 165.

Extended keywords

This chapter describes the extended keywords that support specific features of the M16C/R8C Series CPU core and the general syntax rules for the keywords. Finally the chapter gives a detailed description of each keyword.

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

General syntax rules for extended keywords

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

The compiler provides a set of attributes that can be used on functions or data objects to support specific features of the M16C/R8C Series CPU core. 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 210.

Note: The extended keywords are only available when language extensions are enabled in the compiler.

In the IDE, language extensions are enabled by default.

Use the `-e` compiler option to enable language extensions. See *-e*, page 165 for additional information.

TYPE ATTRIBUTES

Type attributes define how a function is called, or how a data object is accessed. This means that if you use a type attribute, it must be specified both when a function or data object is defined and when it is declared.

You can either place the type attributes directly in your source code, or use the pragma directive `#pragma type_attribute`.

Type attributes can be further divided into *memory type attributes* and *general type attributes*. Memory type attributes are referred to as simply *memory attributes* in the rest of the documentation.

Memory attributes

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

- Available *function memory attributes*: `__tiny_func`
- Available *data memory attributes*: `__data13`, `__data16`, `__data20`, and `__far`

Data objects, functions, and destinations of pointers or C++ references always have a memory attribute. If no attribute is explicitly specified in the declaration or by the pragma directive `#pragma type_attribute`, an appropriate default attribute is used. You can specify one memory attribute for each level of pointer indirection.

General type attributes

These general type attributes are available:

- *Function type attributes* affect how the function should be called: `__interrupt`, `__monitor`, `__regbank_interrupt`, `__simple`, and `__task`
- *Data type attributes*: `const` and `volatile`

You can specify as many type attributes as required for each level of pointer indirection.

To read more about the type qualifiers `const` and `volatile`, see *Type qualifiers*, page 191.

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 `__data20` type attribute to the variables `i` and `j`; in other words, the variable `i` and `j` is placed in data20 memory. The variables `k` and `l` behave in the same way:

```
__data20 int i, j;
int __data20 k, l;
```

Note that the attribute affects both identifiers.

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

```
#pragma type_attribute=__data20
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 17.

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

```
typedef char __data20 Byte;
typedef Byte *BytePtr;
Byte b;
BytePtr bp;
```

and

```
__data20 char b;
char __data20 *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 __data20 * p;`	The `int` object is located in `__data20` memory.
`int * __data20 p;`	The pointer is located in `__data20` memory.
`__data20 int * p;`	The pointer is located in `__data20` memory.

Syntax for type attributes on functions

The syntax for using type attributes on functions differs slightly from the syntax of type attributes on data objects. For functions, the attribute must be placed either in front of the return type, or in parentheses, for example:

```
__interrupt void my_handler(void);
```

or

```
void (__interrupt my_handler)(void);
```

This declaration of `my_handler` is equivalent with the previous one:

```
#pragma type_attribute=__interrupt
void my_handler(void);
```

Syntax for type attributes on function pointers

To declare a function pointer, use this syntax:

```
int (__data20 * fp) (double);
```

After this declaration, the function pointer `fp` points to data20 memory.

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

```
typedef __data20 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.

These object attributes are available:

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

You can specify as many object attributes as required for a specific function or data object.

For more information about `location` and `@`, see *Controlling data and function placement in memory*, page 127. For more information about `vector`, see *vector*, page 230.

Syntax for object attributes

The object attribute must be placed in front of the type. For example, to place `myarray` in memory that is not initialized at startup:

```
__no_init int myarray[10];
```

The #pragma object_attribute directive can also be used. This declaration is equivalent to the previous one:

```
#pragma object_attribute=__no_init
int myarray[10];
```

Note: Object attributes cannot be used in combination with the typedef keyword.

Summary of extended keywords

This table summarizes the extended keywords:

Extended keyword	Description
`__bitvar`	Controls the storage of variables
`__data13`	Controls the storage of data objects
`__data16`	Controls the storage of data objects
`__data20`	Controls the storage of data objects
`__far`	Controls the storage of data objects
`__huge`	Alias for `__data20`, available for backwards compatibility
`__interrupt`	Supports interrupt functions
`__intrinsic`	Reserved for compiler internal use only
`__monitor`	Supports atomic execution of a function
`__near`	Alias for `__data16`, available for backwards compatibility
`__no_init`	Supports non-volatile memory
`__noreturn`	Informs the compiler that the function will not return
`__regbank_interrupt`	Supports interrupt functions using the secondary register bank
`__root`	Ensures that a function or variable is included in the object code even if unused
`__simple`	Specifies the simple calling convention for a function
`__task`	Relaxes the rules for preserving registers
`__tiny_func`	Specifies using special page function calls

Table 35: Extended keywords summary

Descriptions of extended keywords

These sections give detailed information about each extended keyword.

__bitvar

Syntax

Follows the generic syntax rules for object attributes, see *Object attributes*, page 208.

Description

The compiler allows you to write code that is equivalent with the relocatable bit type available in the previous version of the compiler. Absolute bits are not supported.

The only declaration allowed for this data type is s a structure with unsigned bitfields with the size one (that is a bit). The variable will be stored in the `BITVARS` segment, where each bit variable only occupies one bit.

Note: No pointers to objects can be declared `__bitvar`.

Example

```
__bitvar struct {unsigned char NAME:1;}
```

If you find this syntax inconvenient, you can create a macro to define bit variables:

```
#define __BIT(NAME) __bitvar struct {unsigned char NAME:1;}
```

Then use it like this:

```
__BIT(MY_BIT);
```

__data13

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 205.

Description

The `__data13` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in data13 memory. `__data13` pointers are not allowed. However, `__data16` pointers can point to `__data13` objects.

Storage information

- Address range: `0-0x1FFF`
- Maximum object size: 8192 bytes.
- Pointer size: 2 bytes.

Example

```
__data13 int x;
```

See also

Memory types, page 13.

__data16

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 205.

Description

The `__data16` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in data16 memory. You can also use the `__data16` attribute to create a pointer explicitly pointing to an object located in the data16 memory.

Storage information

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

Example

```
__data16 int x;
```

See also

Memory types, page 13.

__data20

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 205.

Description

The `__data20` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in data20 memory. You can also use the `__data20` attribute to create a pointer explicitly pointing to an object located in the data20 memory.

Storage information

- Address range: `0-0xFFFFF`
- Maximum object size: 1,048,576 bytes.
- Pointer size: 4 bytes.

Example

```
__data20 int x;
```

See also

Memory types, page 13.

__far

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 205.

Description

The `__far` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in far memory. You can also use the `__far` attribute to create a pointer explicitly pointing to an object located in the far memory.

Storage information

- Address range: `0-0xFFFFF`
- Maximum object size: 65535 bytes. An object cannot cross a 64-Kbyte boundary.
- Pointer size: 4 bytes. Arithmetics is only performed on the two lower bytes, except comparison which is always performed on the entire 20-bit address.

Example

```
__far int x;
```

See also

Memory types, page 13.

__huge

This keyword is equivalent to the `__data20` memory attribute and is provided for backward compatibility.

__interrupt

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 205.

Description

The `__interrupt` keyword specifies interrupt functions. To specify one or several interrupt vectors, use the `#pragma vector` directive. The range of the interrupt vectors depends on the device used. It is possible to define an interrupt function without a vector, but then the compiler will not generate an entry in the interrupt vector table.

An interrupt function must have a `void` return type and cannot have any parameters.

The header file `io`*`chip`*`.h`, where *`chip`* corresponds to the selected CPU core, contains predefined names for the existing interrupt vectors.

Example

```
#pragma vector=0x14
__interrupt void my_interrupt_handler(void);
```

See also

Interrupt functions, page 24, *vector*, page 230, *INTVEC*, page 271.

__intrinsic

Description

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

__monitor

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 205.

Description

The `__monitor` keyword causes interrupts to be disabled during execution of the function. This allows atomic operations to be performed, such as operations on semaphores that control access to resources by multiple processes. A function declared with the `__monitor` keyword is equivalent to any other function in all other respects.

Example

```
__monitor int get_lock(void);
```

See also

Monitor functions, page 25. Read also about the intrinsic functions *__disable_interrupt*, page 233, *__enable_interrupt*, page 233, *__get_interrupt_state*, page 234, and *__set_interrupt_state*, page 238.

__near

This keyword is equivalent to the `__data16` memory attribute and is provided for backward compatibility.

__no_init

Syntax

Follows the generic syntax rules for object attributes, see *Object attributes*, page 208.

Description

Use the `__no_init` keyword to place a data object in non-volatile memory. This means that the initialization of the variable, for example at system startup, is suppressed.

Example

```
__no_init int myarray[10];
```

__noreturn

Syntax

Follows the generic syntax rules for object attributes, see *Object attributes*, page 208.

Description

The `__noreturn` keyword can be used on a function to inform the compiler that the function will not return. If you use this keyword on such functions, the compiler can optimize more efficiently. Examples of functions that do not return are `abort` and `exit`.

Example

```
__noreturn void terminate(void);
```

__regbank_interrupt

Syntax	Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 205.
Description	This keyword is the same as `__interrupt` except that registers are saved by switching register banks instead of pushing them on the stack. This requires that `__regbank_interrupt` cannot be interrupted by itself or other register bank interrupts.
See also	*__interrupt*, page 212

__root

Syntax	Follows the generic syntax rules for object attributes, see *Object attributes*, page 208.
Description	A function or variable with the `__root` attribute is kept whether or not it is referenced from the rest of the application, provided its module is included. Program modules are always included and library modules are only included if needed.
Example	`__root int myarray[10];`
See also	To read more about modules, segments, and the link process, see the *IAR Linker and Library Tools Reference Guide.*

__simple

Syntax	Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 205.
Description	Functions that are declared `__simple` use the simple calling convention. This keyword can be specified using the `#pragma type_attribute` directive.
See also	*Calling convention*, page 93, *type_attribute*, page 229. .

__task

Syntax	Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 205.

Description

This keyword allows functions to relax the rules for preserving registers. Typically, the keyword is used on the `main` function.

By default, functions save the contents of used preserved registers on the stack upon entry, and restore them at exit. Functions that are declared `__task` do not save all registers, and therefore require less stack space.

Because a function declared `__task` can corrupt registers that are needed by the calling function, you should only use `__task` on functions that do not return or call such a function from assembler code.

The function `main` can be declared `__task`, unless it is explicitly called from the application. In real-time applications with more than one task, the root function of each task can be declared `__task`.

The `__task` keyword must be specified both in the function declaration and when the function is defined.

Example

```
__task void my_handler(void);
```

The #pragma type_attribute directive can also be used. The following declaration of `my_handler` is equivalent with the previous one:

```
#pragma type_attribute=__task
void my_handler(void);
```

__tiny_func

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 205.

Description

The `__tiny_func` memory attribute overrides the default storage of functions and places individual functions in the special page area. An entry in a jump table will be allocated for the function. The `__tiny_func` declared function will be called with the instruction `jsrs` via an entry in the jump table. The address range is `0xF0000-0xFFFFF`.

Note: This keyword is only available when the compiler is used in M16C mode, that is when the option `--cpu=M16C` is used.

Example

This example shows the assembler code generated when a function is declared `__tiny_func`:

```
1 #pragma language=extended
2
\       In segment TINYFUNC, align 1
3  __tiny_func void myFunc()
```

```
\           myFunc:
4 {}
\ 000000 F3    RTS
5

\       In segment CODE, align 1
6  void test()
\        test:
7 {
8      myFunc();
\ 000000        REQUIRE ?flist?myFunc
\ 000000 EF..   JSRS #((0xffffe - ?flist?myFunc) >> 0x1) & 0xff)
9 }
\ 000002 F3  RTS

\      In segment FLIST, align 1
\        ?flist?myFunc:
\ 000000  ....  DC16  LWRD(myFunc-0xF0000)
\ 000002        REQUIRE myFunc
```

See also *FLIST*, page 270 and *TINYFUNC*, page 272.

Pragma directives

This chapter describes the pragma directives of the compiler.

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

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

The pragma directives are always enabled in the compiler.

Summary of pragma directives

This table lists the pragma directives of the compiler that can be used either with the `#pragma` preprocessor directive or the `_Pragma()` preprocessor operator:

Pragma directive	Description
`basic_template_matching`	Makes a template function fully memory attribute-aware
`bitfields`	Controls the order of bitfield members
`constseg`	Places constant variables in a named segment
`data_alignment`	Gives a variable a higher (more strict) alignment
`dataseg`	Places variables in a named segment
`diag_default`	Changes the severity level of diagnostic messages
`diag_error`	Changes the severity level of diagnostic messages
`diag_remark`	Changes the severity level of diagnostic messages
`diag_suppress`	Suppresses diagnostic messages
`diag_warning`	Changes the severity level of diagnostic messages
`include_alias`	Specifies an alias for an include file
`inline`	Inlines a function
`language`	Controls the IAR Systems language extensions
`location`	Specifies the absolute address of a variable, or places groups of functions or variables in named segments

Table 36: Pragma directives summary

Pragma directive	Description
`message`	Prints a message
`object_attribute`	Changes the definition of a variable or a function
`optimize`	Specifies the type and level of an optimization
`pack`	Specifies the alignment of structures and union members
`__printf_args`	Verifies that a function with a printf-style format string is called with the correct arguments
`required`	Ensures that a symbol that is needed by another symbol is included in the linked output
`rtmodel`	Adds a runtime model attribute to the module
`__scanf_args`	Verifies that a function with a scanf-style format string is called with the correct arguments
`segment`	Declares a segment name to be used by intrinsic functions
`type_attribute`	Changes the declaration and definitions of a variable or function
`vector`	Specifies the vector of an interrupt or special page function

Table 36: Pragma directives summary (Continued)

Note: For portability reasons, see also *Recognized pragma directives (6.8.6)*, page 279 and the *M16C/R8C IAR Embedded Workbench® Migration Guide*.

Descriptions of pragma directives

This section gives detailed information about each pragma directive.

basic_template_matching

Syntax

```
#pragma basic_template_matching
```

Description

Use this pragma directive in front of a template function declaration to make the function fully memory-attribute aware, in the rare cases where this is useful. That template function will then match the template without the modifications described in *Templates and data memory attributes*, page 116.

Example

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

fun((int __data16 *) 0); /* Template parameter T becomes
                             int __data16 */
```

bitfields

Syntax

```
#pragma bitfields={reversed|default}
```

Parameters

`reversed`	Bitfield members are placed from the most significant bit to the least significant bit.
`default`	Bitfield members are placed from the least significant bit to the most significant bit.

Description

Use this pragma directive to control the order of bitfield members.

By default, the compiler places bitfield members from the least significant bit to the most significant bit in the container type. Use the `#pragma bitfields=reversed` directive to place the bitfield members from the most significant to the least significant bit. This setting remains active until you turn it off again with the `#pragma bitfields=default` directive.

See also

Floating-point types, page 186.

constseg

Syntax

```
#pragma constseg=[memoryattribute ]{SEGMENT_NAME|default}
```

Parameters

`memoryattribute`	An optional memory attribute denoting in what memory the segment will be placed; if not specified, default memory is used.
`SEGMENT_NAME`	A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.
`default`	Uses the default segment for constants.

Description

Use this pragma directive to place constant variables in a named segment. The segment name cannot be a segment name predefined for use by the compiler and linker. The setting remains active until you turn it off again with the `#pragma constseg=default` directive.

Example

```
#pragma constseg=__data20 MY_CONSTANTS
const int factorySettings[] = {42, 15, -128, 0};
#pragma constseg=default
```

data_alignment

Syntax

```
#pragma data_alignment=expression
```

Parameters

expression	A constant which must be a power of two (1, 2, 4, etc.).

Description

Use this pragma directive to give a variable a higher (more strict) alignment of the start address than it would otherwise have. This directive can be used on variables with static and automatic storage duration.

When you use this directive on variables with automatic storage duration, there is an upper limit on the allowed alignment for each function, determined by the calling convention used.

Note: Normally, the size of a variable is a multiple of its alignment. The `data_alignment` directive only affects the alignment of the variable's start address, and not its size, and can thus be used for creating situations where the size is not a multiple of the alignment.

dataseg

Syntax

```
#pragma dataseg=[memoryattribute ]{SEGMENT_NAME|default}
```

Parameters

memoryattribute	An optional memory attribute denoting in what memory the segment will be placed; if not specified, default memory is used.
SEGMENT_NAME	A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.
`default`	Uses the default segment.

Description

Use this pragma directive to place variables in a named segment. The segment name cannot be a segment name predefined for use by the compiler and linker. The variable will not be initialized at startup, and can for this reason not have an initializer, which means it must be declared `__no_init`. The setting remains active until you turn it off again with the `#pragma constseg=default` directive.

Example

```
#pragma dataseg=__data20 MY_SEGMENT
__no_init char myBuffer[1000];
#pragma dataseg=default
```

diag_default

Syntax `#pragma diag_default=tag[,tag,...]`

Parameters

tag	The number of a diagnostic message, for example the message number `Pe117`.

Description Use this pragma directive to change the severity level back to the default, or to the severity level defined on the command line by any of the options `--diag_error`, `--diag_remark`, `--diag_suppress`, or `--diag_warnings`, for the diagnostic messages specified with the tags.

See also *Diagnostics*, page 148.

diag_error

Syntax `#pragma diag_error=tag[,tag,...]`

Parameters

tag	The number of a diagnostic message, for example the message number `Pe117`.

Description Use this pragma directive to change the severity level to `error` for the specified diagnostics.

See also *Diagnostics*, page 148.

diag_remark

Syntax `#pragma diag_remark=tag[,tag,...]`

Parameters

tag	The number of a diagnostic message, for example the message number `Pe177`.

Description Use this pragma directive to change the severity level to `remark` for the specified diagnostic messages.

See also *Diagnostics*, page 148.

diag_suppress

Syntax

```
#pragma diag_suppress=tag[,tag,...]
```

Parameters

tag	The number of a diagnostic message, for example the message number `Pe117`.

Description

Use this pragma directive to suppress the specified diagnostic messages.

See also

Diagnostics, page 148.

diag_warning

Syntax

```
#pragma diag_warning=tag[,tag,...]
```

Parameters

tag	The number of a diagnostic message, for example the message number `Pe826`.

Description

Use this pragma directive to change the severity level to `warning` for the specified diagnostic messages.

See also

Diagnostics, page 148.

include_alias

Syntax

```
#pragma include_alias ("orig_header" , "subst_header")
#pragma include_alias (<orig_header> , <subst_header>)
```

Parameters

orig_header	The name of a header file for which you want to create an alias.
subst_header	The alias for the original header file.

Description

Use this pragma directive to provide an alias for a header file. This is useful for substituting one header file with another, and for specifying an absolute path to a relative file.

This pragma directive must appear before the corresponding `#include` directives and `subst_header` must match its corresponding `#include` directive exactly.

Example

```
#pragma include_alias (<stdio.h> , <C:\MyHeaders\stdio.h>)
#include <stdio.h>
```

This example will substitute the relative file `stdio.h` with a counterpart located according to the specified path.

See also

Include file search procedure, page 146.

inline

Syntax

```
#pragma inline[=forced]
```

Parameters

`forced`	Disables the compiler's heuristics and forces inlining.

Description

Use this pragma directive to advise the compiler that the function whose declaration follows immediately after the directive should be inlined—that is, expanded into the body of the calling function. Whether the inlining actually occurs is subject to the compiler's heuristics.

This is similar to the C++ keyword `inline`, but has the advantage of being available in C code.

Specifying `#pragma inline=forced` disables the compiler's heuristics and forces inlining. If the inlining fails for some reason, for example if it cannot be used with the function type in question (like `printf`), an error message is emitted.

Note: Because specifying `#pragma inline=forced` disables the compiler's heuristics, including the inlining heuristics, the function declared immediately after the directive will not be inlined on optimization levels None or Low. No error or warning message will be emitted.

language

Syntax

```
#pragma language={extended|default}
```

Parameters

`extended`	Turns on the IAR Systems language extensions and turns off the `--strict_ansi` command line option.
`default`	Uses the language settings specified by compiler options.

Description

Use this pragma directive to enable the compiler language extensions or for using the language settings specified on the command line.

location

Syntax

```
#pragma location={address|NAME}
```

Parameters

address	The absolute address of the global or static variable for which you want an absolute location.
NAME	A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.

Description

Use this pragma directive to specify the location—the absolute address—of the global or static variable whose declaration follows the pragma directive. The variable must be declared either `__no_init` or `const`. Alternatively, the directive can take a string specifying a segment for placing either a variable or a function whose declaration follows the pragma directive.

Example

```
#pragma location=0x3E1
__no_init volatile char PORT1; /* PORT1 is located at address
                                  0x3E1 */

#pragma location="foo"
char PORT1; /* PORT1 is located in segment foo */

/* A better way is to use a corresponding mechanism */
#define FLASH _Pragma("location=\"FLASH\"")
...
FLASH int i; /* i is placed in the FLASH segment */
```

See also

Controlling data and function placement in memory, page 127.

message

Syntax

```
#pragma message(message)
```

Parameters

message	The message that you want to direct to the standard output stream.

Description

Use this pragma directive to make the compiler print a message to the standard output stream when the file is compiled.

Example:

```
#ifdef TESTING
#pragma message("Testing")
#endif
```

object_attribute

Syntax

```
#pragma object_attribute=object_attribute[,object_attribute,...]
```

Parameters

For a list of object attributes that can be used with this pragma directive, see *Object attributes*, page 208.

Description

Use this pragma directive to declare a variable or a function with an object attribute. This directive affects the definition of the identifier that follows immediately after the directive. The object is modified, not its type. Unlike the directive `#pragma type_attribute` that specifies the storing and accessing of a variable or function, it is not necessary to specify an object attribute in declarations.

Example

```
#pragma object_attribute=__no_init
char bar;
```

See also

General syntax rules for extended keywords, page 205.

optimize

Syntax

```
#pragma optimize=param[ param...]
```

Parameters

`balanced|size|speed`	Optimizes balanced between speed and size, optimizes for size, or optimizes for speed
`none|low|medium|high`	Specifies the level of optimization
`no_code_motion`	Turns off code motion
`no_cse`	Turns off common subexpression elimination
`no_inline`	Turns off function inlining
`no_tbaa`	Turns off type-based alias analysis
`no_unroll`	Turns off loop unrolling
`no_scheduling`	Turns off instruction scheduling

Description

Use this pragma directive to decrease the optimization level, or to turn off some specific optimizations. This pragma directive only affects the function that follows immediately after the directive.

The parameters `speed`, `size`, and `balanced` only have effect on the `high` optimization level and only one of them can be used as it is not possible to optimize for speed and size at the same time. It is also not possible to use preprocessor macros embedded in this pragma directive. Any such macro will not be expanded by the preprocessor.

Note: If you use the `#pragma optimize` directive to specify an optimization level that is higher than the optimization level you specify using a compiler option, the pragma directive is ignored.

Example

```
#pragma optimize=speed
int small_and_used_often()
{
    ...
}

#pragma optimize=size no_inline
int big_and_seldom_used()
{
    ...
}
```

pack

Syntax

```
#pragma pack(n)
#pragma pack()
#pragma pack({push|pop}[,name] [,n])
```

Parameters

n	Sets an optional structure alignment; one of: `1`, `2`, `4`, `8`, or `16`
Empty list	Restores the structure alignment to default
`push`	Sets a temporary structure alignment
`pop`	Restores the structure alignment from a temporarily pushed alignment
name	An optional pushed or popped alignment label

Description

Use this pragma directive to specify the maximum alignment of `struct` and `union` members.

The `#pragma pack` directive affects declarations of structures following the pragma directive to the next `#pragma pack` or end of file.

Note: This can result in significantly larger and slower code when accessing members of the structure.

See also

Structure types, page 189.

__printf_args

Syntax

```
#pragma __printf_args
```

Description

Use this pragma directive on a function with a printf-style format string. For any call to that function, the compiler verifies that the argument to each conversion specifier (for example `%d`) is syntactically correct.

Example

```
#pragma __printf_args
int printf(char const *,...);


/* Function call */
printf("%d",x);  /* Compiler checks that x is a double */
```

required

Syntax

`#pragma required=`*`symbol`*

Parameters

symbol	Any statically linked function or variable.

Description

Use this pragma directive to ensure that a symbol which is needed by a second symbol is included in the linked output. The directive must be placed immediately before the second symbol.

Use the directive if the requirement for a symbol is not otherwise visible in the application, for example if a variable is only referenced indirectly through the segment it resides in.

Example

```
const char copyright[] = "Copyright by me";

#pragma required=copyright
int main()
{
  /* Do something here. */
}
```

Even if the copyright string is not used by the application, it will still be included by the linker and available in the output.

rtmodel

Syntax	`#pragma rtmodel="key","value"`

Parameters

`"key"`	A text string that specifies the runtime model attribute.
`"value"`	A text string that specifies the value of the runtime model attribute. Using the special value * is equivalent to not defining the attribute at all.

Description

Use this pragma directive to add a runtime model attribute to a module, which can be used by the linker to check consistency between modules.

This pragma directive is useful for enforcing consistency between modules. All modules that are linked together and define the same runtime attribute key must have the same value for the corresponding key, or the special value `*`. It can, however, be useful to state explicitly that the module can handle any runtime model.

A module can have several runtime model definitions.

Note: The predefined compiler runtime model attributes start with a double underscore. To avoid confusion, this style must not be used in the user-defined attributes.

Example

```
#pragma rtmodel="I2C","ENABLED"
```

The linker will generate an error if a module that contains this definition is linked with a module that does not have the corresponding runtime model attributes defined.

See also

Checking module consistency, page 73.

__scanf_args

Syntax

```
#pragma __scanf_args
```

Description

Use this pragma directive on a function with a scanf-style format string. For any call to that function, the compiler verifies that the argument to each conversion specifier (for example `%d`) is syntactically correct.

Example

```
#pragma __scanf_args
int printf(char const *,...);


/* Function call */
scanf("%d",x);  /* Compiler checks that x is a double */
```

segment

Syntax

```
#pragma segment="NAME" [memoryattribute] [align]
```

Parameters

NAME	The name of the segment
memoryattribute	An optional memory attribute identifying the memory the segment will be placed in; if not specified, default memory is used.
align	Specifies an alignment for the segment part. The value must be a constant integer expression to the power of two.

Description

Use this pragma directive to define a segment name that can be used by the segment operators `__segment_begin` and `__segment_end`. All segment declarations for a specific segment must have the same memory type attribute and alignment.

If an optional memory attribute is used, the return type of the segment operators `__segment_begin` and `__segment_end` is:

`void` *`memoryattribute`* `*`.

Example

```
#pragma segment="MYDATA20" __data20 4
```

See also

Important language extensions, page 196. For more information about segments and segment parts, see the chapter *Placing code and data*.

type_attribute

Syntax

```
#pragma type_attribute=type_attribute[,type_attribute,...]
```

Parameters

For a list of type attributes that can be used with this pragma directive, see *Type attributes*, page 205.

Description

Use this pragma directive to specify IAR-specific *type attribute*s, which are not part of the ISO/ANSI C language standard. Note however, that a given type attribute might not be applicable to all kind of objects.

This directive affects the declaration of the identifier, the next variable, or the next function that follows immediately after the pragma directive.

Example

In this example, an `int` object with the memory attribute `__data16` is defined:

```
#pragma type_attribute=__data16
int x;
```

This declaration, which uses extended keywords, is equivalent:

```
__data16 int x;
```

See also

See the chapter *Extended keywords* for more details.

vector

Syntax

```
#pragma vector=vector1[, vector2, vector3, ...]
```

Parameters

vector	The vector number(s) of an interrupt or special page function.

Description

Use this pragma directive to specify the vector(s) of an interrupt or special page function whose declaration follows the pragma directive. Note that several vectors can be defined for each function.

Example!

```
#pragma vector=0x14
__interrupt void my_handler(void);
```

Intrinsic functions

This chapter gives reference information about the intrinsic functions, a predefined set of functions available in the compiler.

The intrinsic functions provide direct access to low-level processor operations and can be very useful in, for example, time-critical routines. The intrinsic functions compile into inline code, either as a single instruction or as a short sequence of instructions.

Summary of intrinsic functions

To use intrinsic functions in an application, include the header file `intrinsics.h`.

Note that the intrinsic function names start with double underscores, for example:

`__disable_interrupt`

This table summarizes the intrinsic functions:

Intrinsic function	Description
`__break`	Inserts a `brk` instruction
`__BTSTS`	Inserts a `btsts` instruction
`__disable_interrupt`	Disables interrupts
`__enable_interrupt`	Enables interrupts
`__get_FLG_register`	Reads the value of the `FLG` register
`__get_interrupt_level`	Returns the interrupt level
`__get_interrupt_state`	Returns the interrupt state
`__illegal_opcode`	Inserts an illegal operation code
`__no_operation`	Inserts a `nop` instruction
`__overflow`	Reads the value of the overflow flag
`__require`	Ensures that the module containing the specified symbol is linked
`__RMPA_B`	Inserts an `rmpa.b` instruction
`__RMPA_B_INTO`	Inserts an `rmpa.b` instruction, followed by an `into` instruction

Table 37: Intrinsic functions summary

Intrinsic function	Description
`__RMPA_B_overflow`	Inserts an `rmpa.b` instruction and stores the value of the overflow flag
`__RMPA_W`	Inserts an `rmpa.w` instruction
`__RMPA_W_INTO`	Inserts an `rmpa.w` instruction, followed by an `into` instruction
`__RMPA_W_overflow`	Inserts an `rmpa.w` instruction and stores the value of the overflow flag
`__segment_begin`	Returns the start address of a segment
`__segment_end`	Returns the end address of a segment
`__set_FLG_register`	Sets the value of the `FLG` register
`__set_INTB_register`	Sets the value of the `INTB` register
`__set_interrupt_level`	Sets the interrupt level
`__set_interrupt_state`	Restores the interrupt state
`__SMOVB_B`	Inserts a `smovb.b` instruction
`__SMOVB_W`	Inserts a `smovb.w` instruction
`__SMOVF_B`	Inserts a `smovf.b` instruction
`__SMOVF_W`	Inserts a `smovf.w` instruction
`__software_interrupt`	Inserts an `int` instruction
`__SSTR_B`	Inserts a `sstr.b` instruction
`__SSTR_W`	Inserts a `sstr.w` instruction
`__wait_for_interrupt`	Inserts a `wait` instruction

Table 37: Intrinsic functions summary (Continued)

Descriptions of intrinsic functions

This section gives reference information about each intrinsic function.

__break

Syntax

```
void __break(void);
```

Description

Inserts a `brk` instruction.

__BTSTS

Syntax `void __BTSTS(unsigned char b, unsigned char __data16 * a);`

Description Inserts a `btsts` instruction, addressing the bit *b* at address *a*. Note that the address must be in data13 memory, because the instruction uses bit addressing.

__disable_interrupt

Syntax `void __disable_interrupt(void);`

Description Clears the Interrupt Enable flag (flag `I` of the `FLG` register).

__enable_interrupt

Syntax `void __enable_interrupt(void);`

Description Sets the Interrupt Enable flag (flag `I` of the `FLG` register).

__get_FLG_register

Syntax `unsigned short __get_FLG_register();`

Description Returns the value of the `FLG` register.

__get_interrupt_level

Syntax `__ilevel_t __get_interrupt_level(void);`

Description Returns the current interrupt level. The return type `__ilevel_t` has this definition:

```
typedef unsigned char __ilevel_t;
```

The return value of `__get_interrupt_level` can be used as an argument to the `__set_interrupt_level` intrinsic function.

__get_interrupt_state

Syntax

```
__istate_t __get_interrupt_state(void);
```

Description

Returns the global interrupt state. The return value can be used as an argument to the `__set_interrupt_state` intrinsic function, which will restore the interrupt state.

Example

```
__istate_t s = __get_interrupt_state();
__disable_interrupt();

/* Do something here. */

__set_interrupt_state(s);
```

The advantage of using this sequence of code compared to using `__disable_interrupt` and `__enable_interrupt` is that the code in this example will not enable any interrupts disabled before the call of `__get_interrupt_state`.

__illegal_opcode

Syntax

```
void __illegal_opcode(void);
```

Description

Inserts an illegal operation code.

__no_operation

Syntax

```
void __no_operation(void);
```

Description

Inserts a `nop` instruction.

__overflow

Syntax

```
extern unsigned char __data13 __overflow();
```

Description

Reads the value of the overflow flag, as previously stored by the `__RMPA_B_overflow` or `__RMPA_W_overflow` intrinsic functions. At higher optimization levels, the overflow flag will be used directly if it is preserved between the call to `__RMPA_B_overflow` or `__RMPA_B_overflow` and the accessing of `__overflow`.

__require

Syntax

```
void __require(void __data20 *);
```

Description

Sets a constant literal as required.

XLINK excludes anything that is not needed. This is good because it reduces the resulting code size to a minimum. However, in some situations you might want to explicitly include a piece of code or a variable even though it is not directly used.

The argument to `__require` could be a variable, a function name, or an exported assembler label. However, it must be a constant literal. The label referred to will be treated as if it would be used at the location of the `__require` call.

__RMPA_B

Syntax

```
short __RMPA_B(const signed char __data16 * v1, const signed
char __data16 * v2, short init, unsigned short n);
```

Description

Inserts an `rmpa.b` instruction.

This instruction performs a scalar product between the vectors *`v1`* and *`v2`*, where *`n`* specifies the size of the vectors and *`init`* is added to the result. The result returned is undefined in the event of an overflow.

__RMPA_B_INTO

Syntax

```
short __RMPA_B_INTO(const signed char __data16 * v1, const
signed char __data16 * v2, short init, unsigned short n);
```

Description

Inserts an `rmpa.b` instruction, followed by an `into` instruction.

This instruction performs a scalar product between the vectors *`v1`* and *`v2`*, where *`n`* specifies the size of the vectors and *`init`* is added to the result. The result returned is undefined in the event of an overflow.

__RMPA_B_overflow

Syntax

```
short __RMPA_B_overflow(const signed char __data16 * v1, const
signed char __data16 * v2, short init, unsigned short n);
```

Description

Inserts an `rmpa.b` instruction and stores the value of the overflow flag.

This instruction performs a scalar product between the vectors *v1* and *v2*, where *n* specifies the size of the vectors and *init* is added to the result. The result returned is undefined in the event of an overflow.

See also

__overflow, page 234.

__RMPA_W

Syntax

```
short __RMPA_W(const signed char __data16 * v1, const signed
char __data16 * v2, short init, unsigned short n);
```

Description

Inserts an `rmpa.w` instruction.

This instruction performs a scalar product between the vectors *v1* and *v2*, where *n* specifies the size of the vectors and *init* is added to the result. The result returned is undefined in the event of an overflow.

__RMPA_W_INTO

Syntax

```
short __RMPA_W_INTO(const signed char __data16 * v1, const
signed char __data16 * v2, short init, unsigned short n);
```

Description

Inserts an `rmpa.w` instruction, followed by an `into` instruction.

This instruction performs a scalar product between the vectors *v1* and *v2*, where *n* specifies the size of the vectors and *init* is added to the result. The result returned is undefined in the event of an overflow.

__RMPA_W_overflow

Syntax

```
short __RMPA_W_overflow(const signed char __data16 * v1, const
signed char __data16 * v2, short init, unsigned short n);
```

Description

Inserts an `rmpa.w` instruction and stores the value of the overflow flag.

This instruction performs a scalar product between the vectors *v1* and *v2*, where *n* specifies the size of the vectors and *init* is added to the result. The result returned is undefined in the event of an overflow.

See also

__overflow, page 234.

__segment_begin

Syntax `__segment_begin(`*`segment`*`);`

Description Returns the address of the first byte of the named *`segment`*. The named *`segment`* must be a string literal that has been declared earlier with the `#pragma segment` directive. See *segment*, page 229.

If the segment was declared with a memory attribute *`memattr`*, the type of the `__segment_begin` function is pointer to *`memattr`* `void`. Otherwise, the type is a default pointer to `void`.

Note: You must have enabled language extensions to use this intrinsic function.

__segment_end

Syntax `__segment_end(`*`segment`*`);`

Description Returns the address of the first byte *after* the named *`segment`*. The named *`segment`* must be a string literal that has been declared earlier with the `#pragma segment` directive. See *segment*, page 229.

If the segment was declared with a memory attribute *`memattr`*, the type of the `__segment_end` function is pointer to *`memattr`* `void`. Otherwise, the type is a default pointer to `void`.

Note: You must have enabled language extensions to use this intrinsic function.

__set_FLG_register

Syntax `void __set_FLG_register(unsigned short);`

Description Sets the value of the `FLG` register.

__set_INTB_register

Syntax `void __set_INTB_register(unsigned long);`

Description Sets the value of the `INTB` register.

__set_interrupt_level

Syntax
`void __get_interrupt_level(__ilevel_t);`

Description
Sets the interrupt level. For information about the `__ilevel_t` type, see *__get_interrupt_level*, page 233.

__set_interrupt_state

Syntax
`void __set_interrupt_state(__istate_t);`

Descriptions
Restores the interrupt state to a value previously returned by the `__get_interrupt_state` function.

For information about the `__istate_t` type, see *__get_interrupt_state*, page 234.

__SMOVB_B

Syntax
```
void __SMOVB_B(char __data16 * dest, const char __far * src,
unsigned short n);
```

Description
Inserts an `smovb.b` instruction. This instruction copies *n* bytes from the address *src* to the address *dest* by decreasing the addresses after each byte copied.

__SMOVB_W

Syntax
```
void __SMOVB_W(short __data16 * dest, const short __far * src,
unsigned short n);
```

Description
Inserts an `smovb.w` instruction. This instruction copies *n* words from the address *src* to the address *dest* by decreasing the addresses after each word copied.

__SMOVF_B

Syntax
```
void __SMOVF_B(char __data16 * dest, const char __far * src,
unsigned short n);
```

Description
Inserts an `smovf.b` instruction. This instruction copies *n* bytes from the address *src* to the address *dest* by increasing the addresses after each byte copied.

__SMOVF_W

Syntax

```
void __SMOVF_W(short __data16 * dest, const short __far * src,
unsigned short n);
```

Description

Inserts an `smovf.w` instruction. This instruction copies *n* words from the address *src* to the address *dest* by increasing the addresses after each word copied.

__software_interrupt

Syntax

```
void __software_interrupt(void);
```

Description

Inserts an `int` instruction.

__SSTR_B

Syntax

```
void __SSTR_B(char __data16 * dest, signed char init, unsigned
short n);
```

Description

Inserts an `sstr.b` instruction. This instruction stores the data *init* to *n* bytes starting at the address *dest*.

__SSTR_W

Syntax

```
void __SSTR_W(short __data16 * dest, short init, unsigned short
n);
```

Description

Inserts an `sstr.w` instruction. This instruction stores the data *init* to *n* words starting at the address *dest*.

__wait_for_interrupt

Syntax

```
void __wait_for_interrupt(void);
```

Description

Inserts a `wait` instruction.

The preprocessor

This chapter gives a brief overview of the preprocessor, including reference information about the different preprocessor directives, symbols, and other related information.

Overview of the preprocessor

The preprocessor of the IAR C/C++ Compiler for M16C/R8C adheres to the ISO/ANSI standard. The compiler also makes these preprocessor-related features available to you:

- Predefined preprocessor symbols

  These symbols allow you to inspect the compile-time environment, for example the time and date of compilation. For details, see *Descriptions of predefined preprocessor symbols*, page 242.
- User-defined preprocessor symbols defined using a compiler option

  In addition to defining your own preprocessor symbols using the `#define` directive, you can also use the option `-D`, see *-D*, page 160.
- Preprocessor extensions

  There are several preprocessor extensions, for example many pragma directives; for more information, see the chapter *Pragma directives* in this guide. Read also about the corresponding `_Pragma` operator and the other extensions related to the preprocessor, see *Descriptions of miscellaneous preprocessor extensions*, page 245.
- Preprocessor output

  Use the option `--preprocess` to direct preprocessor output to a named file, see *--preprocess*, page 177.

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 279.

Descriptions of predefined preprocessor symbols

This table describes the predefined preprocessor symbols:

Predefined symbol	Identifies
`__BASE_FILE__`	A string that identifies the name of the base source file (that is, not the header file), being compiled. See also `__FILE__`, page 243, and *–no_path_in_file_macros*, page 172.
`__BUILD_NUMBER__`	An 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.
`__CONSTANT_DATA__`	An integer that identifies the default placement of constant data. The symbol reflects the `--constant_data` option and is defined to `__CONSTANT_DATA_NEAR__`, `__CONSTANT_DATA_FAR__`, or `__CONSTANT_DATA_HUGE__`. These symbolic names can be used when testing the `__CONSTANT_DATA__` symbol.
`__CORE__`	An integer that identifies the chip core in use. The symbol reflects the `--core` option and is defined to `0` for M16C mode or `1` for R8C mode.
`__cplusplus`	An integer which is defined when the compiler runs in any of the C++ modes, otherwise it is undefined. When defined, its value is `199711L`. This symbol can be used with `#ifdef` to detect whether the compiler accepts C++ code. It is particularly useful when creating header files that are to be shared by C and C++ code.*
`__DATA_MODEL__`	An integer that identifies the data model in use. The symbol reflects the `--data_model` option and is defined to `__DATA_MODEL_NEAR__`, `__DATA_MODEL_FAR__`, or `__DATA_MODEL_HUGE__`. These symbolic names can be used when testing the `__DATA_MODEL__` symbol.
`__DATE__`	A string that identifies the date of compilation, which is returned in the form "`Mmm dd yyyy`", for example "`Oct 30 2008`". *
`__embedded_cplusplus`	An integer which is defined to `1` when the compiler runs in any of the C++ modes, otherwise the symbol is undefined. This symbol can be used with `#ifdef` to detect whether the compiler accepts C++ code. It is particularly useful when creating header files that are to be shared by C and C++ code.*

Table 38: Predefined symbols

Predefined symbol	Identifies
`__FILE__`	A string that identifies the name of the file being compiled, which can be both the base source file and any included header file. See also *`__BASE_FILE__`*, page 242, and *--no_path_in_file_macros*, page 172.*
`__func__`	A string that identifies the name of the function in which the symbol is used. This is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see -e, page 165. See also *`__PRETTY_FUNCTION__`*, page 243.
`__FUNCTION__`	A string that identifies the name of the function in which the symbol is used. This is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see -e, page 165. See also *`__PRETTY_FUNCTION__`*, page 243.
`__IAR_SYSTEMS_ICC__`	An integer that identifies the IAR compiler platform. The current value is 7. Note that the number could be higher in a future version of the product. This symbol can be tested with `#ifdef` to detect whether the code was compiled by a compiler from IAR Systems.
`__ICCM16C__`	An integer that is set to `1` when the code is compiled with the IAR C/C++ Compiler for M16C/R8C, and otherwise to `0`.
`__LINE__`	An integer that identifies the current source line number of the file being compiled, which can be both the base source file and any included header file.*
`__PRETTY_FUNCTION__`	A string that identifies the function name, including parameter types and return type, of the function in which the symbol is used, for example `"void func(char)"`. This symbol is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see -e, page 165. See also *`__func__`*, page 243.
`__STDC__`	An integer that is set to `1`, which means the compiler adheres to the ISO/ANSI C standard. This symbol can be tested with `#ifdef` to detect whether the compiler in use adheres to ISO/ANSI C.*
`__STDC_VERSION__`	An integer that identifies the version of ISO/ANSI C standard in use. The symbols expands to `199409L`. This symbol does not apply in EC++ mode.*

Table 38: Predefined symbols (Continued)

Predefined symbol	Identifies
`__SUBVERSION__`	An integer that identifies the subversion number of the compiler version number, for example 3 in 1.2.3.4.

`__TID__`	Target identifier for the IAR C/C++ Compiler for M16C/R8C. Expands to the target identifier which contains these parts:
	• A number unique for each IAR compiler. For the M16C/R8C Series of CPU cores, the target identifier is `28`.
	• The value of the `cpu`, which is 0 in this compiler.
	• The value corresponding to the `--data_model` option. The value is 0, 1, or 2 for the near, far, and huge data model, respectively.
	• An intrinsic flag. This flag is set for M16C/R8C because the compiler supports intrinsic functions.
	The `__TID__` value is constructed as:
	`((i << 15) | (t << 8) | (c << 4) | d)`
	You can extract the values like this:
	`i = (__TID__ >> 15) & 0x01; /* intrinsic flag */`
	`t = (__TID__ >> 8) & 0x7F;  /* target identifier */`
	`c = (__TID__ >> 4) & 0x0F); /* cpu */`
	`d = __TID__ >> & 0x0F;      /* data model */`
	In other words, the `__TID__` symbol is `0x9C00` when the near data model is used, `0x9C01` when the far data model is used, and `0x9C02` when the huge data model is used.

`__TIME__`	A string that identifies the time of compilation in the form `"hh:mm:ss"`.*
`__VARIABLE_DATA__`	Identifies where variables are placed in memory. The value of this symbol is `__VARIABLE_DATA_`*`X`*`__`, where *`X`* is `NEAR`, `FAR`, or `HUGE` depending on the data model.
`__VER__`	An integer that identifies the version number of the IAR compiler in use. The value of the number is calculated in this way: `(100 * the major version number + the minor version number)`. For example, for compiler version 3.34, 3 is the major version number and 34 is the minor version number. Hence, the value of `__VER__` is 334.

Table 38: Predefined symbols (Continued)

* **This symbol is required by the ISO/ANSI standard.**

Descriptions of miscellaneous preprocessor extensions

This section gives reference information about the preprocessor extensions that are available in addition to the predefined symbols, pragma directives, and ISO/ANSI directives.

NDEBUG

Description

This preprocessor symbol determines whether any assert macros you have written in your application shall be included or not in the built application.

If this symbol is not defined, all assert macros are evaluated. If the symbol is defined, all assert macros are excluded from the compilation. In other words, if the symbol is:

- **defined**, the assert code will *not* be included
- **not defined**, the assert code will be included

This means that if you write any assert code and build your application, you should define this symbol to exclude the assert code from the final application.

Note that the assert macro is defined in the `assert.h` standard include file.

In the IDE, the `NDEBUG` symbol is automatically defined if you build your application in the Release build configuration.

_Pragma()

Syntax

`_Pragma("string")`

where *`string`* follows the syntax of the corresponding pragma directive.

Description

This preprocessor operator is part of the C99 standard and can be used, for example, in defines and is equivalent to the `#pragma` directive.

Note: The `-e` option—enable language extensions—does not have to be specified.

Example

```
#if NO_OPTIMIZE
  #define NOOPT _Pragma("optimize=none")
#else
  #define NOOPT
#endif
```

See also

See the chapter *Pragma directives*.

#warning message

Syntax

```
#warning message
```

where *message* can be any string.

Description

Use this preprocessor directive to produce messages. Typically, this is useful for assertions and other trace utilities, similar to the way the ISO/ANSI standard `#error` directive is used.

__VA_ARGS__

Syntax

```
#define P(...)       __VA_ARGS__
#define P(x, y, ...)   x + y + __VA_ARGS__
```

`__VA_ARGS__` will contain all variadic arguments concatenated, including the separating commas.

Description

Variadic macros are the preprocessor macro equivalents of `printf` style functions. `__VA_ARGS__` is part of the C99 standard.

Example

```
#if DEBUG
  #define DEBUG_TRACE(S, ...) printf(S, __VA_ARGS__)
#else
  #define DEBUG_TRACE(S, ...)
#endif
/* Place your own code here */
DEBUG_TRACE("The value is:%d\n",value);
```

will result in:

```
printf("The value is:%d\n",value);
```

Library functions

This chapter gives an introduction to the C 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 compiler provides two different libraries:

- IAR DLIB Library is a complete ISO/ANSI C and C++ library. This library also supports floating-point numbers in IEEE 754 format and it can be configured to include different levels of support for locale, file descriptors, multibyte characters, et cetera.
- IAR CLIB Library is a light-weight library, which is not fully compliant with ISO/ANSI C. Neither does it fully support floating-point numbers in IEEE 754 format or does it support Embedded C++.

Note that different customization methods are normally needed for these two libraries. For additional information, see the chapter *The DLIB runtime environment* and *The CLIB runtime environment*, respectively.

For detailed information about the library functions, see the online documentation supplied with the product. There is also keyword reference information for the DLIB library functions. To obtain reference information for a function, select the function name in the editor window and press F1.

For additional information about library functions, see the chapter *Implementation-defined behavior* in this guide.

HEADER FILES

Your application program gains access to library definitions through header files, which it incorporates using the `#include` directive. The definitions are divided into several different header files, each covering a particular functional area, letting you include just those that are required.

It is essential to include the appropriate header file before making any reference to its definitions. Failure to do so can cause the call to fail during execution, or generate error or warning messages at compile time or link time.

LIBRARY OBJECT FILES

Most of the library definitions can be used without modification, that is, directly from the library object files that are supplied with the product. For information about how to choose a runtime library, see *Basic 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 these functions and parts are not reentrant because they need static data:

- Heap functions—`malloc`, `free`, `realloc`, `calloc`, and the C++ operators `new` and `delete`
- Time functions—`asctime`, `localtime`, `gmtime`, `mktime`
- Multibyte functions—`mbrlen`, `mbrtowc`, `mbsrtowc`, `wcrtomb`, `wcsrtomb`, `wctomb`
- The miscellaneous functions `setlocale`, `rand`, `atexit`, `strerror`, `strtok`
- Functions that use files in some way. This includes `printf`, `scanf`, `getchar`, and `putchar`. The functions `sprintf` and `sscanf` are not included.

For the CLIB library, the `qsort` function and functions that use files in some way are non-reentrant. This includes `printf`, `scanf`, `getchar`, and `putchar`. The functions `sprintf` and `sscanf` are not included.

Some functions also share the same storage for `errno`. These functions are not reentrant, since an `errno` value resulting from one of these functions can be destroyed by a subsequent use of the function before it is read. Among these functions are:

```
exp, exp10, ldexp, log, log10, pow, sqrt, acos, asin, atan2,
cosh, sinh, strtod, strtol, strtoul
```

Remedies for this are:

- Do not use non-reentrant functions in interrupt service routines
- Guard calls to a non-reentrant function by a mutex, or a secure region, etc.

IAR DLIB Library

The IAR DLIB Library provides most of the important C and C++ library definitions that apply to embedded systems. These are of the following types:

- Adherence to a free-standing implementation of the ISO/ANSI standard for the programming language C. For additional information, see the chapter *Implementation-defined behavior* in this guide.
- Standard C library definitions, for user programs.
- Embedded C++ library definitions, for user programs.
- `CSTARTUP`, the module containing the start-up code. It is described in the chapter *The DLIB runtime environment* in this guide.
- Runtime support libraries; for example low-level floating-point routines.
- Intrinsic functions, allowing low-level use of M16C/R8C Series features. See the chapter *Intrinsic functions* for more information.

In addition, the IAR DLIB Library includes some added C functionality, partly taken from the C99 standard, see *Added C functionality*, page 252.

C HEADER FILES

This section lists the header files specific to the DLIB library C definitions. Header files may additionally contain target-specific definitions; these are documented in the chapter *Compiler extensions*.

The following table lists the C header files:

Header file	Usage
`assert.h`	Enforcing assertions when functions execute
`ctype.h`	Classifying characters
`errno.h`	Testing error codes reported by library functions
`float.h`	Testing floating-point type properties
`inttypes.h`	Defining formatters for all types defined in `stdint.h`
`iso646.h`	Using Amendment 1—iso646.h standard header
`limits.h`	Testing integer type properties
`locale.h`	Adapting to different cultural conventions
`math.h`	Computing common mathematical functions
`setjmp.h`	Executing non-local goto statements
`signal.h`	Controlling various exceptional conditions
`stdarg.h`	Accessing a varying number of arguments

Table 39: Traditional standard C header files—DLIB

Header file	Usage
`stdbool.h`	Adds support for the `bool` data type in C.
`stddef.h`	Defining several useful types and macros
`stdint.h`	Providing integer characteristics
`stdio.h`	Performing input and output
`stdlib.h`	Performing a variety of operations
`string.h`	Manipulating several kinds of strings
`time.h`	Converting between various time and date formats
`wchar.h`	Support for wide characters
`wctype.h`	Classifying wide characters

Table 39: Traditional standard C header files—DLIB (Continued)

C++ HEADER FILES

This section lists the C++ header files.

Embedded C++

The following table lists the Embedded C++ header files:

Header file	Usage
`complex`	Defining a class that supports complex arithmetic
`exception`	Defining several functions that control exception handling
`fstream`	Defining several I/O stream classes that manipulate external files
`iomanip`	Declaring several I/O stream manipulators that take an argument
`ios`	Defining the class that serves as the base for many I/O streams classes
`iosfwd`	Declaring several I/O stream classes before they are necessarily defined
`iostream`	Declaring the I/O stream objects that manipulate the standard streams
`istream`	Defining the class that performs extractions
`new`	Declaring several functions that allocate and free storage
`ostream`	Defining the class that performs insertions
`sstream`	Defining several I/O stream classes that manipulate string containers
`stdexcept`	Defining several classes useful for reporting exceptions
`streambuf`	Defining classes that buffer I/O stream operations
`string`	Defining a class that implements a string container
`strstream`	Defining several I/O stream classes that manipulate in-memory character sequences

Table 40: Embedded C++ header files

The following table lists additional C++ header files:

Header file	Usage
`fstream.h`	Defining several I/O stream classes that manipulate external files
`iomanip.h`	Declaring several I/O stream manipulators that take an argument
`iostream.h`	Declaring the I/O stream objects that manipulate the standard streams
`new.h`	Declaring several functions that allocate and free storage

Table 41: 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
`algorithm`	Defines several common operations on sequences
`deque`	A deque sequence container
`functional`	Defines several function objects
`hash_map`	A map associative container, based on a hash algorithm
`hash_set`	A set associative container, based on a hash algorithm
`iterator`	Defines common iterators, and operations on iterators
`list`	A doubly-linked list sequence container
`map`	A map associative container
`memory`	Defines facilities for managing memory
`numeric`	Performs generalized numeric operations on sequences
`queue`	A queue sequence container
`set`	A set associative container
`slist`	A singly-linked list sequence container
`stack`	A stack sequence container
`utility`	Defines several utility components
`vector`	A vector sequence container

Table 42: Standard template library header files

Using standard C libraries in C++

The C++ library works in conjunction with 15 of the header files from the standard C library, sometimes with small alterations. The header files come in two forms—new and traditional—for example, `cassert` and `assert.h`.

The following table shows the new header files:

Header file	Usage
`cassert`	Enforcing assertions when functions execute
`cctype`	Classifying characters
`cerrno`	Testing error codes reported by library functions
`cfloat`	Testing floating-point type properties
`cinttypes`	Defining formatters for all types defined in `stdint.h`
`climits`	Testing integer type properties
`clocale`	Adapting to different cultural conventions
`cmath`	Computing common mathematical functions
`csetjmp`	Executing non-local goto statements
`csignal`	Controlling various exceptional conditions
`cstdarg`	Accessing a varying number of arguments
`cstdbool`	Adds support for the `bool` data type in C.
`cstddef`	Defining several useful types and macros
`cstdint`	Providing integer characteristics
`cstdio`	Performing input and output
`cstdlib`	Performing a variety of operations
`cstring`	Manipulating several kinds of strings
`ctime`	Converting between various time and date formats
`cwchar`	Support for wide characters
`cwctype`	Classifying wide characters

Table 43: New standard C header files—DLIB

LIBRARY FUNCTIONS AS INTRINSIC FUNCTIONS

Certain C library functions will under some circumstances be handled as intrinsic functions and will generate inline code instead of an ordinary function call, for example `memcpy`, `memset`, and `strcat`.

ADDED C FUNCTIONALITY

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

The following include files provide these features:

- `ctype.h`
- `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 definded:

`__write_array`	Corresponds to `fwrite` on `stdout`.
`__ungetchar`	Corresponds to `ungetc` on `stdout`.
`__gets`	Corresponds to `fgets` on `stdin`.

stdlib.h

In `stdlib.h`, the following C99 functions are defined:

`_Exit`, `llabs`, `lldiv`, `strtoll`, `strtoull`, `atoll`, `strtof`, `strtold`.

The function `strtod` has added functionality from the C99 standard. For reference information about this functions, see the library reference available from the **Help** menu.

The `__qsortbbl` 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:

`vfwscanf`, `vswscanf`, `vwscanf`, `wcstof`, `wcstolb`.

wctype.h

In `wctype.h`, the C99 function `iswblank` is defined.

IAR CLIB Library

The IAR CLIB Library provides most of the important C library definitions that apply to embedded systems. These are of the following types:

- Standard C library definitions available for user programs. These are documented in this chapter.

- The system startup code. It is described in the chapter *The CLIB runtime environment* in this guide.
- Runtime support libraries; for example low-level floating-point routines.
- Intrinsic functions, allowing low-level use of M16C/R8C Series features. See the chapter *Intrinsic functions* for more information.

LIBRARY DEFINITIONS SUMMARY

This following table lists the header files specific to the CLIB library:

Header file	Description
`assert.h`	Assertions
`ctype.h*`	Character handling
`errno.h`	Error return values
`float.h`	Limits and sizes of floating-point types
`iccbutl.h`	Low-level routines
`limits.h`	Limits and sizes of integral types
`math.h`	Mathematics
`setjmp.h`	Non-local jumps
`stdarg.h`	Variable arguments
`stdbool.h`	Adds support for the `bool` data type in C
`stddef.h`	Common definitions including `size_t`, `NULL`, `ptrdiff_t`, and `offsetof`
`stdio.h`	Input/output
`stdlib.h`	General utilities
`string.h`	String handling

Table 44: IAR CLIB Library header files

*** The functions `is`*`xxx`*, `toupper`, and `tolower` declared in the header file `ctype.h` evaluate their argument more than once. This is not according to the ISO/ANSI standard.**

Segment reference

The compiler places code and data into named segments which are referred to by the IAR XLINK Linker. Details about the segments are required for programming assembler language modules, and are also useful when interpreting the assembler language output from the compiler.

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

Summary of segments

The table below lists the segments that are available in the compiler:

Segment	Description
`BITVARS`	Holds bit variables.
`CHECKSUM`	Holds the checksum generated by the linker.
`CODE`	Holds the program code.
`CSTACK`	Holds the stack used by C or C++ programs.
`CSTART`	Holds the startup code.
`DATA13_AC`	Holds `__data13` located constant data.
`DATA13_AN`	Holds `__data13` located uninitialized data.
`DATA13_C`	Holds `__data13` constant data.
`DATA13_I`	Holds `__data13` static and global initialized variables.
`DATA13_ID`	Holds initial values for `__data13` static and global variables in `DATA13_I`.
`DATA13_N`	Holds `__no_init __data13` static and global variables.
`DATA13_Z`	Holds zero-initialized `__data13` static and global variables.
`DATA16_AC`	Holds `__data16` located constant data.
`DATA16_AN`	Holds `__data16` located uninitialized data.
`DATA16_C`	Holds `__data16` constant data.
`DATA16_HEAP`	Holds the heap used for dynamically allocated data in data16 memory.
`DATA16_I`	Holds `__data16` static and global initialized variables.
`DATA16_ID`	Holds initial values for `__data16` static and global variables in `DATA16_I`.

Table 45: Segment summary

Segment	Description
DATA16_N	Holds __no_init __data16 static and global variables.
DATA16_Z	Holds zero-initialized __data16 static and global variables.
DATA20_AC	Holds __data20 located constant data.
DATA20_AN	Holds __data20 located uninitialized data.
DATA20_C	Holds __data20 constant data.
DATA20_HEAP	Holds the heap used for dynamically allocated data in data20 memory.
DATA20_I	Holds __data20 static and global initialized variables.
DATA20_ID	Holds initial values for __data20 static and global variables in DATA20_I.
DATA20_N	Holds __no_init __data20 static and global variables.
DATA20_Z	Holds zero-initialized __data20 static and global variables.
DIFUNCT	Holds pointers to code, typically C++ constructors, that should be executed by the system startup code before main is called.
FAR_AC	Holds __far located constant data.
FAR_AN	Holds __far located uninitialized data.
FAR_C	Holds __far constant data.
FAR_HEAP	Holds the heap used for dynamically allocated data in far memory.
FAR_I	Holds __far static and global initialized variables.
FAR_ID	Holds initial values for __far static and global variables in FAR_I.
FAR_N	Holds __no_init __far static and global variables.
FAR_Z	Holds zero-initialized __far static and global variables.
FLIST	Holds the jump table for __tiny_func functions.
HEAP	Holds the heap used for dynamically allocated data.
INTVEC	Contains the reset and interrupt vectors.
INTVEC1	Contains the fixed reset and interrupt vectors.
ISTACK	Holds the stack used by interrupts and exceptions.
TINYFUNC	Holds __tiny_func declared functions to be placed in the special page area.

Table 45: Segment summary (Continued)

Descriptions of segments

This section gives reference information about each segment.

The segments are placed in memory by the segment placement linker directives `-Z` and `-P`, for sequential and packed placement, respectively. Some segments cannot use packed placement, as their contents must be continuous.

In each description, the segment memory type—`BIT`, `CODE`, `CONST`, `DATA`, `FARCONST`, `HUGECONST`, `NEARCONST`, `FARDATA`, `HUGEDATA`, or `NEARDATA`—indicates whether the segment should be placed in ROM or RAM memory; see Table 6, *XLINK segment memory types*, page 32.

For information about the `-Z` and the `-P` directives, see the *IAR Linker and Library Tools Reference Guide.*

For information about how to define segments in the linker command file, see *Customizing the linker command file*, page 33.

For detailed information about the extended keywords mentioned here, see the chapter *Extended keywords*.

BITVARS

Description	Holds bit variables.
Segment memory type	`BIT`
Memory placement	`0-0x1FFF`
Access type	Read/write

CHECKSUM

Description	Holds the checksum bytes generated by the linker. This segment also holds the `__checksum` symbol. Note that the size of this segment is affected by the linker option `-J`.
Segment memory type	`CONST`
Memory placement	This segment can be placed anywhere in ROM memory.
Access type	Read-only

CODE

Description	Holds program code, except the code for system initialization.
Segment memory type	`CODE`
Memory placement	`0-0xFFFFF`
Access type	Read-only
See also	*Calling convention*, page 93, *Calling assembler routines from C*, page 90, and *Calling assembler routines from C++*, page 92.

CSTACK

Description	Holds the internal data stack.
Segment memory type	`NEARDATA`
Memory placement	`0-0xFFFF`
Access type	Read/write
See also	*The stack*, page 38.

CSTART

Description	Holds the startup code. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`CODE`
Memory placement	This segment must be placed at the address where the chip starts executing after reset.
Access type	Read-only

DATA13_AC

Description	Holds `__data13` located constant data. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

DATA13_AN

Description	Holds `__no_init __data13` located data. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

DATA13_C

Description	Holds `__data13` constant data.
Segment memory type	`NEARCONST`
Memory placement	`0-0x1FFF`
Access type	Read-only

DATA13_I

Description	Holds `__data13` static and global initialized variables initialized by copying from the segment `DATA13_ID` at application startup. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`NEARDATA`
Memory placement	`0-0x1FFF`
Access type	Read/write

DATA13_ID

Description	Holds initial values for `__data13` static and global variables in the `DATA13_I` segment. These values are copied from `DATA13_ID` to `DATA13_I` at application startup. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`FARCONST`
Memory placement	`0-0xFFFFF`
Access type	Read-only

DATA13_N

Description	Holds static and global `__no_init __data13` variables.
Segment memory type	`NEARDATA`
Memory placement	`0-0x1FFF`
Access type	Read/write

DATA13_Z

Description	Holds zero-initialized `__data13` static and global variables. The contents of this segment is declared by the system startup code. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`NEARDATA`
Memory placement	`0-0x1FFF`
Access type	Read/write

DATA16_AC

Description	Holds `__data16` located constant data. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

DATA16_AN

Description	Holds `__no_init __data16` located data. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

DATA16_C

Description	Holds `__data16` constant data.
Segment memory type	`NEARCONST`
Memory placement	`0-0xFFFF`
Access type	Read-only

DATA16_HEAP

Description	Holds the heap used for dynamically allocated data in data16 memory, in other words data allocated by `data16_malloc` and `data16_free`, and in C++, `new` and `delete`. **Note:** This segment is only used when you use the DLIB library.
Segment memory type	`DATA`
Memory placement	`0-0xFFFF`
Access type	Read/write
See also	*The heap*, page 40 and *New and Delete operators*, page 115.

DATA16_I

Description	Holds `__data16` static and global initialized variables initialized by copying from the segment `DATA16_ID` at application startup. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`NEARDATA`
Memory placement	`0-0xFFFF`
Access type	Read/write

DATA16_ID

Description	Holds initial values for `__data16` static and global variables in the `DATA16_I` segment. These values are copied from `DATA16_ID` to `DATA16_I` at application startup. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`FARCONST`
Memory placement	`0-0xFFFFF`
Access type	Read-only

DATA16_N

Description	Holds static and global `__no_init __data16` variables.
Segment memory type	`NEARDATA`
Memory placement	`0-0xFFFF`
Access type	Read/write

DATA16_Z

Description	Holds zero-initialized `__data16` static and global variables. The contents of this segment is declared by the system startup code. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`NEARDATA`
Memory placement	`0-0xFFFF`
Access type	Read/write

DATA20_AC

Description	Holds `__data20` located constant data. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

DATA20_AN

Description	Holds `__no_init __data20` located data. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

DATA20_C

Description	Holds `__data20` constant data.
Segment memory type	`HUGECONST`
Memory placement	`0-0xFFFFF`
Access type	Read-only

DATA20_HEAP

Description	Holds the heap used for dynamically allocated data in data20 memory, in other words data allocated by `data20_malloc` and `data20_free`, and in C++, `new` and `delete`. **Note:** This segment is only used when you use the DLIB library.
Segment memory type	`DATA`
Memory placement	`0-0xFFFFF`
Access type	Read/write
See also	*The heap*, page 40 and *New and Delete operators*, page 115.

DATA20_I

Description	Holds `__data20` static and global initialized variables initialized by copying from the segment `DATA20_ID` at application startup. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`DATA`
Memory placement	`0-0xFFFFF`
Access type	Read/write

DATA20_ID

Description	Holds initial values for `__data20` static and global variables in the `DATA20_I` segment. These values are copied from `DATA20_ID` to `DATA20_I` at application startup. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`CONST`
Memory placement	`0-0xFFFFF`

Access type	Read-only

DATA20_N

Description	Holds static and global `__no_init __data20` variables.
Segment memory type	`DATA`
Memory placement	`0-0xFFFFF`
Access type	Read/write

DATA20_Z

Description	Holds zero-initialized `__data20` static and global variables. The contents of this segment is declared by the system startup code. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`DATA`
Memory placement	`0-0xFFFFF`
Access type	Read/write

DIFUNCT

Description	Holds the dynamic initialization vector used by C++.
Segment memory type	`FARCONST` **Note:** The reason why this segment is of the `FARCONST` type is that the *far* access method is used by the initializing code in `cstartup`.
Memory placement	`0-0xFFFFF`
Access type	Read-only

FAR_AC

Description	Holds `__far` located constant data. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

FAR_AN

Description	Holds `__no_init __far` located data. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive. Because the location is known, this segment does not need to be specified in the linker command file.

FAR_C

Description	Holds `__far` constant data.
Segment memory type	`FARCONST`
Memory placement	`0-0xFFFFF`
Access type	Read-only

FAR_HEAP

Description	Holds the heap used for dynamically allocated data in far memory, in other words data allocated by `far_malloc` and `far_free`, and in C++, `new` and `delete`. **Note:** This segment is only used when you use the DLIB library.
Segment memory type	`DATA`
Memory placement	`0-0xFFFFF`
Access type	Read/write
See also	*The heap*, page 40 and *New and Delete operators*, page 115.

FAR_I

Description	Holds `__far` static and global initialized variables initialized by copying from the segment `FAR_ID` at application startup. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`FARDATA`
Memory placement	`0-0xFFFFF`
Access type	Read/write

FAR_ID

Description	Holds initial values for `__far` static and global variables in the `FAR_I` segment. These values are copied from `FAR_ID` to `FAR_I` at application startup. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`FARCONST`
Memory placement	`0-0xFFFFF`
Access type	Read-only

FAR_N

Description	Holds static and global `__no_init __far` variables. **Note:** This segment is only available when the compiler is used in M16C mode, that is when the option `--cpu=M16C` is used.
Segment memory type	`FARDATA`
Memory placement	`0-0xFFFFF`
Access type	Read/write

FAR_Z

Description	Holds zero-initialized `__far` static and global variables. The contents of this segment is declared by the system startup code. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker command file, the `-Z` directive must be used.
Segment memory type	`FARDATA`
Memory placement	`0-0xFFFFF`
Access type	Read/write

FLIST

Description	Holds the jump table with an entry for each `__tiny_func` function. Each entry is two bytes and contains the lower 16 bits for the destination address. The high 4 bits are always `F`. **Note:** This segment is only available when the compiler is used in M16C mode, that is when the option `--cpu=M16C` is used.
Segment memory type	`CONST`
Memory placement	`0xFFE00-0xFFFDB`
Access type	Read-only

HEAP

Description

Holds the heap used for dynamically allocated data, in other words data allocated by `malloc` and `free`, and in C++, `new` and `delete`. This segment is used under any of these conditions:

- when the compiler is used in R8C mode, that is when the option `--cpu=R8C` is used
- when the IAR CLIB Library is used
- when the compiler is used in M16C mode, that is when the option `--cpu=M16C` is used; and when the IAR DLIB Library is used; and finally, if the *near* data model is used.

	This segment is an alias for the `DATA16_HEAP` segment.
Segment memory type	Depends on the data model. In the supplied linker command files, the type is `NEAR`, because no M16C/R8C CPU currently has RAM outside data16 memory.
Memory placement	Depends on the data model.
Access type	Read/write
See also	*The heap*, page 40.

INTVEC

Description	Holds the interrupt vector table populated by the use of the `__interrupt` extended keyword in combination with the `#pragma vector` directive.
Segment memory type	`CONST`
Memory placement	`0-0xFFFFF`
Access type	Read-only

INTVEC1

Description

Holds the fixed interrupt vector table populated by the use of the `__interrupt` extended keyword. The function must have one of these names:

```
__undefined_instruction_handler
__overflow_handler
__break_instruction_handler
__address_match_handler
__single_step_handler
__watchdog_timer_handler
__DBC_handler
__NMI_handler
```

Note: The interrupt sources Watchdog Timer, Oscillation Stop, Re-Oscillation Detection, and Voltage Down Detection share the same interrupt vector.

The vector entries consist of the addresses of the named functions. Thus, each of these named functions is represented by a vector entry.

Segment memory type `CONST`

Memory placement	● When the compiler is used in M16C mode: `0xFFFDC-0xFFFFF` ● When the compiler is used in R8C mode: `0xFFDC-0xFFFF`
Access type	Read-only

ISTACK

Description	Holds the internal stack used by interrupts and exceptions.
Segment memory type	`NEARDATA`
Memory placement	`0-0xFFFF`
Access type	Read/write
See also	*The stack*, page 38.

TINYFUNC

Description	Holds `__tiny_func` declared functions to be placed in the special page area. **Note:** This segment is only available when the compiler is used in M16C mode, that is when the option `--cpu=M16C` is used.
Segment memory type	`CODE`
Memory placement	`FFE00-FFFDB`
Access type	Read-only

Implementation-defined behavior

This chapter describes how the compiler handles the implementation-defined areas of the C language.

ISO 9899:1990, the International Organization for Standardization standard - *Programming Languages - C* (revision and redesign of ANSI X3.159-1989, American National Standard), changed by the ISO Amendment 1:1994, *Technical Corrigendum 1,* and *Technical Corrigendum 2*, contains an appendix called *Portability Issues*. The ISO appendix lists areas of the C language that ISO leaves open to each particular implementation.

Note: The compiler adheres to a freestanding implementation of the ISO standard for the C programming language. This means that parts of a standard library can be excluded in the implementation.

Descriptions of implementation-defined behavior

This section follows the same order as the ISO appendix. Each item covered includes references to the ISO chapter and section (in parenthesis) that explains the implementation-defined behavior.

Translation

Diagnostics (5.1.1.3)

Diagnostics are produced in the form:

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

where `filename` is the name of the source file in which the error was encountered, `linenumber` is the line number at which the compiler detected the error, `level` is the level of seriousness of the message (remark, warning, error, or fatal error), `tag` is a unique tag that identifies the message, and `message` is an explanatory message, possibly several lines.

Environment

Arguments to main (5.1.2.2.2.1)

The function called at program startup is called `main`. No prototype was declared for `main`, and the only definition supported for `main` is:

```
int main(void)
```

To change this behavior for the IAR DLIB runtime environment, see *Customizing system initialization*, page 61. To change this behavior for the IAR CLIB runtime environment, see *Customizing system initialization*, page 84.

Interactive devices (5.1.2.3)

The streams `stdin` and `stdout` are treated as interactive devices.

Identifiers

Significant characters without external linkage (6.1.2)

The number of significant initial characters in an identifier without external linkage is 200.

Significant characters with external linkage (6.1.2)

The number of significant initial characters in an identifier with external linkage is 200.

Case distinctions are significant (6.1.2)

Identifiers with external linkage are treated as case-sensitive.

Characters

Source and execution character sets (5.2.1)

The source character set is the set of legal characters that can appear in source files. The default source character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the source character set will be the host computer's default character set.

The execution character set is the set of legal characters that can appear in the execution environment. The default execution character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the execution character set will be the host computer's default character set. The IAR DLIB Library needs a multibyte character scanner to support a multibyte execution character set. The IAR CLIB Library does not support multibyte characters.

See *Locale*, page 66.

Bits per character in execution character set (5.2.4.2.1)

The number of bits in a character is represented by the manifest constant `CHAR_BIT`. The standard include file `limits.h` defines `CHAR_BIT` as 8.

Mapping of characters (6.1.3.4)

The mapping of members of the source character set (in character and string literals) to members of the execution character set is made in a one-to-one way. In other words, the same representation value is used for each member in the character sets except for the escape sequences listed in the ISO standard.

Unrepresented character constants (6.1.3.4)

The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or in the extended character set for a wide character constant generates a diagnostic message, and will be truncated to fit the execution character set.

Character constant with more than one character (6.1.3.4)

An integer character constant that contains more than one character will be treated as an integer constant. The value will be calculated by treating the leftmost character as the most significant character, and the rightmost character as the least significant character, in an integer constant. A diagnostic message will be issued if the value cannot be represented in an integer constant.

A wide character constant that contains more than one multibyte character generates a diagnostic message.

Converting multibyte characters (6.1.3.4)

The only locale supported—that is, the only locale supplied with the IAR C/C++ Compiler—is the 'C' locale. If you use the command line option `--enable_multibytes`, the IAR DLIB Library will support multibyte characters if you add a locale with multibyte support or a multibyte character scanner to the library. The IAR CLIB Library does not support multibyte characters.

See *Locale*, page 66.

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 184, 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 186, 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 188, for information about `size_t`.

Conversion from/to pointers (6.3.4)

See *Casting*, page 188, for information about casting of data pointers and function pointers.

ptrdiff_t (6.3.6, 7.1.1)

See *ptrdiff_t*, page 188, 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 184, for information about the alignment requirement for data objects.

Sign of 'plain' bitfields (6.5.2.1)

A 'plain' `int` bitfield is treated as a signed `int` bitfield. All integer types are allowed as bitfields.

Allocation order of bitfields within a unit (6.5.2.1)

Bitfields are allocated within an integer from least-significant to most-significant bit.

Can bitfields straddle a storage-unit boundary (6.5.2.1)

Bitfields cannot straddle a storage-unit boundary for the chosen bitfield integer type.

Integer type chosen to represent enumeration types (6.5.2.2)

The chosen integer type for a specific enumeration type depends on the enumeration constants defined for the enumeration type. The chosen integer type is the smallest possible.

Qualifiers

Access to volatile objects (6.5.3)

Any reference to an object with volatile qualified type is an access.

Declarators

Maximum numbers of declarators (6.5.4)

The number of declarators is not limited. The number is limited only by the available memory.

Statements

Maximum number of case statements (6.6.4.2)

The number of case statements (case values) in a switch statement is not limited. The number is limited only by the available memory.

Preprocessing directives

Character constants and conditional inclusion (6.8.1)

The character set used in the preprocessor directives is the same as the execution character set. The preprocessor recognizes negative character values if a 'plain' character is treated as a `signed` character.

Including bracketed filenames (6.8.2)

For file specifications enclosed in angle brackets, the preprocessor does not search directories of the parent files. A parent file is the file that contains the `#include` directive. Instead, it begins by searching for the file in the directories specified on the compiler command line.

Including quoted filenames (6.8.2)

For file specifications enclosed in quotes, the preprocessor directory search begins with the directories of the parent file, then proceeds through the directories of any grandparent files. Thus, searching begins relative to the directory containing the source file currently being processed. If there is no grandparent file and the file is not found, the search continues as if the filename was enclosed in angle brackets.

Character sequences (6.8.2)

Preprocessor directives use the source character set, except for escape sequences. Thus, to specify a path for an include file, use only one backslash:

```
#include "mydirectory\myfile"
```

Within source code, two backslashes are necessary:

```
file = fopen("mydirectory\\myfile","rt");
```

Recognized pragma directives (6.8.6)

In addition to the pragma directives described in the chapter *Pragma directives*, the following directives are recognized and will have an indeterminate effect:

```
alignment
baseaddr
building_runtime
can_instantiate
codeseg
cspy_support
define_type_info
```

```
do_not_instantiate
early_dynamic_initialization
function
hdrstop
important_typedef
instantiate
keep_definition
memory
module_name
no_pch
once
__printf_args
public_equ
__scanf_args
section
STDC
system_include
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 69.

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 65.

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 65.

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 65.

%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 68.

system() (7.10.4.5)

How the command processor works depends on how you have implemented the `system` function. See *Environment interaction*, page 68.

Message returned by strerror() (7.11.6.2)

The messages returned by `strerror()` depending on the argument is:

Argument	Message
`EZERO`	`no error`
`EDOM`	`domain error`
`ERANGE`	`range error`
`EFPOS`	`file positioning error`
`EILSEQ`	`multi-byte encoding error`
`<0 || >99`	`unknown error`
all others	`error` *`nnn`*

Table 46: Message returned by strerror()—IAR DLIB library

The time zone (7.12.1)

The local time zone and daylight savings time implementation is described in *Time*, page 70.

clock() (7.12.2.1)

From where the system clock starts counting depends on how you have implemented the `clock` function. See *Time*, page 70.

IAR CLIB Library functions

NULL macro (7.1.6)

The `NULL` macro is defined to `(void *) 0.`

Diagnostic printed by the assert function (7.2)

The `assert()` function prints:

```
Assertion failed: expression, file Filename, line linenumber
```

when the parameter evaluates to zero.

Domain errors (7.5.1)

HUGE_VAL, the largest representable value in a double floating-point type, will be returned by the mathematic functions on domain errors.

Underflow of floating-point values sets errno to ERANGE (7.5.1)

The mathematics functions set the integer expression errno to ERANGE (a macro in errno.h) on underflow range errors.

fmod() functionality (7.5.6.4)

If the second argument to fmod() is zero, the function returns zero (it does not change the integer expression errno).

signal() (7.7.1.1)

The signal part of the library is not supported.

Terminating newline character (7.9.2)

stdout stream functions recognize either newline or end of file (EOF) as the terminating character for a line.

Blank lines (7.9.2)

Space characters written to the stdout stream immediately before a newline character are preserved. There is no way to read the line through the stdin stream that was written through the stdout stream.

Null characters appended to data written to binary streams (7.9.2)

There are no binary streams implemented.

Files (7.9.3)

There are no other streams than stdin and stdout. This means that a file system is not implemented.

remove() (7.9.4.1)

There are no other streams than stdin and stdout. This means that a file system is not implemented.

rename() (7.9.4.2)

There are no other streams than `stdin` and `stdout`. This means that a file system is not implemented.

%p in printf() (7.9.6.1)

The argument to a `%p` conversion specifier, print pointer, to `printf()` is treated as having the type `'char *'`. The value will be printed as a hexadecimal number, similar to using the `%x` conversion specifier.

%p in scanf() (7.9.6.2)

The `%p` conversion specifier, scan pointer, to `scanf()` reads a hexadecimal number and converts it into a value with the type `'void *'`.

Reading ranges in scanf() (7.9.6.2)

A - (dash) character is always treated explicitly as a - character.

File position errors (7.9.9.1, 7.9.9.4)

There are no other streams than `stdin` and `stdout`. This means that a file system is not implemented.

Message generated by perror() (7.9.10.4)

`perror()` is not supported.

Allocating zero bytes of memory (7.10.3)

The `calloc()`, `malloc()`, and `realloc()` functions accept zero as an argument. Memory will be allocated, a valid pointer to that memory is returned, and the memory block can be modified later by `realloc`.

Behavior of abort() (7.10.4.1)

The `abort()` function does not flush stream buffers, and it does not handle files, because this is an unsupported feature.

Behavior of exit() (7.10.4.3)

The `exit()` function does not return.

Environment (7.10.4.4)

Environments are not supported.

system() (7.10.4.5)

The `system()` function is not supported.

Message returned by strerror() (7.11.6.2)

The messages returned by `strerror()` depending on the argument are:

Argument	Message
`EZERO`	`no error`
`EDOM`	`domain error`
`ERANGE`	`range error`
`<0 || >99`	`unknown error`
all others	`error No.`*`xx`*

Table 47: Message returned by strerror()—IAR CLIB library

The time zone (7.12.1)

The time zone function is not supported.

clock() (7.12.2.1)

The `clock()` function is not supported.