ColdFire® IAR C/C++ Compiler

Reference Guide

for Freescale's

ColdFire Microcontroller Family

EDITION NOTICE

Second edition: January 2008

Part number: CCF-2

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

Internal reference: ISUD.

Brief contents

Contents

Tables

Preface

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

Who should read this guide

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

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

How to use this guide

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

When you are familiar with the compiler and have already configured your project, you can focus more on *Part 2. 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 IAR Embedded Workbench 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 ColdFire IAR C/C++ Compiler for efficiently developing your application.
- *Data storage* describes how data can be stored in memory, with emphasis on the different data memory type attributes.
- *Functions* gives a brief overview of function-related extensions—mechanisms for controlling functions—and describes some of these mechanisms in more detail.
- *Placing code and data* describes the concept of segments, introduces the linker command file, and describes how code and data are placed in memory.
- *The DLIB runtime environment* describes the runtime environment in which an application executes. It covers how you can modify it by setting options, overriding default library modules, or building your own library. The chapter also describes system initialization and introduces the file `cstartup`, as well as how to use modules for locale, and file I/O.
- *Assembler language interface* contains information required when parts of an application are written in assembler language. This includes the calling convention.
- *Using C++* gives an overview of the two levels of C++ support: The industry-standard EC++ and IAR Extended EC++.
- *Efficient coding for embedded applications* gives hints about how to write code that compiles to efficient code for an embedded application.

Part 2. 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 ColdFire-specific keywords that are extensions to the standard C/C++ language.
- *Pragma directives* gives reference information about the pragma directives.
- *Intrinsic functions* gives reference information about the functions that can be used for accessing ColdFire-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 ColdFire IAR C/C++ Compiler. handles the implementation-defined areas of the C language standard.

Other documentation

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

- Using the IAR Embedded Workbench® IDE with the IAR C-SPY Debugger®, refer to the *IAR Embedded Workbench® IDE User Guide*
- Programming for the ColdFire IAR Assembler, refer to the *ColdFire® 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
- Developing safety-critical applications using the MISRA C guidelines, refer to the *IAR Embedded Workbench® MISRA C Reference Guide*.

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

FURTHER READING

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

- Barr, Michael, and Andy Oram, ed. *Programming Embedded Systems in C and C++*. O'Reilly & Associates.

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

We recommend that you visit the following web sites:

- The Freescale web site, **www.freescale.com**, contains information and news about the ColdFire microcontrollers.
- 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.

TYPOGRAPHIC CONVENTIONS

This guide uses the following typographic conventions:

Style	Used for
`computer`	Text that you enter or that appears on the screen.
`parameter`	A label representing the actual value you should enter as part of a command.
`[option]`	An optional part of a command.
`{option}`	A mandatory part of a command.
`a|b|c`	Alternatives in a command.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
reference	A cross-reference within this guide or to another guide.

Table 1: Typographic conventions used in this guide

Style	Used for
...	An ellipsis indicates that the previous item can be repeated an arbitrary number of times.
🛠	Identifies instructions specific to the IAR Embedded Workbench interface.
🖥	Identifies instructions specific to the command line interface.
💡	Identifies helpful tips and programming hints.

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

Part 1. Using the compiler

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

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

2

Getting started

This chapter gives the information you need to get started using the ColdFire IAR C/C++ Compiler for efficiently developing your application.

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

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

IAR language overview

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

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

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

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

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

Supported ColdFire devices

The ColdFire IAR C/C++ Compiler supports all families based on the Freescale ColdFire V1 and V2 cores. The object code that the compiler generates is binary compatible between the cores as long as the same instruction set architecture is used. For more details, see *Processor configuration*, page 6.

Building applications—an overview

A typical application is built from a number of source files and libraries. The source files can be written in C, C++, or assembler language, and can be compiled into object files by the ColdFire IAR C/C++ Compiler or the ColdFire IAR 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 IAR Embedded Workbench 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.r68` using the default settings:

```
icccf myfile.c
```

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

LINKING

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

- A number of object files and possibly certain libraries
- The standard library containing the runtime environment and the standard language functions
- A program start label
- A linker command file that describes the 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.r68 myfile2.r68 -s __program_start -f lnkM5213.xcl
dlcfaffn.r68 -o aout.a68 -r
```

In this example, `myfile.r68` and `myfile2.r68` are object files, `lnkM5213.xcl` is the linker command file, and `dlcfaffn.r68` is the runtime library. The option `-s` specifies the label where the application starts. The option `-o` specifies the name of the output file, and the option `-r` is used for specifying the output format UBROF, which can be used for debugging in C-SPY®.

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

Basic settings for project configuration

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

The basic settings are:

- Processor configuration, which includes core, instruction set architecture, division instructions
- Code model
- Data model

- Optimization settings
- Runtime environment.

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

PROCESSOR CONFIGURATION

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

Core

The ColdFire IAR C/C++ Compiler supports the ColdFire V1 and V2 cores. The object code that the compiler generates is binary compatible between the cores as long as the same instruction set architecture is used. Therefore it is crucial to specify a core option to the compiler.

In the IAR Embedded Workbench IDE, choose **Project>Options>General Options>Target>Device.**

Use the `--core={v1|v2}` option to select the core for which the code is to be optimized.

The `--core` option for example controls speed optimizations.

Instruction set architecture

The ColdFire IAR C/C++ Compiler supports the isa_a, isa_a+, isa_b, and the isa_c instruction set architectures.

In the IAR Embedded Workbench IDE, choose **Project>Options>General Options>Target>ISA variant.**

Use the `--isa={isa_a|isa_a+|isa_b|isa_c}` option to select the instruction set for which the code is to be generated.

Division instructions

Some ColdFire devices do not support the `DIVS`/`DIVU` and `REMS`/`REMU` instructions, for example V1 devices. If you are using such a device, you must specify to the compiler not to generate code that uses these instructions.

In the IAR Embedded Workbench IDE, choose **Project>Options>General Options>Target>No DIV/REM instruction.**

Use the `--no_div` option to disable support for division instructions.

CODE MODEL

The ColdFire IAR C/C++ Compiler supports *code models* that you can set on file- or function-level to control which function calls are generated, which determines where in memory functions are placed. The following code models are available:

- The *Near* code model has an upper limit of 32 Kbytes
- The *Far* code model can access the entire 32-bit address space.

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

DATA MODEL

One of the characteristics of the ColdFire microcontroller is that there is a trade-off regarding the way 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 ColdFire IAR C/C++ Compiler, you can set a default memory access method by selecting a *data model*. The following data models are supported:

- The *Near relative* data model places objects in a position-independent 64-Kbyte memory block that can be placed anywhere in the entire memory area
- The *Far* data model can access the entire memory area.

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

OPTIMIZATION FOR SPEED AND SIZE

The ColdFire IAR C/C++ Compiler is a state-of-the-art compiler with an optimizer that performs, among other things, dead-code elimination, constant propagation, inlining, and common sub-expression elimination. It also performs loop optimizations, such as unrolling and induction variable elimination.

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

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

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

RUNTIME ENVIRONMENT

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

The runtime library provided is the IAR DLIB Library, which supports ISO/ANSI C and C++. This library also supports floating-point numbers in IEEE 754 format and it can be configured to include different levels of support for locale, file descriptors, multibyte characters, et cetera.

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

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

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

Choosing a runtime library in the IAR Embedded Workbench IDE

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

Note that for the DLIB library there are different configurations—Normal and Full—which include different levels of support for locale, file descriptors, multibyte characters, et cetera. See *Library configurations*, page 43, 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 cf\inc`	Specifies the include paths
`*libraryfile*.r68`	Specifies the library object file
`--dlib_config C:\...\*configfile*.h`	Specifies the library configuration file

Table 2: Command line options for specifying library and dependency files

For more information about prebuilt library object files for the IAR DLIB Library, see *Using a prebuilt library*, page 44. 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 46.
- The size of the stack and the heap, see *The stack*, page 33, and *The heap*, page 35, respectively.

Special support for embedded systems

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

EXTENDED KEYWORDS

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

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

The command line option `-e` makes the extended keywords available, and reserves them so that they cannot be used as variable names. See, *-e*, page 126 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 ColdFire IAR C/C++ Compiler. They are consistent with ISO/ANSI C, and are very useful when you want to make sure that the source code is portable.

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

PREDEFINED SYMBOLS

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

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

SPECIAL FUNCTION TYPES

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

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

ACCESSING LOW-LEVEL FEATURES

For hardware-related parts of your application, accessing low-level features is essential. The ColdFire IAR C/C++ Compiler supports several ways of doing this: intrinsic functions, mixing C and assembler modules, and inline assembler. For information about the different methods, see *Mixing C and assembler*, page 69.

Data storage

This chapter gives a brief introduction to the memory layout of the ColdFire microcontroller and the fundamental ways data can be stored in memory: on the stack, in static (global) memory, or in heap memory. For efficient memory usage, ColdFire IAR C/C++ Compiler provides a set of data memory attributes, allowing you to fine-tune the access methods, resulting in smaller code size. The concepts of 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 ColdFire IAR C/C++ Compiler has one continuous 4-Gbyte memory space. The memory for the ColdFire microcontroller is divided into different memory areas, depending on access method. To read more about this, see *Memory types*, page 13.

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

- On the stack. This is memory space that can be used by a function as long as it is executing. When the function returns to its caller, the memory space is no longer valid.
- Static memory. This kind of memory is allocated once and for all; it remains valid through the entire execution of the application. Variables that are either global or declared static are placed in this type of memory. The word *static* in this context means that the amount of memory allocated for this type of variable does not change while the application is running.
- On the heap. Once memory has been allocated on the heap, it remains valid until it is explicitly released back to the system by the application. This type of memory is useful when the number of objects is not known until the application executes. Note that there are potential risks connected with using the heap in systems with a limited amount of memory, or systems that are expected to run for a long time.

Data models

The ColdFire IAR C/C++ Compiler supports data models for applications with different data requirements.

Technically, the data model specifies the default memory type. This means that the data model controls the default placement of static and global variables. 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 14.

SPECIFYING A DATA MODEL

Two data models are implemented: Near relative and Far. These models are controlled by the `--data_model` option. If you do not specify a data model option, the compiler will use the Far data model.

The following table summarizes the different data models:

Data model name	Default memory attribute	Default pointer attribute	Placement of data
Near relative	`__near_rel`	`__far`	A 64-Kbyte memory area that can be placed anywhere in memory.
Far	`__far`	`__far`	The entire 4-Gbyte memory

Table 3: Data model characteristics

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 by explicitly specifying a memory attribute, using either keywords or the `#pragma type_attribute` directive.

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

Use the `--data_model` option to specify the data model for your project; see *--data_model*, page 213.

The Near relative data model

The Near relative data model uses near relative addressing by default, which means that objects are placed in a position-independent 64-Kbyte memory area that can be placed anywhere in memory. The advantage is that the compiler can generate more compact code in most cases.

The Far data model

The Far data model places objects anywhere in the entire 4-Gbyte of memory and in contrast with the Near relative data model, there is no object size limitation.

Memory types

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

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

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

It is possible to specify—for individual variables—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 82.

NEAR

The near memory consists of the low and high 32 Kbytes of memory. In hexadecimal notation this is the addresses `0x00000000`–`0x00007FFF` and `0xFFFF8000`–`0xFFFFFFFF`.

This combination of memory ranges may at first sight seem odd. The explanation, however, is that when an address expression becomes negative, the calculation wraps around. Because the address space on the ColdFire microcontroller is 32 bits, the address below 0 can be seen as `0xFFFFFFFF`. Hence, an alternative way to see the memory range in the memory accessible is simply ±32 Kbytes around address 0.

Accessing near memory is more efficient considering size.

NEAR RELATIVE

The Near relative memory type refers to memory that uses relative addressing, which means a 64-Kbyte area of the memory can be accessed. This area can be placed at any location in memory. The advantage is that the compiler can generate more compact code in most cases.

FAR

The ColdFire microcontroller has an address space of up to 4 Gbytes. Using the far memory type, the data objects can be placed anywhere in memory. Also, unlike near relative memory, there is no limitation on the size of the objects that can be placed in far memory. The drawback of far memory is that the code generated to access the memory is larger than for near memory. The far memory is the default memory.

USING DATA MEMORY ATTRIBUTES

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

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

Memory type	Keyword	Pointer size	Placement of data
Near	`__near`	32 bits	Low 32 Kbytes or high 32 Kbytes.
Near relative	`__near_rel`	32 bits	A 64-Kbyte memory area that can be placed anywhere in memory.
Far (default)	`__far`	32 bits	The entire 4 Gbytes of memory.

Table 4: Memory types and their corresponding memory attributes

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

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

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

Use the `-e` compiler option to enable language extensions. See *-e*, page 126 for additional information.

For reference information about each keyword, see *Descriptions of extended keywords*, page 165.

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 161.

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

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

Note that the keyword affects both identifiers.

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

Type definitions

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

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

and

```
__near char b;
char __near *bp;
```

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 near memory.

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

The following declaration is incorrect:

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

MORE EXAMPLES

The following is a series of examples with descriptions. First, some integer variables are defined and then pointer variables are introduced. To read the following examples, start from the left and add one qualifier at each step

`int a;`	A variable defined in default memory.
`int __near b;`	A variable in near memory.
`__far int c;`	A variable in far memory.
`int * d;`	A pointer stored in default memory. The pointer points to an integer in default 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.

Example

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

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

  __far static int gamma;
};
```

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

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

The stack and auto variables

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

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

The stack can contain:

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

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

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

Advantages

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

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

Potential problems

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

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

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

Dynamic memory on the heap

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

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

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

Potential problems

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

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

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

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

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

Functions

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

Function-related extensions

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

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

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

For more information about optimizations, see *Writing efficient code*, page 102. For information about the available intrinsic functions for accessing hardware operations, see the chapter *Intrinsic functions*.

Code models and memory attributes for function storage

By means of *code models*, the ColdFire IAR C/C++ Compiler supports placing functions in a default part of memory, or in other words, use a default size of the function address. Technically, the code models control the following:

- The default memory range for storing the function
- The maximum module size
- The maximum application size
- The default memory attribute.

The compiler supports two code models—Near and Far. If you do not specify a code model, the compiler will use the Far code model as default. Your project can only use one code model at a time, and the same model must be used by all user modules and all library modules.

The following table summarizes the different code models:

Code model name	Default function type (memory attributes)
Near	`__near_func`
Far (default)	`__far_func`

Table 5: Code models

For more information about the function memory attributes, see Table 6, *Function memory attributes*.

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

Use the `--code_model` option to specify the code model for your project; see *--code_model*, page 120.

USING FUNCTION MEMORY ATTRIBUTES

It is possible to override the default placement for individual functions. You specify this by using the appropriate *function memory attribute*. The following attributes are available:

Function memory attribute	Address range	Pointer size	Default in code model	Description
`__near_func`	`0xFFFF8000-0x00007FFF`	4 bytes	Near	The function can be called from anywhere in memory.
`__far_func`	`0-0xFFFFFFFF`	4 bytes	Far	The function can be called from anywhere in memory.

Table 6: Function memory attributes

Pointers with function memory attributes have restrictions in implicit and explicit casts between pointers and between pointers and integer types. For details about the restrictions, see *Casting*, page 146.

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

Primitives for interrupts, concurrency, and OS-related programming

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

- The extended keywords `__interrupt` and `__monitor`

- The intrinsic functions `__disable_interrupt`, `__get_status_register`, and `__set_status_register`.

INTERRUPT FUNCTIONS

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

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

The ColdFire microcontroller supports many interrupt sources. For each interrupt source, an interrupt routine can be written. Each interrupt routine is associated with a vector number, which is specified in the ColdFire microcontroller documentation from the chip manufacturer. The interrupt vector is the offset into the interrupt vector table. For the ColdFire microcontroller, the interrupt vector table start address is device-specific.

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 is used. For example:

```
__interrupt void my_interrupt_routine(void)
{
  /* Do something */
}
```

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

To specify an interrupt vector to your interrupt routine, you can write a small assembler routine, for example:

```
        PUBLIC exampleVector
        EXTERN my_interrupt_routine
        COMMON INTVEC:CODE(2)
        ORG 0x120    /* Offset in vector table */
exampleVector:
        DC32 my_interrupt_routine
        END
```

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

MONITOR FUNCTIONS

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

To define a monitor function, you can use the `__monitor` keyword. For reference information, see *__monitor*, page 167.

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

Example of implementing a semaphore in C

In the following example, a semaphore is implemented using one static variable and two monitor functions. A semaphore can be locked by one process, and is used for preventing processes from simultaneously using resources that can only be used by one process at a time, for example a USART. The `__monitor` keyword assures that the lock operation is atomic; in other words it cannot be interrupted.

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

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

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


/* release_lock -- Unlock the lock. */

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

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

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

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

C++ AND SPECIAL FUNCTION TYPES

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

Placing code and data

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

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

Segments and memory

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

WHAT IS A SEGMENT?

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

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

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

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

Segment memory type

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

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

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

Table 7: XLINK segment memory types

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

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

Placing segments in memory

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

In particular, the linker command file specifies:

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

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

CUSTOMIZING THE LINKER COMMAND FILE

The `config` directory contains many ready-made linker command files (filename extension `xcl`). The files contain the information required by the linker, and is ready to be used.The only change you will normally have to make to the supplied linker command file is to customize it so it fits the target system memory map. If, for example, your application uses additional external RAM, you need to add details about the external RAM memory area.

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

Range	Type
`0x0000–0x3FFFF`	ROM
`0x20000000–0x20008000`	RAM

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

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

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

The contents of the linker command file

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

- The CPU used:

  ```
  -ccf
  ```

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

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

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

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

Using the -Z command for sequential placement

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

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

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

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

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

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

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

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

Using the -P command for packed placement

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

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

```
-P(DATA)MYDATA=0-1FFF,10000-11FFF
```

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

```
-P(DATA)MYDATA=0-1FFF,F000–F7FF,10000-11FFF
```

Data segments

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

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

STATIC MEMORY SEGMENTS

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

- Variables that are initialized to a non-zero value
- Variables that are initialized to zero
- Variables that are declared as `const` and therefore can be stored in ROM
- Variables defined with the `__no_init` keyword, meaning that they should not be initialized at all.

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

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

Segment naming

The names of the segments consist of two parts—the segment group name and a *suffix*—for instance, `NEAR_Z`. The names of the segment groups are derived from the memory type and the corresponding keyword, for example `NEAR` and `__near`.

There is a segment group for each memory type, where each segment group holds different categories of declared data.

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 28.

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

Categories of declared data	Segment memory type	Suffix
Non-initialized data	`DATA`	`N`
Zero-initialized data	`DATA`	`Z`
Non-zero initialized data	`DATA`	`I`
Initializers for the above	`CONST`	`ID`

Table 9: Segment name suffixes

Categories of declared data	Segment memory type	Suffix
Constants	CONST	C

Table 9: Segment name suffixes (Continued)

For a summary of all supported segments, see *Summary of segments*, page 203.

Examples

Assume the following examples:

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

`__no_init __near int j;`	The near non-initialized variables will be placed in the segment `NEAR_N`.
`__near int j = 4;`	The near non-zero initialized variables will be placed in the segment `NEAR_I`, and initializer data in segment `NEAR_ID`.

Initialized data

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

1 It clears the memory of the variables that should be initialized to zero.

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

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

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

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

NEAR_I	`0x1000-0x10FF` and `0x1200-0x12FF`
NEAR_ID	`0x4000-0x41FF`

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

NEAR_I	`0x1000-0x10FF` and `0x1200-0x12FF`
NEAR_ID	`0x4000-0x40FF` and `0x4200-0x42FF`

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 the following directives to place the static data segments:

```
//First, the segments to be placed in ROM are defined:
-Z(CONST)NEAR_C=2000-7FFF
-Z(CONST)FAR_C=20000-3FFFFF
-Z(CONST)FAR_ID,NEARPID_ID,NEAR_ID

//Then, the RAM data segments are placed in memory:
-Z(DATA)NEAR_I,NEAR_Z,NEAR_N=0-7FFF
-Z(DATA)PIDBASE,NEARPID_I,NEARPID_Z,NEARPID_N=20000-2FFFF
-Z(DATA)FAR_I,FAR_Z,FAR_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 `A7 (SP)`.

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

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

Stack size allocation in the IAR Embedded Workbench IDE

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

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

If the supervisor stack is to be used, enable the **Use supervisor stack** option and add the required stack size in the **Supervisor 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
```

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#20000000-20008000
```

Note:

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

Stack size considerations

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

THE HEAP

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

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

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

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

Heap size allocation in the IAR Embedded Workbench IDE

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

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

Heap size allocation from the command line

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

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

```
-D_HEAP_SIZE=size
```

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

Specify the appropriate size for your application.

Placement of heap segment

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

```
-Z(DATA)HEAP+_HEAP_SIZE=20000000-20008000
```

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

Heap size and standard I/O

If you have excluded `FILE` descriptors from the DLIB runtime environment, as in the normal configuration, there are no input and output buffers at all. Otherwise, as in the full configuration, be aware that the size of the input and output buffers is set to 512 bytes in the `stdio` library header file. If the heap is too small, I/O will not be buffered, which is considerably slower than when I/O is buffered. If you execute the application using the 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 ColdFire microcontroller. If you use the standard I/O library, you should set the heap size to a value which accommodates the needs of the standard I/O buffer.

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 203.

STARTUP CODE

The segment `RCODE` contains code used during system startup (`cstartup`), system termination (`cexit`), and other assembler routines. The system startup code can be placed anywhere. In addition, the segments must be placed into one continuous memory space, which means the `-P` segment directive cannot be used.

In the default linker command file, the following line will place the `RCODE` segment at the address `0x1000`:

```
-Z(CODE)RCODE=1000
```

NORMAL CODE

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

If you use the Near code model, or if the function is explicitly declared __near_func, the code is placed in `CODE` segment. If you use the Far code model, or if the function is explicitly declared `__far_func`, the code is placed in `FCODE` segment. Again, this is a simple operation in the linker command file:

```
-Z(CODE)CODE=0-7FFF
-Z(CODE)FCODE=0-FFFFFFFF
```

INTERRUPT VECTORS

The interrupt vector table contains pointers to interrupt routines, including the reset routine. The table should be placed in the segment `INTVEC`. For the ColdFire microcontroller, the interrupt vector table can be mapped to even 1-Mbyte aligned base addresses. To place the `INTVEC` segment in the memory range `0-3FF`, the linker directive looks like this:

```
-Z(CONST)INTVEC=_VBR_ADDRESS:+400
```

where `_VBR_ADDRESS` is the symbol holding the vector base address (by default set to `0`) and `400` is the length of the area.

C++ dynamic initialization

INITIALIZATION

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

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

For example:

```
-Z(CONST)EARLYDIFUNCT,DIFUNCT=0000-1FFFF
```

For additional information, see *DIFUNCT*, page 205 and *EARLYDIFUNCT*, page 205.

DESTRUCTION AND ATEXIT() HANDLING

Information about C++ objects to be destructed and the result of calls to `atexit` are allocated in the `.iar.dynexit` segment. The segment needs an entry for each dynamically initialized C++ object with a static life span and an entry for each call to `atexit` performed by your application.

The default linker command file sets up a symbol, representing the number of elements, at the beginning of the linker file:

```
-D_DYNEXIT_ELEMENTS=number_of_elements
```

You should specify the number appropriately depending on your application requirements.

In the linker command file, the segment is then allocated in the memory area available for it, for example:

```
-Z(DATA)...,.iar.dynexit+((_DYNEXIT_ELEMENTS+14)*C)=20000000
```

By default, the linker command file adds 20 entries (`14` in hexadecimal notation) required internally by the compilation system. The size of each entry is 12 bytes (`C` in hexadecimal notation).

If there are no more available entries in the `.iar.dynexit` segment when a new entry is required at runtime, either when initializing a C++ object with a static life span that also needs destruction, or due to a call to the `atexit` function, the program will terminate by calling the `abort` function.

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. It is also at link time it is known whether all segments will fit in the reserved memory ranges. If the contents of a segment do not fit in the address range defined in the linker command file, XLINK will issue a *segment too long* error.

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

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

LINKER MAP FILE

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

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

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

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

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

The DLIB runtime environment

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

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

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

Introduction to the runtime environment

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

This section gives an overview of:

- The runtime environment and its components
- Library selection.

RUNTIME ENVIRONMENT FUNCTIONALITY

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

The runtime library is delivered both as prebuilt libraries and as source files, and you can find them in the product subdirectories `cf\lib` and `cf\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
  - Target-specific arithmetic support modules like hardware multipliers or floating-point coprocessors.
- Runtime environment support, that is, startup and exit code and low-level interface to some library functions.
- Special compiler support for some functions, for instance functions for floating-point arithmetics.

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

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

LIBRARY SELECTION

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

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

- Setting library options, for example, for choosing `scanf` input and `printf` output formatters, and for specifying the size of the stack and the heap
- Overriding certain library functions, for example `cstartup.s68`, with your own customized versions
- Choosing the level of support for certain standard library functionality, for example, locale, file descriptors, and multibyte characters, by choosing a *library configuration*: normal or full.

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

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

SITUATIONS THAT REQUIRE LIBRARY BUILDING

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

You must build your own library when:

- 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 50.

LIBRARY CONFIGURATIONS

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

The following DLIB library configurations are available:

Library configuration	**Description**
Normal DLIB	No locale interface, C locale, no file descriptor support, no multibyte characters in `printf` and `scanf`, and no hex floats in `strtod`.
Full DLIB	Full locale interface, C locale, file descriptor support, multibyte characters in `printf` and `scanf`, and hex floats in `strtod`.

Table 10: Library configurations

In addition to these configurations, you can define your own configurations, which means that you must modify the configuration file. Note that the library configuration file describes how a library was built and thus cannot be changed unless you rebuild the library. For further information, see *Building and using a customized library*, page 50.

The prebuilt libraries are based on the default configurations, see *Using a prebuilt library*, page 44. There is also a ready-made library project template that you can use if you want to rebuild the runtime library.

DEBUG SUPPORT IN THE RUNTIME LIBRARY

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

The following table describes the different levels of debugging support:

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

Table 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 will be replaced by functions that communicate with the IAR C-SPY Debugger. For further information, see *C-SPY Debugger runtime interface*, page 64.

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

Using a prebuilt library

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

- Instruction set architecture
- Code model
- Data model
- Division instructions
- Library configuration—Normal or Full.

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

`<`*`type`*`><`*`target`*`><`*`isa`*`><`*`code_model`*`><`*`data_model`*`><`*`div`*`><`*`lib_config`*`>.r68` where

- <*type*> is `dl` for the IAR DLIB runtime environment
- <*target*> is `cf` for ColdFire
- <*isa*> is one of `a`, `ap`, `b`, or `c`, for isa_a, isa_a+, isa_b, and isa_c, respectively
- <*code_model*> is one of `n` or `f` for Near and Far code model, respectively
- <*data_model*> is one of `n` or `f` for Near relative and Far data model, respectively
- <*div*> is `d` when no `DIV`/`REM` instructions are used and empty when they are not used
- <*lib_config*> is one of `n` or `f` for normal and full, respectively.

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

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

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

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

  ```
  dlcfcffn.r68
  ```
- Specify the include paths for the compiler and assembler:

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

  ```
  --dlib_config C:\...\dlcfcffn.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 `cf\lib`.

CUSTOMIZING A PREBUILT LIBRARY WITHOUT REBUILDING

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

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

The following items can be customized:

Items that can be customized	Described in
Formatters for printf and scanf	*Choosing formatters for printf and scanf*, page 46
Startup and termination code	*System startup and termination*, page 51
Low-level input and output	*Standard streams for input and output*, page 55
File input and output	*File input and output*, page 58
Low-level environment functions	*Environment interaction*, page 61
Low-level signal functions	*Signal and raise*, page 62
Low-level time functions	*Time*, page 63
Size of heaps, stacks, and segments	*Placing code and data*, page 27

Table 12: Customizable items

For a description about how to override library modules, see *Overriding library modules*, page 48.

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 57.

CHOOSING PRINTF FORMATTER

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

The following table summarizes the capabilities of the different formatters:

Formatting capabilities	_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 13: 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 57.

Specifying the print formatter in the IAR Embedded Workbench IDE

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

Specifying printf formatter from the command line

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

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

CHOOSING SCANF FORMATTER

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

The following table summarizes the capabilities of the different formatters:

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

Table 14: 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 57.

Specifying scanf formatter in the IAR Embedded Workbench IDE

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

Specifying scanf formatter from the command line

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

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

Overriding library modules

The library contains modules which you probably need to override with your own customized modules, for example functions for character-based I/O and `cstartup`. This can be done without rebuilding the entire library. This section describes the procedure for including your version of the module in the application project build process. The library files that you can override with your own versions are located in the `cf\src\lib` directory.

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

Overriding library modules using the IAR Embedded Workbench IDE

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

1 Copy the appropriate *`library_module`*`.c` file to your project directory.

2 Make the required additions to the file (or create your own routine, using the default file as a model), and make sure that it has the same *module name* as the original module. The easiest way to achieve this is to save the new file under the same name as the original file.

3 Add the customized file to your project.

4 Rebuild your project.

Overriding library modules from the command line

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

1 Copy the appropriate *`library_module`*`.c` to your project directory.

2 Make the required additions to the file (or create your own routine, using the default file as a model), and make sure that it has the same *module name* as the original module. The easiest way to achieve this is to save the new file under the same name as the original file.

3 Compile the modified file using the same options as for the rest of the project:

```
icccf library_module
```

This creates a replacement object module file named *`library_module`*`.r68`.

Note: Make sure to use a library that matches the settings of the rest of your application.

4 Add *`library_module`*`.r68` to the XLINK command line, either directly or by using an extended linker command file, for example:

```
xlink library_module dlcf1cfn.r68
```

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

Run XLINK to rebuild your application.

This will use your version of `library_module.r68`, 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 43, it is necessary to rebuild the library. In those cases you need to:

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

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

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

SETTING UP A LIBRARY PROJECT

The IAR Embedded Workbench IDE provides a library project template which can be used for customizing the runtime environment configuration. This library template has full library configuration, see Table 10, *Library configurations*, page 43.

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

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

MODIFYING THE LIBRARY FUNCTIONALITY

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

The library functionality is determined by a set of *configuration symbols*. The default values of these symbols are defined in the file `Dlib_defaults.h`. This read-only file describes the configuration possibilities. In addition, your library has its own library configuration file `dlcfCustom.h`, which sets up that specific library with full library configuration. For more information, see Table 12, *Customizable items*, page 46.

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

Modifying the library configuration file

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

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

USING A CUSTOMIZED LIBRARY

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

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

1 Choose **Project>Options** and click the **Library Configuration** tab in the **General Options** category.

2 Choose **Custom DLIB** from the **Library** drop-down menu.

3 In the **Library file** text box, locate your library file.

4 In the **Configuration file** text box, locate your library configuration file.

System startup and termination

This section describes the runtime environment actions performed during startup and termination of applications.The code for handling startup and termination is located in the source files `cstartup.s68`, `cexit.s68`, and `low_level_init.c` located in the `cf\src\lib` directory.

In addition, for device-specific initializations, the `cstart`*`device`*`.s68` file must be included to your project. The files are located in the `cf\src` directory. In the IAR

Embedded Workbench IDE, the appropriate file is automatically included if you create your project based on a template project.

SYSTEM STARTUP

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

For the hardware intialization, it looks like this:

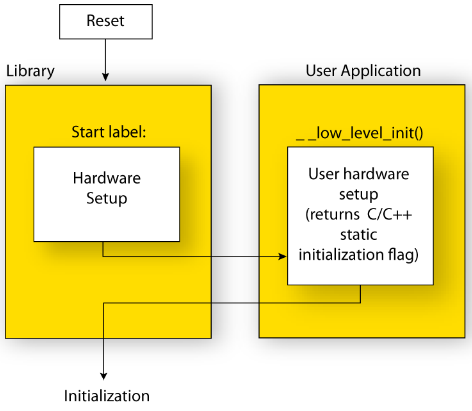


Figure 1: Target hardware initialization phase

- When the CPU is reset it will jump directly to the program entry label `__program_start` in the `cstart`*`device`*`.s68` file, if that file is available in your project. In that case, device-specific intitializations are performed, such as enabling flash and RAM areas, and setting up the vector base address and the SFR base addresses. Then, execution falls in to the `cstartup.s68` file.

  If the `cstart`*`device`*`.s68` file is not available in your project, the CPU will at reset jump directly to the program entry label `__program_start` in the `cstartup.s68` file.
- The stack pointer is initialized to the end of the `CSTACK` segment.
- The function `__low_level_init` is called if you have defined it, giving the application a chance to perform early initializations.

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

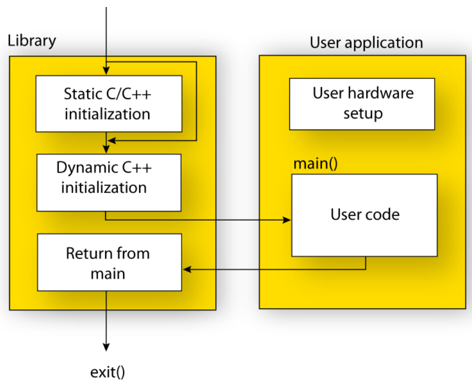


Figure 2: C/C++ initialization phase

- Static variables are initialized; this includes clearing zero-initialized memory and copying the ROM image of the RAM memory of the rest of the initialized variables depending on the return value of `__low_level_init`. For more details, see *Initialized data*, page 32
- Static C++ objects are constructed
- The `main` function is called, which starts the application.

SYSTEM TERMINATION

If your embedded application terminates, there are different ways it can terminate in a controlled way:

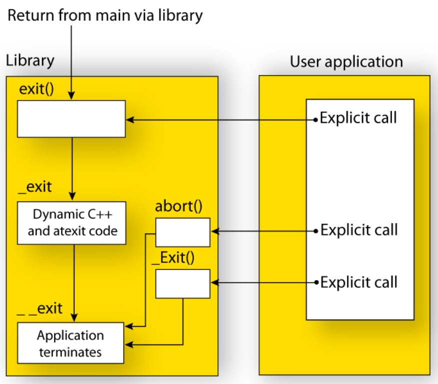


Figure 3: System termination phase

An application can terminate normally in two different ways:

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

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

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

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

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

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

C-SPY interface to system termination

If your project is linked with the XLINK options **With runtime control modules** or **With I/O emulation modules**, the normal `__exit` and `abort` functions are replaced with special ones. C-SPY will then recognize when those functions are called and can take appropriate actions to simulate program termination. For more information, see *C-SPY Debugger runtime interface*, page 64.

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.s68` before the data segments are initialized. Modifying the file `cstartup` directly should be avoided.

The code for handling system startup is located in the source files `cstartup.s68` and `low_level_init.c`, located in the `cf\src\lib` directory.

Note: Normally, there is no need for customizing either of the files `cmain.s68` or `cexit.s68`.

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 50.

Note: Regardless of whether you modify the routine `__low_level_init` or the file `cstartup.s68`, you do not have to rebuild the library.

__LOW_LEVEL_INIT

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

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

MODIFYING THE FILE CSTARTUP.S68

As noted earlier, you should not modify the file `cstartup.s68` if a customized version of `__low_level_init` is enough for your needs. However, if you do need to modify the file `cstartup.s68`, 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 48.

Standard streams for input and output

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

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

IMPLEMENTING LOW-LEVEL CHARACTER INPUT AND OUTPUT

To implement low-level functionality of the `stdin` and `stdout` streams, you must write the functions `__read` and `__write`, respectively. You can find template source code for these functions in the `cf\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 50. Note that customizing the low-level routines for input and output does not require you to rebuild the library.

Note: If you write your own variants of `__read` or `__write`, special considerations for the C-SPY runtime interface are needed, see *C-SPY Debugger runtime interface*, page 64.

Example of using __write and __read

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

```
__no_init volatile unsigned char LCD_IO @ address;

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

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

```
__no_init volatile unsigned char KB_IO @ address;

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

For information about the `@` operator, see *Controlling data and function placement in memory*, page 94.

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 46.

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

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

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

Printf configuration symbols	**Includes support for**
`_DLIB_PRINTF_MULTIBYTE`	Multibyte characters
`_DLIB_PRINTF_LONG_LONG`	Long long (`ll` qualifier)
`_DLIB_PRINTF_SPECIFIER_FLOAT`	Floating-point numbers
`_DLIB_PRINTF_SPECIFIER_A`	Hexadecimal floats
`_DLIB_PRINTF_SPECIFIER_N`	Output count (`%n`)
`_DLIB_PRINTF_QUALIFIERS`	Qualifiers `h`, `l`, `L`, `v`, `t`, and `z`
`_DLIB_PRINTF_FLAGS`	Flags `-`, `+`, `#`, and `0`
`_DLIB_PRINTF_WIDTH_AND_PRECISION`	Width and precision

Table 15: Descriptions of printf configuration symbols

Printf configuration symbols	Includes support for
`_DLIB_PRINTF_CHAR_BY_CHAR`	Output char by char or buffered

Table 15: Descriptions of printf configuration symbols (Continued)

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

Scanf configuration symbols	Includes support for
`_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 16: Descriptions of scanf configuration symbols

CUSTOMIZING FORMATTING CAPABILITIES

To customize the formatting capabilities, you need to set up a library project, see *Building and using a customized library*, page 50. Define the configuration symbols according to your application requirements.

File input and output

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

Note that file I/O capability in the library is only supported by libraries with full library configuration, see *Library configurations*, page 43. In other words, file I/O is supported when the configuration symbol `__DLIB_FILE_DESCRIPTOR` is enabled. If not enabled, functions taking a *`FILE *`* argument cannot be used.

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

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

Table 17: Low-level I/O files

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

Note: If you link your library with I/O debugging support, C-SPY variants of the low-level I/O functions will be linked for interaction with C-SPY. For more information, see *Debug support in the runtime library*, page 43.

Locale

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

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

The DLIB library can be used in two main modes:

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

LOCALE SUPPORT IN PREBUILT LIBRARIES

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

- All prebuilt libraries support the C locale only

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

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

CUSTOMIZING THE LOCALE SUPPORT

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

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

Locale configuration symbols

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

```
#define _DLIB_FULL_LOCALE_SUPPORT 1
#define _LOCALE_USE_C        /* C locale */
#define _LOCALE_USE_EN_US    /* US english */
#define _LOCALE_USE_EN_GB    /* UK english */
#define _LOCALE_USE_SV_SE    /* Swedish swedish */
```

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 50.

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

Building a library without support for locale interface

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

Building a library with support for locale interface

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

CHANGING LOCALES AT RUNTIME

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

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

`lang_REGION`

or

`lang_REGION.encoding`

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

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

Example

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

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

Environment interaction

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

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

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

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

```
key=value\0
```

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

For example:

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

If you need a more sophisticated environment variable handling, you should implement your own `getenv`, and possibly `putenv` function. This does not require that you rebuild the library. You can find source templates in the files `getenv.c` and `environ.c` in the `cf\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 48.

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

If you decide to rebuild the library, you can find source templates in the library project template. For further information, see *Building and using a customized library*, page 50.

Note: If you link your application with support for I/O debugging, the functions `getenv` and `system` will be replaced by C-SPY variants. For further information, see *Debug support in the runtime library*, page 43.

Signal and raise

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

This does not require that you rebuild the library. You can find source templates in the files `signal.c` and `raise.c` in the `cf\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 48.

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 50.

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 `cf\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 48.

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 50.

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

Note: If you link your application with support for I/O debugging, the functions `clock` and `time` will be replaced by C-SPY variants that return the host clock and time respectively. For further information, see *C-SPY Debugger runtime interface*, page 64.

Strtod

The function `strtod` does not accept hexadecimal floating-point strings in libraries with the normal library configuration. To make a library do so, you need to rebuild the library, see *Building and using a customized library*, page 50. Enable the configuration symbol `_DLIB_STRTOD_HEX_FLOAT` in the library configuration file.

Assert

If you have linked your application with support for runtime debugging, C-SPY will be notified about failed asserts. If this is not the behavior you require, you must add the source file `xreportassert.c` to your application project. Alternatively, you can rebuild the library. The `__ReportAssert` function generates the assert notification. You can find template code in the `cf\src\lib` directory. For further information, see *Building and using a customized library*, page 50. To turn off assertions, you must define the symbol `NDEBUG`.

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

C-SPY Debugger runtime interface

To include support for runtime and I/O debugging, you must link your application with the XLINK options **With runtime control modules** or **With I/O emulation modules**, see *Debug support in the runtime library*, page 43.

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

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

Table 18: Functions with special meanings when linked with debug info

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

LOW-LEVEL DEBUGGER RUNTIME INTERFACE

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

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

The mechanism used for implementing this feature works as follows:

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

THE DEBUGGER TERMINAL I/O WINDOW

To make the Terminal I/O window available, the application must be linked with support for I/O debugging, see *Debug support in the runtime library*, page 43. This means that when the functions `__read` or `__write` are called to perform I/O operations on the streams `stdin`, `stdout`, or `stderr`, data will be sent to or read from the C-SPY Terminal I/O window.

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

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

Speeding up terminal output

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

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

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

```
-e__write_buffered=__write
```

Checking module consistency

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

When developing an application, it is important to ensure that incompatible modules are not used together. For example, if you have a UART that can run in two modes, you can specify a runtime model attribute, for example `uart`. For each mode, specify a value, for example `mode1` and `mode2`. You should declare this in each module that assumes that the UART is in a particular mode.

The tools provided by IAR Systems use a set of predefined runtime model attributes to automatically ensure module consistency.

RUNTIME MODEL ATTRIBUTES

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

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

Example

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

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

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

Table 19: Example of runtime model attributes

USING RUNTIME MODEL ATTRIBUTES

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

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

For detailed syntax information, see *rtmodel*, page 181.

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

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

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

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.

Assembler language interface

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

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

Finally, the chapter covers how functions are called in the different code models, 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 ColdFire IAR C/C++ Compiler provides several ways to mix C or C++ and assembler:

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

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

INTRINSIC FUNCTIONS

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

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

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

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

MIXING C AND ASSEMBLER MODULES

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

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

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

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

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

The first issue is discussed in the section *Calling assembler routines from C*, page 72. The following two are covered in the section *Calling convention*, page 75.

The section on memory access methods, page 82, 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 82.

The recommended method for mixing C or C++ and assembler modules is described in *Calling assembler routines from C*, page 72, and *Calling assembler routines from C++*, page 74, respectively.

INLINE ASSEMBLER

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

```
bool flag;

void foo()
{
  while (!flag)
  {
    asm("MOVE.B (0x1000).W, D0 \n");
    asm("MOVE.B D0, (flag).L");
  }
}
```

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

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

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

- The compiler's various optimizations will disregard any effects of the inline sequences, which will not be optimized at all
- In general, assembler directives will cause errors or have no meaning. Data definition directives will work as expected
- Alignment cannot be controlled; this means, for example, that `DC32` directives may be misaligned
- Auto variables cannot be accessed.

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

Calling assembler routines from C

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

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

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

  or

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

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

CREATING SKELETON CODE

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

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

```
extern int gInt;
extern double gDouble;
```

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

int main()
{
  int locInt = gInt;
  gInt = func(locInt, gDouble);
  return 0;
}
```

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

COMPILING THE CODE

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

Use the following options to compile the skeleton code:

```
icccf skeleton -lA .
```

The `-lA` option creates an assembler language output file including C or C++ source lines as assembler comments. The `.` (period) specifies that the assembler file should be named in the same way as the C or C++ module (`skeleton`), but with the filename extension `s68`. Also remember to specify the code model you are using as well as a low level of optimization and `-e` for enabling language extensions.

The result is the assembler source output file `skeleton.s68`.

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

`-lA`. Note that `CFI` information must be included in the source code to make the C-SPY Call Stack window work.

The output file

The output file contains the following important information:

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

The `CFI` directives describe the call frame information needed by the Call Stack window in the IAR C-SPY Debugger.

Calling assembler routines from C++

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

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

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

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

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

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

```
class X;

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

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

```
class X
{
public:
  inline void doit(int arg) { ::doit(this, arg); }
};
```

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

- Assembler list files resulting from compiling C++ files cannot, in general, be passed through the assembler.

It is not possible to refer to or define C++ functions that do not have C linkage in assembler.

Calling convention

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

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

This section describes the calling conventions used by the ColdFire IAR C/C++ Compiler. The following items are examined:

- 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.

FUNCTION DECLARATIONS

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

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

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

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

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.

The following is an example of a declaration of a function with C linkage:

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

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

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

   int f(int);

#ifdef __cplusplus
}
#endif
```

PRESERVED VERSUS SCRATCH REGISTERS

The general ColdFire 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.

Any of the registers `D0`, `D1`, `D2`, `A0`, and `A1` can be used as a scratch register by the function.

Preserved registers

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

The registers `D3` through to `D7`, as well as `A2` through to `A6` are preserved registers.

Special registers

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

- Structure types: `struct`, `union`, and classes. If the structure only has one member, the structure is passed in the same way as its member.
- The data type `long long` and `double` (64-bit floating-point numbers), unless an FPU is available
- Unnamed parameters to variable length functions; in other words, functions declared as `foo(`*`param1`*`, ...)`, for example `printf`.

All `char` and `short` declared objects are cast to `int` at function entrance.

Note: Interrupt functions cannot take any parameters.

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, the memory location where the structure is to be stored is passed in the register as a hidden parameter, depending on the size of the structure and its members, see *Registers used for returning values*, page 79.

Register parameters

The registers available for passing parameters are `D0`, `D1`, `D2`, `A0`, and `A1`.

Parameters	**Passed in registers**
`char` and `short`	`D0`–`D2`, extended to a 32-bit value
`long`	`D0`
pointers	`A0`

Table 20: Registers used for passing parameters

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 the available register or registers. Should there be no suitable register available, the parameter is passed on the stack.

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 (towards 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 that is divisible by four, etc.

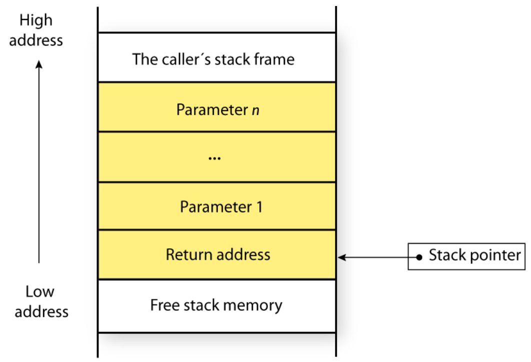


Figure 4: Stack image after the function call

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.

For structures that only has one member, the structure is returned in the same way as its member. Otherwise, it is returned via a pointer.

Registers used for returning values

The registers available for returning values are `D0`, `D0:D1`, and `A0`.

Return values	**Passed in registers**
`char` and `short`	`D0`, extended to a 32-bit value
`long` and `int`	`D0`
`long long` and `double`	`D0:D1`
pointers	`A0`

Table 21: Registers used for returning values

Values larger than 8 bytes are returned on the stack instead of in registers.

Stack layout

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

Return address handling

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

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 towards the end.

Example 1

Assume that we have the following function declaration:

```
int add1(int);
```

This function takes one parameter in the register `D0`, and the return value is passed back to its caller in the register `D0`.

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

```
ADDQ.L    #1,D0
RTS
```

Example 2

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

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

The calling function must reserve 4 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 `D0`. The return value is passed back to its caller in the register `D0`.

Example 3

The function below will return a `struct`.

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

It is the responsibility of the calling function to allocate a memory location for the return value and pass a pointer to it as a hidden first parameter. The pointer to the location where the return value should be stored is passed in `A0`. The parameter *x* is passed in `D0`. A pointer is returned in `A0`.

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

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

In this case, the return value is a scalar, so there is no hidden parameter. The parameter `x` is passed in `D0`, and the return value is returned in `A0`.

FUNCTION DIRECTIVES

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

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

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

Calling functions

In this section, we describe how functions are called in the different code models.

The normal function calling instruction is the jump-to-subroutine instruction:

```
JSR <ea>
```

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

In the Near code model, a direct call is simply:

```
JSR (label).W
```

In the Far code model, a direct call is simply:

```
JSR (label).L
```

Calls via a function pointer reach the whole 32-bit address space.

Memory access methods

This section describes the different memory types presented in the chapter *Data storage*.

For near memory, a direct memory access is simply:

```
MOVE.L D0,(var).W
```

For far memory, a direct memory access is simply:

```
MOVE.L D0,(var).L
```

Data pointers reach the whole 32-bit address space.

Call frame information

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

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

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

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

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

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

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

Resource	Description
A0–A6	Address registers
D0–D7	Data registers
FP0–FP7	Floating-point registers
SP	The stack pointer
?RET	The return address

Table 22: Call frame information resources defined in a names block

Example

The following is an example of an assembler routine that stores a permanent register, as well as the return register, to the stack:

```
        NAME test

        RSEG CSTACK:DATA:NOROOT(2)

        PUBLIC test
        FUNCTION test,0203H
        ARGFRAME CSTACK, 0, STACK
        LOCFRAME CSTACK, 4, STACK

        CFI Names cfiNames0
        CFI StackFrame CFA SP DATA
        CFI Resource A0:32, A1:32, A2:32, A3:32, A4:32, A5:32,
A6:32,
D0:32
        CFI Resource D1:32, D2:32, D3:32, D4:32, D5:32, D6:32,
D7:32
        CFI Resource SP:32
        CFI VirtualResource ?RET:32
        CFI EndNames cfiNames0

        CFI Common cfiCommon0 Using cfiNames0
        CFI CodeAlign 2
        CFI DataAlign 1
        CFI ReturnAddress ?RET CODE
        CFI CFA SP+4
        CFI A0 Undefined
        CFI A1 Undefined
        CFI A2 SameValue
        CFI A3 SameValue
        CFI A4 SameValue
        CFI A5 SameValue
        CFI A6 SameValue
        CFI D0 Undefined
        CFI D1 Undefined
        CFI D2 Undefined
        CFI D3 SameValue
        CFI D4 SameValue
        CFI D5 SameValue
        CFI D6 SameValue
        CFI D7 SameValue
        CFI ?RET Frame(CFA, -4)
        CFI EndCommon cfiCommon0

        EXTERN printf
        FUNCTION printf,0202H


        RSEG CODE:CODE:NOROOT(1)
test:
        CFI Block cfiBlock0 Using cfiCommon0
        CFI Function test
        MOVE.L    D0, -(A7)
        CFI CFA SP+8
        MOVEA.L   #`?<Constant "%d\\n">`, A0
        JSR       (printf).L
        ADDQ.L    #4, A7
        CFI CFA SP+4
        RTS
        CFI EndBlock cfiBlock0

        RSEG FAR_C:CONST:REORDER:NOROOT(0) `?<Constant "%d\\n">`:
        DC8 "%d\012"

        END
```

Using C++

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

Overview

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

STANDARD EMBEDDED C++

The following C++ features are supported:

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

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

Standard Embedded C++ lacks the following features of C++:

- Templates
- Multiple and virtual inheritance
- Exception handling

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

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

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

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

EXTENDED EMBEDDED C++

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

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

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

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

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

ENABLING C++ SUPPORT

In the ColdFire IAR C/C++ Compiler, the default language is C. To be able to compile files written in Embedded C++, you must use the `--ec++` compiler option. See *--ec++*, page 126.

To take advantage of *Extended* Embedded C++ features in your source code, you must use the `--eec++` compiler option. See *--eec++*, page 126.

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

Feature descriptions

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

CLASSES

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

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

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 149.

Example

```
class A {
  public:
    static __near __no_init int i; //Located in near memory
    static __near_func void f(); //Located in nearfunc memory
    __near_func void g();        //Located in nearfunc memory
    virtual __near_func void h();//Located in nearfunc memory
};
virtual void m() const volatile @ "SPECIAL" = 0; //m() placed in
                                                          SPECIAL
```

FUNCTIONS

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

Example

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

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

f(f1);                       // Always works
f(f2);                       // fpCpp is compatible with fpC
```

TEMPLATES

Extended EC++ supports templates according to the C++ standard, except for the support of the `export` keyword. The implementation uses a two-phase lookup which means that the keyword `typename` has to be inserted wherever needed. Furthermore, at each use of a template, the definitions of all possible templates must be visible. This means that the definitions of all templates have to be in include files or in the actual source file.

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 86.

STL and the IAR C-SPY® Debugger

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

VARIANTS OF CASTS

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

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

MUTABLE

The `mutable` attribute is supported in Extended EC++. A `mutable` symbol can be changed even though the whole class object is `const`.

NAMESPACE

The namespace feature is only supported in *Extended* EC++. This means that you can use namespaces to partition your code. Note, however, that the library itself is not placed in the `std` namespace.

THE STD NAMESPACE

The `std` namespace is not used in either standard EC++ or in Extended EC++. If you have code that refers to symbols in the `std` namespace, simply define `std` as nothing; for example:

```
#define std  // Nothing here
```

USING INTERRUPTS AND EC++ DESTRUCTORS

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

To avoid this, 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 have enabled IAR language extensions, the following C++ language extensions are available in the compiler:

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

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

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

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

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

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

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

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

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

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

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

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

- Default arguments may be specified for function parameters not only in the top-level function declaration, which is according to the standard, but also in `typedef` declarations, in pointer-to-function function declarations, and in pointer-to-member function declarations.

- In a function that contains a non-static local variable and a class that contains a non-evaluated expression (for example a `sizeof` expression), the expression may reference the non-static local variable. However, a warning is issued.

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.

- Try to avoid 64-bit data types, such as `double` and `long long`.
- Bitfields with sizes other than 1 bit should be avoided because they will result in inefficient code compared to bit operations.
- Using floating-point types on a microprocessor without a math co-processor is very inefficient, both in terms of code size and execution speed.
- Declaring a pointer to `const` data tells the calling function that the data pointed to will not change, which opens for better optimizations.

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

FLOATING-POINT TYPES

Using floating-point types on a microprocessor without a math coprocessor is very inefficient, both in terms of code size and execution speed. The ColdFire IAR C/C++ Compiler supports two floating-point formats—32 and 64 bits. The 32-bit floating-point type `float` is more efficient in terms of code size and execution speed. However, the 64-bit format `double` supports higher precision and larger numbers.

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

Note that a floating-point constant in the source code is treated as being of the type `double`. This can cause innocent-looking expressions to be evaluated in double precision. In the example below `a` is converted from a `float` to a `double`, `1` is added and the result is converted back to a `float`:

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

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

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

ALIGNMENT OF ELEMENTS IN A STRUCTURE

The ColdFire microcontroller accesses data more efficiently if data in memory is aligned. Each element in a structure needs to be aligned according to its specified type requirements. This means that the compiler might need to insert *pad bytes* to keep the alignment correct.

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

- External demands, for example, network communication protocols are usually specified in terms of data types with no padding in between
- There is a need to save data memory.

For information about alignment requirements, see *Alignment*, page 141.

There are two ways to solve the problem:

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

For further details about the `#pragma pack` directive, see *pack*, page 179.

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 ColdFire IAR C/C++ Compiler they can be used in C if language extensions are enabled.

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

Use the `-e` compiler option to enable language extensions. See *-e*, page 126, for additional information.

Example

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

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

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

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

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

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

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

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

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

Controlling data and function placement in memory

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

- Code models

  Use the compiler option for code models to take advantage of the different addressing modes available for the microcontroller and thereby also place functions in different parts of memory. To read more about code models, see *Code models and memory attributes for function storage*, page 21.

- Memory attributes

  Use memory attributes to override the default addressing mode and placement of individual functions and data objects. To read more about memory attributes for data and functions, see *Using data memory attributes*, page 14, and *Using function memory attributes*, page 22, respectively.

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

  Use the `@` operator or the `#pragma location` directive to place individual global and static variables at absolute addresses. The variables must be declared `__no_init`. This is useful for individual data objects that must be located at a fixed address, for example variables with external requirements. 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 segments can, for example, be placed in specific areas of memory, or initialized or copied in controlled ways using the segment begin and end operators. This is also useful if you want an interface between separately linked units, for example an application project and a boot loader project. Use named segments when absolute control over the placement of individual variables is not needed, or not useful.

At compile time, data and functions are placed in different segments as described in *Data segments*, page 30, and *Code segments*, page 36, 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 28.

DATA PLACEMENT AT AN ABSOLUTE LOCATION

The `@` operator, alternatively the `#pragma location` directive, can be used for placing global and static variables at absolute addresses. The variables must be declared using one of the following combinations of keywords:

- `__no_init`
- `__no_init` and `const` (without initializers)

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

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

Examples

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

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

In the following example, there is a const declared object, which is not initialized. The object is placed in ROM. This is useful for configuration parameters, which are accessible from an external interface.

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

The actual value must be set by other means. The typical use is for configurations where the values are loaded to ROM separately, or for special function registers that are read-only.

The following examples show incorrect usage:

```
int delta @ 0xFF2006;              /* Error, not __no_init */
```

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 there are more than one variable 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:

```
extern volatile const __no_init int x @ 0x100;  /* the extern
                                                  /* keyword makes x public */
```

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

DATA AND FUNCTION PLACEMENT IN SEGMENTS

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

- The `@` operator, alternatively the `#pragma location` directive, can be used for placing individual variables or individual functions in named segments. The named segment can either be a predefined segment, or a user-defined segment.

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.

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.

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

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

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

```
__far __no_init int alpha @ "NOINIT";/* Placed in far */
```

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, for example:

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

Controlling compiler optimizations

The compiler performs many transformations on your application in order to generate the best possible code. Examples of such transformations are storing values in registers instead of memory, removing superfluous code, reordering computations in a more efficient order, and replacing arithmetic operations by cheaper operations.

The linker should also be considered an integral part of the compilation system, because there are some optimizations that are performed by the linker. For instance, all unused functions and variables are removed and not included in the final output.

SCOPE FOR PERFORMED OPTIMIZATIONS

You can decide whether optimizations should be performed on your whole application or on individual files. By default, the same types of optimizations are used for an entire project, but you should consider using different optimization settings for individual files. For example, put code that must execute very quickly into a separate file and compile it for minimal execution time, and the rest of the code for minimal code size. This will give a small program, which is still fast enough where it matters.

In addition, you can exclude individual functions from the performed optimizations. The `#pragma optimize` directive allows you to either lower the optimization level, or specify another type of optimization to be performed. Refer to *optimize*, page 295, for information about the pragma directive.

OPTIMIZATION LEVELS

The ColdFire IAR C/C++ Compiler supports different levels of optimizations. The following table lists the optimizations that are performed on each level:

Optimization level	Description
None (Best debug support)	Variables live through their entire scope Redundant label elimination Redundant branch elimination
Low	Same as above but variables only live for as long as they are needed, not necessarily through their entire scope Dead code elimination Live-dead analysis and optimization

Table 23: Compiler optimization levels

Optimization level	Description
Medium	Same as above Code hoisting Register content analysis and optimization Common subexpression elimination
High (Balanced)	Same as above Peephole optimization Instruction scheduling (when optimizing for speed) Cross jumping Loop unrolling Function inlining Code motion Type-based alias analysis

Table 23: Compiler optimization levels (Continued)

Note: Some of the performed optimizations can be individually enabled or disabled. For more information about these, see *Fine-tuning enabled transformations*, page 99.

A high level of optimization might result in increased compile time, and will most likely also make debugging more difficult, because it will be less clear how the generated code relates to the source code. For example, at the low, medium, and high optimization levels, variables do not live through their entire scope, which means processor registers used for storing variables can be reused immediately after they were last used. Due to this, the C-SPY Watch window might not be able to display the value of the variable throughout its scope. At any time, if you experience difficulties when debugging your code, try lowering the optimization level.

SPEED VERSUS SIZE

At the high optimization level, the compiler balances between size and speed optimizations. However, it is possible to fine-tune the optimizations explicitly for either size or speed. They only differ in what thresholds that are used; speed will trade size for speed, whereas size will trade speed for size. Note that one optimization sometimes enables other optimizations to be performed, and an application may in some cases become smaller even when optimizing for speed rather than size.

FINE-TUNING ENABLED TRANSFORMATIONS

At each optimization level you can disable some of the transformations individually. To disable a transformation, use either the appropriate option, for instance the command line option `--no_inline`, alternatively its equivalent in the IAR Embedded Workbench IDE **Function inlining**, or the `#pragma optimize` directive. The following transformations can be disabled:

- Common subexpression elimination

- Loop unrolling
- Function inlining
- Code motion
- Type-based alias analysis.

Common subexpression elimination

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

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

To read more about the command line option, see *--no_cse*, page 132.

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 134.

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 133.

Code motion

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

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

Type-based alias analysis

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

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

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

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

To read more about the command line option, see *--no_tbaa*, page 133.

Example

```
short f(short * p1, long * p2)
{
  *p2 = 0;
  *p1 = 1;
  return *p2;
}
```

With type-based alias analysis, it is assumed that a write access to the `short` pointed to by `p1` cannot affect the `long` value that `p2` points to. Thus, it is known at compile time that this function returns 0. However, in non-standard-conforming C or C++ code these pointers could overlap each other by being part of the same union. By using explicit casts, you can also force pointers of different pointer types to point to the same memory location.

Writing efficient code

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

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

- Local variables—auto variables and parameters—are preferred over static or global variables. The reason is that the optimizer must assume, for example, that called functions may modify non-local variables. When the life spans for local variables end, the previously occupied memory can then be reused. Globally declared variables will occupy data memory during the whole program execution.
- Avoid taking the address of local variables using the `&` operator. There are two main reasons why this is inefficient. First, the variable must be placed in memory, and thus cannot be placed in a processor register. This results in larger and slower code. Second, the optimizer can no longer assume that the local variable is unaffected over function calls.
- Module-local variables—variables that are declared static—are preferred over global variables. Also avoid taking the address of frequently accessed static variables.
- The compiler is capable of inlining functions. This means that instead of calling a function, the compiler inserts the content of the function at the location where the function was called. The result is a faster, but often larger, application. Also, inlining may enable further optimizations. The compiler often inlines small functions declared static. The use of the `#pragma inline` directive and the C++ keyword `inline` gives you fine-grained control, and it is the preferred method compared to the traditional way of using preprocessor macros. This feature can be disabled using the `--no_inline` command line option; see *--no_inline*, page 133.
- 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 69.

SAVING STACK SPACE AND RAM MEMORY

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

- If stack space is limited, avoid long call chains and recursive functions.
- Avoid using large non-scalar types, such as structures, as parameters or return type; in order to save stack space, you should instead pass them as pointers or, in C++, as references.

FUNCTION PROTOTYPES

It is possible to declare and define functions using one of two different styles:

- Prototyped
- Kernighan & Ritchie C (K&R C)

Both styles are included in the C standard; however, it is recommended to use the prototyped style, since it makes it easier for the compiler to find problems in the code. In addition, using the prototyped style will make it possible to generate more efficient code, since type promotion (implicit casting) is not needed. The K&R style is only supported for compatibility reasons.

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

Prototyped style

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

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

Kernighan & Ritchie style

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

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

INTEGER TYPES AND BIT NEGATION

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

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

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

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

PROTECTING SIMULTANEOUSLY ACCESSED VARIABLES

Variables that are accessed asynchronously, for example by interrupt routines or by code executing in separate threads, must be properly marked and have adequate protection. The only exception to this is a variable that is always *read-only*.

To mark a variable properly, use the `volatile` keyword. This informs the compiler, among other things, that the variable can be changed from other threads. The compiler will then avoid optimizing on the variable (for example, keeping track of the variable in registers), will not delay writes to it, and be careful accessing the variable only the number of times given in the source code. To read more about the `volatile` type qualifier, see *Declaring objects volatile*, page 149.

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

ACCESSING SPECIAL FUNCTION REGISTERS

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

See the appropriate header file for further details. You can also use the header files as templates when you create new header files for other ColdFire devices.

NON-INITIALIZED VARIABLES

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

The compiler supports the declaration of variables that will not be initialized, using the `__no_init` type modifier. They can be specified either as a keyword or using the `#pragma object_attribute` directive. The compiler places such variables in separate segment, according to the specified memory attribute. 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 168. Note that to use this keyword, language extensions must be enabled; see *-e*, page 126. For information about the `#pragma object_attribute`, see page 177.

Part 2. Reference information

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

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

COMPILER INVOCATION SYNTAX

The invocation syntax for the compiler is:

```
icccf [options] [sourcefile] [options]
```

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

```
icccf prog --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 `icccf` command, either before or after the source filename; see *Invocation syntax*, page 109.

- Via environment variables

  The compiler automatically appends the value of the environment variables to every command line; see *Environment variables*, page 110.
- Via a text file by using the `-f` option; see *-f*, page 127.

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

ENVIRONMENT VARIABLES

The following environment variables can be used with the ColdFire IAR C/C++ Compiler:

Environment variable	Description
C_INCLUDE	Specifies directories to search for include files; for example: `C_INCLUDE=c:\program files\iar systems\embedded workbench 4.`*`n`*`\cf\inc;c:\headers`
QCCCF	Specifies command line options; for example: `QCCCF=-lA asm.lst`

Table 24: Compiler environment variables

Include file search procedure

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

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

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

  it searches the following directories for the file to include:

  1. The directories specified with the `-I` option, in the order that they were specified, see *-I*, page 128.
  2. The directories specified using the `C_INCLUDE` environment variable, if any, see *Environment variables*, page 110.
- If the compiler encounters the name of an `#include` file in double quotes, for example:

  ```
  #include "vars.h"
  ```

  it searches the directory of the source file in which the `#include` statement occurs, and then performs the same sequence as for angle-bracketed filenames.

If there are nested `#include` files, the compiler starts searching the directory of the file that was last included, iterating upwards for each included file, searching the source file directory last. For example:

```
src.c in directory dir\src
   #include "src.h"
   ...
src.h in directory dir\include
   #include "config.h"
   ...
```

When `dir\exe` is the current directory, use the following command for compilation:

```
icccf ..\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 `r68`.
- Optional list files

  Different types of list files can be specified using the compiler option `-l`, see *-l*, page 129. By default, these files will have the filename extension `lst`.
- Optional preprocessor output files

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

- Diagnostic messages

  Diagnostic messages are directed to `stderr` and displayed on the screen, as well as printed in an optional list file. To read more about diagnostic messages, see *Diagnostics*, page 112.
- 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 112.
- Size information

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

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

Error return codes

The ColdFire IAR C/C++ Compiler returns status information to the operating system that can be tested in a batch file.

The following command line error codes are supported:

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

Table 25: Error return codes

Diagnostics

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

MESSAGE FORMAT

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

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

with the following elements:

`filename`	The name of the source file in which the issue was encountered
`linenumber`	The line number at which the compiler detected the issue
`level`	The level of seriousness of the issue
`tag`	A unique tag that identifies the diagnostic message
`message`	An explanation, possibly several lines long

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

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

SEVERITY LEVELS

The diagnostic messages are divided into different levels of severity:

Remark

A diagnostic message that is produced when the compiler finds a source code construction that can possibly lead to erroneous behavior in the generated code. Remarks are by default not issued, but can be enabled, see *--remarks*, page 139.

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 135.

Error

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

SETTING THE SEVERITY LEVEL

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

See *Compiler options summary*, page 118, for a description of the compiler options that are available for setting severity levels.

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

INTERNAL ERROR

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

```
Internal error: message
```

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

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

Compiler options

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

Options syntax

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

Refer to the *IAR Embedded Workbench® IDE User Guide* for information about the compiler options available in the IAR Embedded Workbench IDE and how to set them.

TYPES OF OPTIONS

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

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

For information about the different methods for passing options, see *Passing options*, page 109.

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:

```
--misrac=n
```

Rules for mandatory parameters

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

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

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

```
--diagnostics_tables=filename
```

or

```
--diagnostics_tables filename
```

Rules for options with both optional and mandatory parameters

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

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

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

```
-lA filename
```

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

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

Rules for specifying a filename or directory as parameters

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

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

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

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

  ```
  icccf prog -l ..\listings\
  ```

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

  ```
  icccf prog -l .
  ```

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

  ```
  icccf prog -l -
  ```

Additional rules

In addition, the following rules apply:

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

  ```
  icccf prog -l ---r
  ```

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

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

  Alternatively, the option may be repeated for each argument, for example:

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

Compiler options summary

The following table summarizes the compiler command line options:

Command line option	Description
`--char_is_signed`	Treats `char` as signed
`--code_model`	Specifies the code model
`--core`	Specifies a 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
`--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
`--isa`	Selects instruction set for architecture version
`-l`	Creates a list file
`--library_module`	Creates a library module
`--mac`	Enables support for multiply-accumulate (MAC)
`--misrac`	Enables MISRA C-specific error messages
`--misrac_verbose`	Enables verbose logging of MISRA C checking
`--module_name`	Sets the object module name

Table 26: Compiler options summary

Command line option	Description
`--no_code_motion`	Disables code motion optimization
`--no_cse`	Disables common subexpression elimination
`--no_div`	Disables support for `DIVS`/`DIVU` and `REMS`/`REMU` instructions
`--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
`--preinclude`	Includes an include file before reading the source file
`--preprocess`	Generates preprocessor output
`--public_equ`	Defines a global named assembler label
`-r`	Generates debug information
`--remarks`	Enables remarks
`--require_prototypes`	Verifies that functions are declared before they are defined
`--silent`	Sets silent operation
`--strict_ansi`	Checks for strict compliance with ISO/ANSI C
`--warnings_affect_exit_code`	Warnings affects exit code
`--warnings_are_errors`	Warnings are treated as errors

Table 26: 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 IAR Embedded Workbench IDE does not perform an instant check for consistency problems like conflicting options, duplication of options, or use of irrelevant options.

--char_is_signed

Syntax `--char_is_signed`

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

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

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

--code_model

Syntax `--code_model={n|near|f|far}`

Parameters

`n|near`	Supports the address range `0xFFFF8000-0x00007FFF`
`f|far` (default)	Supports the entire address range `0x0000000-0xFFFFFFFF`

Description Use this option to select the code model for which the code is to be generated. If you do not choose a code model option, the compiler uses the default code model. Note that all modules of your application must use the same code model.

See also *Code models and memory attributes for function storage*, page 21.

Project>Options>General Options>Target>Code model

--core

Syntax `--core={v1|v2|v3|v4}`

Description

Use this option to select the processor core for which the code is to be generated, V1, V2, V3, or V4. If you do not use the option to specify a core, the compiler uses the V2 core as default.

The compiler supports the different ColdFire microcontroller cores and devices based on these cores. The object code that the compiler generates for the different cores is binary compatible if the same instruction set architecture is used.

Note: The `v3` and `v4` parameters are available for future compatibility.

Project>Options>General Options>Target>Core

-D

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

Parameters

`symbol`	The name of the preprocessor symbol
`value`	The value of the preprocessor symbol

Description

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

The option `-D` has the same effect as a `#define` statement at the top of the source file:

`-D`*`symbol`*

is equivalent to:

`#define` *`symbol`* `1`

In order to get the equivalence of:

`#define FOO`

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

`-DFOO=`

Project>Options>C/C++ Compiler>Preprocessor>Defined symbols

--data_model

Syntax `--data_model={near_rel|far}`

Parameters

`near_rel`	Specifies the Near relative data model, which means objects will be located in a position-independent 64-Kbyte data block that can be placed anywhere in memory.
`far` (default)	Specifies the far data model, which means objects can be located anywhere in memory.

Description Use this option to select the data model for which the code is to be generated. If you do not choose 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 11.

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 116.

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.r68: c:\iar\product\include\stdio.h
foo.r68: 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:

```
%.r68 : %.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 IAR Embedded Workbench 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 116.

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 IAR Embedded Workbench IDE.

--dlib_config

Syntax

`--dlib_config` *`filename`*

Parameters

For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 116.

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 `cf\lib`. For examples and a list of prebuilt runtime libraries, see *Using a prebuilt library*, page 44.

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 50.

To set related options, choose:

Project>Options>General Options>Library Configuration

-e

Syntax `-e`

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

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

See also The chapter *Compiler extensions.*

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

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

--ec++

Syntax `--ec++`

Description In the ColdFire IAR C/C++ Compiler, the default language is C. If you use Embedded C++, you must use this option to set the language the compiler uses to Embedded C++.

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

--eec++

Syntax `--eec++`

Description In the ColdFire IAR C/C++ Compiler, the default language is C. If you take advantage of Extended Embedded C++ features like namespaces or the standard template library in your source code, you must use this option to set the language the compiler uses to Extended Embedded C++.

See also *Extended Embedded C++*, page 86.

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 IAR Embedded Workbench IDE.

-f

Syntax `-f filename`

Parameters For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 116.

Descriptions Use this option to make the compilerread 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 IAR Embedded Workbench IDE.

-I

Syntax

`-I` *`path`*

Parameters

`path`	The search path for `#include` files

Description

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

See also

Include file search procedure, page 110.

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

--isa

Syntax

```
--isa={isa_a|isa_a+|isa_b|isa_c}
```

Description

Use this option to select the instruction set architecture version to be used. If you do not use the option to specify what instruction set to use, the compiler uses the `isa_a` instruction set as default.

See also

The device documentation delivered from the chip manufacturer.

Project>Options>General Options>Target>ISA variant

-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 116.

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

--mac

Syntax `--mac={mac|emac|emac_b}`

Description Use this option to enable support for the multiply-accumulate instructions. If you do not use the option to specify what instruction set to use, the compiler does not support MAC instructions.

Note: This option does not currently have any effect.

See also The device documentation delivered from the chip manufacturer.

Project>Options>General Options>Target>MAC variant

--misrac

Syntax `--misrac[={n,o-p,…|all|required}]`

Parameters

`--misrac=n`	Enables checking for the MISRA C rule with number *n*
`--misrac=o,n`	Enables checking for the MISRA C rules with numbers *o* and *n*
`--misrac=o-p`	Enables checking for all MISRA C rules with numbers from *o* to *p*
`--misrac=m,n,o-p`	Enables checking for MISRA C rules with numbers *m*, *n*, and from *o* to *p*
`--misrac=all`	Enables checking for all MISRA C rules
`--misrac=required`	Enables checking for all MISRA C rules categorized as required

Description Use this option to enable the compiler to check for deviations from the rules described in the MISRA *Guidelines for the Use of the C Language in Vehicle Based Software* (1998). By using one or more arguments with the option, you can restrict the checking to a specific subset of the MISRA C rules. If the compiler is unable to check for a rule, specifying the option for that rule has no effect. For instance, MISRA C rule 15 is a documentation issue, and the rule is not checked by the compiler. As a consequence, specifying `--misrac=15` has no effect.

To set related options, choose:

Project>Options>General Options>MISRA C or **Project>Options>C/C++ Compiler>MISRA C**

--misrac_verbose

Syntax

```
--misrac_verbose
```

Description

Use this option to generate a MISRA C log during compilation. This is a list of the rules that are enabled—but not necessarily checked—and a list of rules that are actually checked.

If this option is enabled, the compiler display a text at sign-on that shows both enabled and checked MISRA C rules.

Project>Options>General Options>MISRA C>Log MISRA C Settings

--module_name

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 optimization levels Medium and High, normally reduce code size and execution time. However, the resulting code may be difficult to debug.

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

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Code motion

--no_cse

Syntax	`--no_cse`
Description	Use this option to disable common subexpression elimination. At optimization levels Medium and High, the compiler avoids calculating the same expression more than once. This optimization normally reduces both code size and execution time. However, the resulting code may be difficult to debug.

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

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

--no_div

Syntax	`--no_div`
Description	Use this option to disable support for the `DIVS`/`DIVU` and `REMS`/`REMU` instructions.
See also	*Division instructions*, page 6 and the device documentation delivered from the chip manufacturer.

Project>Options>General Options>Target>No DIV/REM instruction

--no_inline

Syntax `--no_inline`

Description Use this option to disable function inlining. Function inlining means that a simple function, whose definition is known at compile time, is integrated into the body of its caller to eliminate the overhead of the call.

This optimization, which is performed at optimization level High, normally reduces execution time and increases code size. The resulting code may also be difficult to debug.

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

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

Project>Options>C/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 190.

This option is not available in the IAR Embedded Workbench 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 101.

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 the following:

```
Error[Pe144]: a value of type "char *" cannot be used to
initialize an entity of type "MyPtr"
```

If the `--no_typedefs_in_diagnostics` option is used, the error message will be like this:

```
Error[Pe144]: a value of type "char *" cannot be used to
initialize an entity of type "int (*)(char const *)"
```

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

--no_unroll

Syntax

```
--no_unroll
```

Description

Use this option to disable loop unrolling. The code body of a small loop, whose number of iterations can be determined at compile time, is duplicated to reduce the loop overhead.

For small loops, the overhead required to perform the looping can be large compared with the work performed in the loop body.

The loop unrolling optimization duplicates the body several times, reducing the loop overhead. The unrolled body also opens up for other optimization opportunities.

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

The compiler heuristically decides which loops to unroll. Different heuristics are used when optimizing for speed and size.

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

Project>Options>C/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 IAR Embedded Workbench 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 IAR Embedded Workbench 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 98.

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 116.

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 `r68`. Use this option to explicitly specify a different output filename for the object code output.

This option is not available in the IAR Embedded Workbench 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 IAR Embedded Workbench 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 116.

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 `r68`. Use this option to explicitly specify a different output filename for the object code output.

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

--preinclude

Syntax

```
--preinclude includefile
```

Parameters

For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 116.

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 116.

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 may be used more than once on the command line.

This option is not available in the IAR Embedded Workbench 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 113.

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 IAR Embedded Workbench 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 158. 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

--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 IAR Embedded Workbench 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 174.

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 ColdFire IAR C/C++ Compiler.

See the chapter *Efficient coding for embedded applications* for information about which data types and pointers provide the most efficient code for your application.

Alignment

Every C data object has an alignment that controls how the object can be stored in memory. Should an object have an alignment of, for example, 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 lower the alignment requirements on the structure and its members, use `#pragma pack`.

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

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

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

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

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

ALIGNMENT ON THE COLDFIRE MICROCONTROLLER

The ColdFire microcontroller can access memory using 8-, 16-, and 32-bit operations. However, when a 16- or 32-bit access is performed, the data should be 2- and 4-byte aligned, respectively. Misaligned accesses work, but is slower. The ColdFire IAR C/C++ Compiler ensures this by assigning an alignment to every data type, ensuring that the ColdFire microcontroller will be able to read the data efficiently.

Basic data types

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

INTEGER TYPES

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

Data type	Size	Range	Alignment
bool	8 bits	0 to 1	1
char	8 bits	0 to 255	1
signed char	8 bits	-128 to 127	1
unsigned char	8 bits	0 to 255	1
signed short	16 bits	-32768 to 32767	2
unsigned short	16 bits	0 to 65535	2
signed int	32 bits	-2^{31} to $2^{31}-1$	4
unsigned int	32 bits	0 to $2^{32}-1$	4
signed long	32 bits	-2^{31} to $2^{31}-1$	4
unsigned long	32 bits	0 to $2^{32}-1$	4
signed long long	64 bits	-2^{63} to $2^{63}-1$	4
unsigned long long	64 bits	0 to $2^{64}-1$	4

Table 27: Integer types

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

Bool

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

The enum type

The compiler will use the smallest type required to hold `enum` constants, preferring `signed` rather than `unsigned`.

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

To make the compiler use a larger type than it would automatically use, define an `enum` constant with a large enough value. For example,

```
/* Disables usage of the char type for enum */
enum Cards{Spade1, Spade2,
           DontUseChar=257};
```

The char type

The `char` type is by default unsigned in the compiler, but the `--char_is_signed` compiler option allows you to make it signed. Note, however, that the library is compiled with the `char` type as unsigned.

The wchar_t type

The `wchar_t` data type is an integer type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locals.

The `wchar_t` data type is supported by default in the C++ language. To use the `wchar_t` type also in C source code, you must include the file `stddef.h` from the runtime library.

Bitfields

In ISO/ANSI C, `int`, `signed int`, and `unsigned int` can be used as the base type for integer bitfields. It is implementation defined whether the type specified by `int` is the same as `signed int` or `unsigned int`. In the ColdFire IAR C/C++ Compiler, bitfields declared as `int` are treated as `signed int`. Furthermore, 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 most significant to the least significant bit in the container type. A bitfield is assigned to the last available container of its base type which has enough unassigned bits to contain the entire bitfield.

This means that bitfield containers can overlap other structure members as long as the order of the fields in the structure is preserved, for example:

```
struct example
{
  char  a;
  short b : 10;
  int   c : 6;
};
```

Here the first declaration creates an unsigned character which is allocated to bits 24 through 31. The second declaration creates a signed short integer member of size 10 bits. This member is allocated to bits 15 through 6 as it will not fit in the remaining 8 bits of the first short integer container. The last bitfield member declared is placed in the bits 0 through 5. If seen as a 32-bit value, the structure looks like this in memory:

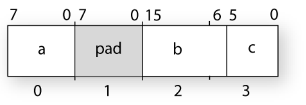


Figure 5: Layout of bitfield members

By using the directive `#pragma bitfields=reversed_disjoint_types`, the bitfield containers are forced to be disjoint, or in other words, to not overlap. The layout of the above example structure would then become:

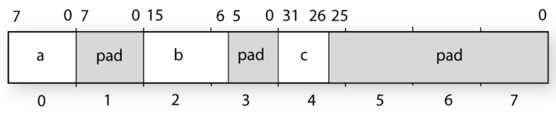


Figure 6: Layout of bitfield members forced to be disjoint

By using the directive `#pragma bitfields=disjoint_types`, the bitfield members are placed from the least significant bit to the most significant bit in non-overlapping storage containers.

FLOATING-POINT TYPES

In the ColdFire IAR C/C++ Compiler, floating-point values are represented in standard IEEE 754 format. The sizes for the different floating-point types are:

Type	Size
float	32 bits
double	64 bits
long double	64 bits

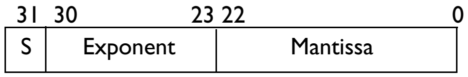
Table 28: Floating-point types

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. The alignment for the `float` type is 4, and for the `long double` type it is 4.

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`

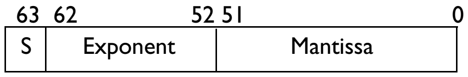
The range of the number is:

`±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`$^{(Exponent-1023)}$ `* 1.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

The following list describes the representation of special floating-point numbers:

- Zero is represented by zero mantissa and exponent. The sign bit signifies positive or negative zero.
- Infinity is represented by setting the exponent to the highest value and the mantissa to zero. The sign bit signifies positive or negative infinity.
- Not a number (`NaN`) is represented by setting the exponent to the highest positive value and the mantissa to a non-zero value. The value of the sign bit is ignored.

Pointer types

The ColdFire IAR C/C++ Compiler has two basic types of pointers: function pointers and data pointers.

FUNCTION POINTERS

The size of function pointers is always 32 bits, and they can address the entire memory.

DATA POINTERS

The size of data pointers is always 32 bits, and they can address the entire memory.

CASTING

Casts between pointers have the following characteristics:

- Casting a *value* of an integer type to a pointer of a smaller type is performed by truncation
- Casting a *value* of an integer type to a pointer of a larger type is performed by zero extension
- Casting a *pointer type* to a smaller integer type is performed by truncation
- Casting a `__near` or `__near_func` pointer to a larger integer type is performed by first sign extending it to a `__far` or `__far_func` pointer. Casting a `__far` or `__far_func` *pointer* 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

size_t

size_t is the unsigned integer type required to hold the maximum size of an object. In the ColdFire IAR C/C++ Compiler, the size of size_t is 32 bits.

ptrdiff_t

ptrdiff_t is the type of the signed integer required to hold the difference between two pointers to elements of the same array. In the ColdFire IAR C/C++ Compiler, the size of ptrdiff_t is 32 bits.

Note: Subtracting the start address of an object from the end address can yield a negative value, because the object can be larger than what the ptrdiff_t can represent. See this example:

```
char buff[60000];           /* Assuming ptrdiff_t is a 16-bit */
char *p1 = buff;            /* signed integer type. */
char *p2 = buff + 60000;
ptrdiff_t diff = p2 - p1;
```

intptr_t

intptr_t is a signed integer type large enough to contain a void *. In the ColdFire IAR C/C++ Compiler, the size of intptr_t is 32 bits.

uintptr_t

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

Structure types

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

ALIGNMENT

The struct and union types inherit the alignment requirements of their members. In addition, the size of a struct is adjusted to allow arrays of aligned structure objects.

GENERAL LAYOUT

Members of a struct (fields) are always allocated in the order given in the declaration. The members are placed in memory according to the given alignment (offsets).

Example

```
struct {
  short s;  /* stored in byte 0 and 1 */
  char c;   /* stored in byte 2 */
  long l;   /* stored in byte 4, 5, 6, and 7 */
  char c2;  /* stored in byte 8 */
} s;
```

The following diagram shows the layout in memory:

s.s	s.c	pad	s.l	s.c2	pad
2 bytes	1 byte	1 byte	4 bytes	1 byte	3 byte

The alignment of the structure is 4 bytes, and its size is 12 bytes.

PACKED STRUCTURE TYPES

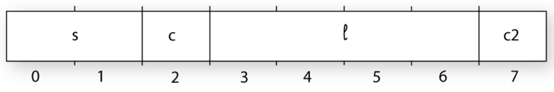
The `#pragma pack` directive is used for relaxing the alignment requirements of the members of a structure. This will change the way the layout of the structure is performed. The members will be placed in the same order as when declared, but there might be less pad space between members.

Note: This can result in significantly larger and slower code when accessing members of the structure.

Example

```
#pragma pack(1)
struct {
  short s;
  char c;
  long l;
  char c2;
} s;
```

will be placed:



For more information, see *Alignment of elements in a structure*, page 92.

Type qualifiers

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

DECLARING OBJECTS VOLATILE

There are three main reasons for declaring an object `volatile`:

- Shared access; the object is shared between several tasks in a multitasking environment
- Trigger access; as for a memory-mapped SFR where the fact that an access occurs has an effect
- Modified access; where the contents of the object can change in ways not known to the compiler.

Definition of access to volatile objects

The ISO/ANSI standard defines an abstract machine, which governs the behavior of accesses to `volatile` declared objects. In general and in accordance to the abstract machine, the compiler:

- Considers each read and write access to an object that has been declared `volatile` as an access
- The unit for the access is either the entire object or, for accesses to an element in a composite object—such as an array, struct, class, or union—the element. For example:

  ```
  char volatile a;
  a = 5;   /* A write access */
  a += 6;  /* First a read then a write access */
  ```
- An access to a bitfield is treated as an access to the underlaying type.

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

Rules for accesses

In the ColdFire IAR C/C++ Compiler, accesses to `volatile` declared objects are subject to the following rules:

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

The ColdFire IAR C/C++ Compiler adheres to these rules for the following combinations of memory types and data types:

Data type	Treated as
8-, 16-, and 32-bit values	All accesses are preserved in the original order
64-bit values	Preserved

Table 29: Volatile accesses

The following object types are treated in a special way:

Type of object	Treated as
Global register variables	Treated as non-triggering volatiles
1-bit bitfields	Preserved

Table 30: Type of volatile accesses treated in a special way

For all combinations of object types not listed, only the rule that states that all accesses are preserved applies.

DECLARING OBJECTS 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.

In C++, objects that require runtime initialization cannot be placed in ROM.

Data types in C++

In C++, all plain C data types are represented in the same way as described earlier in this chapter. However, if any Embedded C++ features are used for a type, no assumptions can be made concerning the data representation. This means, for example, that it is not legal to write assembler code that accesses class members.

Compiler extensions

This chapter gives a brief overview of the ColdFire IAR C/C++ Compiler extensions to the ISO/ANSI C standard. All extensions can also be used for the C++ programming language. More specifically the chapter describes the available C language extensions.

Compiler extensions overview

The compiler offers the standard features of ISO/ANSI C as well as a wide set of extensions, ranging from features specifically tailored for efficient programming in the embedded industry to the relaxation of some minor standards issues.

You can find the extensions available as:

- C/C++ language extensions

  For a summary of available language extensions, see *C language extensions*, page 152. For reference information about the extended keywords, see the chapter *Extended keywords*. For information about C++, the two levels of support for the language, and C++ language extensions; see the chapter *Using C++*.
- Pragma directives

  The `#pragma` directive is defined by the ISO/ANSI C standard and is a mechanism for using vendor-specific extensions in a controlled way to make sure that the source code is still portable.

  The compiler provides a set of predefined pragma directives, which can be used for controlling the behavior of the compiler, for example how it allocates memory, whether it allows extended keywords, and whether it outputs warning messages. Most pragma directives are preprocessed, which means that macros are substituted in a pragma directive. The pragma directives are always enabled in the compiler and many of them have an equivalent C/C++ language extensions. For a list of available pragma directives, see the chapter *Pragma directives*.
- Preprocessor extensions

  The preprocessor of the compiler adheres to the ISO/ANSI standard. In addition, the compiler also makes a number of preprocessor-related extensions available to you. For more information, see the chapter *The preprocessor*.
- Intrinsic functions

  The intrinsic functions provide direct access to low-level processor operations and can be very useful in, for example, time-critical routines. The intrinsic functions compile into inline code, either as a single instruction or as a short sequence of instructions. To read more about using intrinsic functions, see *Mixing C and assembler*, page 69. For a list of available functions, see the chapter *Intrinsic functions*.

- Library functions

  The IAR DLIB Library provides most of the important C and C++ library definitions that apply to embedded systems. In addition, the library also provides some extensions, partly taken from the C99 standard. For more information, see *IAR DLIB Library*, page 196.

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

ENABLING LANGUAGE EXTENSIONS

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

For information about how to enable and disable language extensions from the command line, see the compiler options *-e*, page 126, and *--strict_ansi*, page 140.

C language extensions

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

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

IMPORTANT LANGUAGE EXTENSIONS

The following language extensions available both in the C and the C++ programming languages are well suited for embedded systems programming:

- Memory attributes, type attributes, and object attributes

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

- Placement at an absolute address or in a named segment

  The `@` operator or the directive `#pragma location` can be used for placing global and static variables at absolute addresses, or placing a variable or function in a named segment. For more information about using these primitives, see *Controlling data and function placement in memory*, page 94, and *location*, page 176.

- Alignment

  Each data type has its own alignment, for more details, see *Alignment*, page 141. If you want to change the alignment, the `#pragma pack` and `#pragma data_alignment` directives are available. If you want to check the alignment of an object, use the `__ALIGNOF__()` operator.

  The `__ALIGNOF__` operator is used for accessing the alignment of an object. It takes one of two forms:

  - `__ALIGNOF__ (`*`type`*`)`
  - `__ALIGNOF__ (`*`expression`*`)`

  In the second form, the expression is not evaluated.

- Anonymous structs and unions

  C++ includes a feature 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 93.

- 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 `enum` may be used. The advantage is that the struct will sometimes be smaller. This matches G.5.8 in the appendix of the ISO standard, *ISO Portability Issues*. For more information, see *Bitfields*, page 143.

- Dedicated segment operators `__segment_begin` and `__segment_end`

  The syntax for these operators is:

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

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

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

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

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

See also *segment*, page 182, and *location*, page 176.

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 175.
- Mixing declarations and statements

  It is possible to mix declarations and statements within the same scope. This feature is part of the C99 standard and C++.
- Declaration in `for` loops

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

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

  This feature is part of the C99 standard and C++.
- The `bool` data type

  To use the `bool` type in C source code, you must include the file `stdbool.h`. This feature is part of the C99 standard and C++. (The `bool` data type is supported by default in C++.)
- C++ style comments

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

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

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

Inline assembler

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

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

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

The syntax is:

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

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

```
asm ("Label:      nop\n"
     "            bra.b 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 69.

Compound literals

To create compound literals you can use the following syntax:

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

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

Note:

- A compound literal can be modified unless it is declared `const`
- Compound literals are not supported in Embedded C++ and Extended EC++.
- This feature is part of the C99 standard.

Incomplete arrays at end of structs

The last element of a `struct` may be an incomplete array. This is useful because one chunk of memory can be allocated for the `struct` itself and for the array, regardless of the size of the array.

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

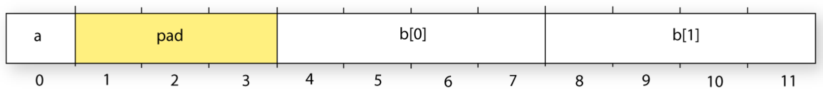
Example

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

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

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

The `struct` will inherit the alignment requirements from all elements, including the alignment of the incomplete array. The array itself will not be included in the size of the struct. However, the alignment requirements will ensure that the struct will end exactly at the beginning of the array; this is known as padding.

In the example, the alignment of `struct str` will be 4 and the size is also 4. (Assuming a processor where the alignment of `unsigned long` is 4.)

The memory layout of `struct str` is described in the following figure.



This feature is part of the C99 standard.

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

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

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

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

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

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

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

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

MINOR LANGUAGE EXTENSIONS

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

- Arrays of incomplete types

  An array may have an incomplete `struct`, `union`, or `enum` type as its element type. The types must be completed before the array is used (if it is), or by the end of the compilation unit (if it is not).

- Forward declaration of `enum` types

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

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

  A warning is issued if the semicolon at the end of a `struct` or `union` specifier is missing.

- Null and `void`

  In operations on pointers, a pointer to `void` is always implicitly converted to another type if necessary, and a null pointer constant is always implicitly converted to a null pointer of the right type if necessary. In ISO/ANSI C, some operators allow such things, while others do not allow them.

- Casting pointers to integers in static initializers

  In an initializer, a pointer constant value may be cast to an integral type if the integral type is large enough to contain it. For more information about casting pointers, see *Casting*, page 146.

- Taking the address of a register variable

  In ISO/ANSI C, it is illegal to take the address of a variable specified as a register variable. The compiler allows this, but a warning is issued.

- Duplicated size and sign specifiers

  Should the size or sign specifiers be duplicated (for example, `short short` or `unsigned unsigned`), an error is issued.

- `long float` means `double`

  The type `long float` is accepted as a synonym for `double`.

- Repeated `typedef` declarations

  Redeclarations of `typedef` that occur in the same scope are allowed, but a warning is issued.

- Mixing pointer types

  Assignment and pointer difference is allowed between pointers to types that are interchangeable but not identical; for example, `unsigned char *` and `char *`. This includes pointers to integral types of the same size. A warning is issued.

Assignment of a string constant to a pointer to any kind of character is allowed, and no warning will be issued.

- Non-top level `const`

  Assignment of pointers is allowed in cases where the destination type has added type qualifiers that are not at the top level (for example, `int **` to `int const **`). Comparing and taking the difference of such pointers is also allowed.

- Non-`lvalue` arrays

  A non-`lvalue` array expression is converted to a pointer to the first element of the array when it is used.

- Comments at the end of preprocessor directives

  This extension, which makes it legal to place text after preprocessor directives, is enabled, unless strict ISO/ANSI mode is used. The purpose of this language extension is to support compilation of legacy code; we do *not* recommend that you write new code in this fashion.

- An extra comma at the end of `enum` lists

  Placing an extra comma is allowed at the end of an `enum` list. In strict ISO/ANSI mode, a warning is issued.

- A label preceding a `}`

  In ISO/ANSI C, a label must be followed by at least one statement. Therefore, it is illegal to place the label at the end of a block. In the ColdFire IAR C/C++ Compiler, a warning is issued.

Note: This also applies to the labels of `switch` statements.

- Empty declarations

  An empty declaration (a semicolon by itself) is allowed, but a remark is issued (provided that remarks are enabled).

- Single-value initialization

  ISO/ANSI C requires that all initializer expressions of static arrays, `structs`, and `unions` are enclosed in braces.

  Single-value initializers are allowed to appear without braces, but a warning is issued. In the ColdFire IAR C/C++ Compiler, the following expression is allowed:

  ```
  struct str
  {
    int a;
  } x = 10;
  ```

- Declarations in other scopes

  External and static declarations in other scopes are visible. In the following example, the variable `y` can be used at the end of the function, even though it should only be visible in the body of the `if` statement. A warning is issued.

```
int test(int x)
{
  if (x)
  {
    extern int y;
    y = 1;
  }

  return y;
}
```

- Expanding function names into strings with the function as context

  Use any of the symbols `__func__` or `__FUNCTION__` inside a function body to make it expand into a string, with the function name as context. Use the symbol `__PRETTY_FUNCTION__` to also include the parameter types and return type. The result might, for example, look like this if you use the `__PRETTY_FUNCTION__` symbol:

  ```
  "void func(char)"
  ```

  These symbols are useful for assertions and other trace utilities and they require that language extensions are enabled, see *-e*, page 126.

Extended keywords

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

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

General syntax rules for extended keywords

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

The ColdFire IAR C/C++ Compiler provides a set of attributes that can be used on functions or data objects to support specific features of the ColdFire microcontroller. There are two types of attributes—*type attributes* and *object attributes*:

- Type attributes affect the *external functionality* of the data object or function
- Object attributes affect the *internal functionality* of the data object or function.

The syntax for the keywords differs slightly depending on whether it is a type attribute or an object attribute, and whether it is applied to a data object or a function.

For information about how to use attributes to modify data, see the chapter *Data storage*. For information about how to use attributes to modify functions, see the chapter *Functions*. For detailed information about each attribute, see *Descriptions of extended keywords*, page 165.

Note: The extended keywords are only available when language extensions are enabled in the ColdFire IAR C/C++ Compiler.

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

Use the `-e` compiler option to enable language extensions. See *-e*, page 126 for additional information.

TYPE ATTRIBUTES

Type attributes define how a function is called, or how a data object is accessed. This means that if you use a type attribute, it must be specified both when a function or data object is defined and when it is declared.

You can either place the type attributes directly in your source code, or use the pragma directive `#pragma type_attribute`.

Type attributes can be further divided into *memory attributes* and *general type attributes*.

Memory attributes

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

- Available *function memory attributes*: `__near_func` and `__far_func`
- Available *data memory attributes*: `__near`, `__near_rel`, 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 only specify one memory attribute for each level of pointer indirection.

General type attributes

The following general type attributes are available:

- *Function type attributes* affect how the function should be called: `__interrupt` and `__monitor`
- *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 149.

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

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

Note that the attribute affects both identifiers.

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

```
#pragma type_attribute=__near
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 16.

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

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

and

```
__near char b;
char __near *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 __near * p;`	The `int` object is located in `__near` memory.
`int * __near p;`	The pointer is located in `__near` memory.
`__near int * p;`	The pointer is located in `__near` memory.

Syntax for type attributes on functions

The syntax for using type attributes on functions, differs slightly from the syntax of type attributes on data objects. For functions, the attribute must be placed either in front of the return type, alternatively in parentheses, for example:

```
__interrupt void my_handler(void);
```

or

```
void (__interrupt my_handler)(void);
```

The following declaration of `my_handler` is equivalent with the previous one:

```
#pragma type_attribute=__interrupt
void my_handler(void);
```

Syntax for type attributes on function pointers

To declare a function pointer, use the following syntax:

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

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

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

```
typedef __far_func void FUNC_TYPE(int);
typedef FUNC_TYPE *FUNC_PTR_TYPE;
FUNC_TYPE func();
FUNC_PTR_TYPE funcptr;
```

Note that `#pragma type_attribute` can be used together with a `typedef` declaration.

OBJECT ATTRIBUTES

Object attributes affect the internal functionality of functions and data objects, but not how the function is called or how the data is accessed. This means that an object attribute does not need to be present in the declaration of an object.

The following object attributes are available:

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

You can specify as many object attributes as required for a specific function or data object.

For more information about `location` and `@`, see *Controlling data and function placement in memory*, page 94.

Syntax for object attributes

The object attribute must be placed in front of the type. For example, to place `myarray` in memory that is not initialized at startup:

```
__no_init int myarray[10];
```

The `#pragma object_attribute` directive can also be used. The following declaration is equivalent to the previous one:

```
#pragma object_attribute=__no_init
int myarray[10];
```

Note: Object attributes cannot be used in combination with the `typedef` keyword.

Summary of extended keywords

The following table summarizes the extended keywords:

Extended keyword	Description
`__far`	Controls the storage of data objects
`__far_func`	Controls the storage of functions
`__interrupt`	Supports interrupt functions
`__intrinsic`	Reserved for compiler internal use only
`__monitor`	Supports atomic execution of a function
`__near`	Controls the storage of data objects
`__near_rel`	Controls the storage of data objects
`__near_func`	Controls the storage of functions
`__no_init`	Supports non-volatile memory
`__noreturn`	Informs the compiler that the declared function will not return
`__root`	Ensures that a function or variable is included in the object code even if unused

Table 31: Extended keywords summary

Descriptions of extended keywords

The following sections give detailed information about each extended keyword.

__far

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 161.

Description

The `__far` memory attribute explicitly places individual variables and constants in far memory.

Storage information

- Address range: Anywhere in memory
- Maximum object size: 4 Gbytes
- Pointer size: 4 bytes

Example

```
__far int x;
```

See also

Memory types, page 13.

__far_func

Syntax

Follows the generic syntax rules for memory type attributes that can be used on functions, see *Type attributes*, page 161.

Description

The `__far_func` memory attribute overrides the default storage of functions given by the selected code model and places individual functions in farfunc memory.

Storage information

- Address range: Anywhere in memory
- Maximum size: 4 Gbytes
- Pointer size: 4 bytes

Example

```
__far_func void myfunction(void);
```

See also

Code models and memory attributes for function storage, page 21.

__interrupt

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 161.

Description

The `__interrupt` keyword specifies interrupt functions.

An interrupt function must have a `void` return type and cannot have any parameters.

Example

```
__interrupt void my_interrupt_handler(void);
```

See also

Interrupt functions, page 23 and *INTVEC*, page 208.

__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 161.

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 24. Read also about the intrinsic functions *__disable_interrupt*, page 186 *__get_status_register*, page 186, and *__set_status_state*, page 187.

__near

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 161.

Description

The `__near` memory attribute overrides the default storage of variables and places individual variables and constants in near memory.

Storage information

- Address range: `0x0–07FFF` and `0xFFFF8000–0xFFFFFFFF` (64 Kbytes)
- Maximum object size: 32 Kbytes
- Pointer size: 4 bytes

Example

```
__near int x;
```

See also

Memory types, page 13.

__near_rel

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 161.

Description

The `__near_rel` memory attribute overrides the default storage of variables and places individual variables and constants in a position-independent 64-Kbyte memory area. Note that this keyword is only available in the Near relative data model.

Storage information

- Address range: Anywhere in memory

- Maximum object size: 64 Kbytes–1 byte
- Pointer size: 4 bytes

Example

```
__near_rel int x;
```

See also *Memory types*, page 13.

__near_func

Syntax: Follows the generic syntax rules for memory type attributes that can be used on functions, see *Type attributes*, page 161.

Description: The `__near_func` memory attribute overrides the default storage of functions given by the selected code model and places individual functions in nearfunc memory.

Storage information

- Address range: `0x0–07FFF` and `0xFFFF8000–0xFFFFFFFF` (64 Kbytes)
- Maximum size: 32 Kbytes. A function cannot cross a 64-Kbyte boundary.
- Pointer size: 4 bytes

Example

```
__near_func void myfunction(void);
```

See also *Code models and memory attributes for function storage*, page 21.

__no_init

Syntax: Follows the generic syntax rules for object attributes, see *Object attributes*, page 164.

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 164.

Description: The `__noreturn` keyword can be used on a function to inform the compiler that the function will not return. If you use this keyword on such functions, the compiler can optimize more efficiently. Examples of functions that do not return are `abort` and `exit`.

Example `__noreturn void terminate(void);`

__root

Syntax Follows the generic syntax rules for object attributes, see *Object attributes*, page 164.

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.*

Pragma directives

This chapter describes the pragma directives of the ColdFire IAR C/C++ Compiler.

The #pragma directive is defined by the ISO/ANSI C standard and is a mechanism for using vendor-specific extensions in a controlled way to make sure that the source code is still portable.

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

The pragma directives are always enabled in the compiler.

Summary of pragma directives

The following table lists the pragma directives of the compiler that can be used either with the `#pragma` preprocessor directive or the `_Pragma()` preprocessor operator:

Pragma directive	Description
`bitfields`	Controls the order of bitfield members
`data_alignment`	Gives a variable a higher (more strict) alignment
`diag_default`	Changes the severity level of diagnostic messages
`diag_error`	Changes the severity level of diagnostic messages
`diag_remark`	Changes the severity level of diagnostic messages
`diag_suppress`	Suppresses diagnostic messages
`diag_warning`	Changes the severity level of diagnostic messages
`include_alias`	Specifies an alias for an include file
`inline`	Inlines a function
`language`	Controls the IAR Systems language extensions
`location`	Specifies the absolute address of a variable, or places groups of functions or variables in named segments
`message`	Prints a message
`object_attribute`	Changes the definition of a variable or a function

Table 32: Pragma directives summary

Pragma directive	Description
`optimize`	Specifies the type and level of an optimization
`pack`	Specifies the alignment of structures and union members
`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
`segment`	Declares a segment name to be used by intrinsic functions
`type_attribute`	Changes the declaration and definitions of a variable or function

Table 32: Pragma directives summary (Continued)

Note: For portability reasons, see also *Recognized pragma directives (6.8.6)*, page 219 and.

Descriptions of pragma directives

This section gives detailed information about each pragma directive.

bitfields

Syntax

```
#pragma bitfields=disjoint_types|joint_types|
                  reversed_disjoint_types|reversed|default|}
```

Parameters

`disjoint_types`	Bitfield members are placed from the least significant bit to the most significant bit. Storage containers of bitfields with different base types may not overlap.
`joint_types`	Bitfield members are placed depending on byte order, see *Bitfields*, page 143. Storage containers of bitfields may overlap other structure members.
`reversed_disjoint_types`	Bitfield members are placed from the most significant bit to the least significant bit. Storage containers of bitfields with different base types may *not* overlap.
`reversed`	This is an alias for `reversed_disjoint_types`.
`default`	The default behavior for the ColdFire IAR C/C++ Compiler is `joint_types`.

Description

Use this pragma directive to control the layout of bitfield members.

See also *Bitfields*, page 143.

data_alignment

Syntax `#pragma data_alignment=`*`expression`*

Parameters

expression	A constant which must be a power of two (1, 2, 4, etc.).

Description Use this pragma directive to give a variable a higher (more strict) alignment than it would otherwise have. It can be used on variables with static and automatic storage duration.

When you use this directive on variables with automatic storage duration, there is an upper limit on the allowed alignment for each function, determined by the calling convention used.

diag_default

Syntax `#pragma diag_default=`*`tag`*`[,`*`tag`*`,...]`

Parameters

tag	The number of a diagnostic message, for example the message number `Pe117`.

Description Use this pragma directive to change the severity level back to default, or to the severity level defined on the command line by using any of the options `--diag_error`, `--diag_remark`, `--diag_suppress`, or `--diag_warnings`, for the diagnostic messages specified with the tags.

See also *Diagnostics*, page 112.

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 112.

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 112.

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 112.

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 112.

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 110.

inline

Syntax

```
#pragma inline[=forced]
```

Parameters

`forced`	Disables the compiler's heuristics and forces inlining.

Description

Use this pragma directive to advise the compiler that the function whose declaration follows immediately after the directive should be inlined—that is, expanded into the body of the calling function. Whether the inlining actually takes place is subject to the compiler's heuristics.

This is similar to the C++ keyword `inline`, but has the advantage of being available in C code.

Specifying `#pragma inline=forced` disables the compiler's heuristics and forces inlining. If the inlining fails for some reason, for example if it cannot be used with the function type in question (like `printf`), an error message is emitted.

Note: Because specifying `#pragma inline=forced` disables the compiler's heuristics, including the inlining heuristics, the function declared immediately after the directive will not be inlined on optimization levels None or Low. No error or warning message will be emitted.

language

Syntax

```
#pragma language={extended|default}
```

Parameters

`extended`	Turns on the IAR Systems language extensions and turns off the `--strict_ansi` command line option.
`default`	Uses the language settings specified by compiler options.

Description

Use this pragma directive to enable the compiler language extensions or for using the language settings specified on the command line.

location

Syntax

```
#pragma location={address|NAME}
```

Parameters

`address`	The absolute address of the global or static variable for which you want an absolute location.
`NAME`	A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.

Description

Use this pragma directive to specify the location—the absolute address—of the global or static variable whose declaration follows the pragma directive. The variable must be declared `__no_init`. 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=0xFF2000
__no_init volatile char PORT1; /* PORT1 is located at address
                                  0xFF2000 */
```

```
#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 94.

message

Syntax

`#pragma message(`*`message`*`)`

Parameters

`message`	The message that you want to direct to `stdout`.

Description

Use this pragma directive to make the compiler print a message to `stdout` when the file is compiled.

Example:

```
#ifdef TESTING
#pragma message("Testing")
#endif
```

object_attribute

Syntax

`#pragma object_attribute=`*`object_attribute`*`[,`*`object_attribute,...`*`]`

Parameters

For a list of object attributes that can be used with this pragma directive, see *Object attributes*, page 164.

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 161.

optimize

Syntax

```
#pragma optimize=param[ param...]
```

Parameters

`balanced|size|speed`	Optimizes balanced between speed and size, optimizes for size, or optimizes for speed
`none|low|medium|high`	Specifies the level of optimization
`no_code_motion`	Turns off code motion
`no_cse`	Turns off common subexpression elimination
`no_inline`	Turns off function inlining
`no_tbaa`	Turns off type-based alias analysis
`no_unroll`	Turns off loop unrolling

Description

Use this pragma directive to decrease the optimization level, or to turn off some specific optimizations. This pragma directive only affects the function that follows immediately after the directive.

The parameters `speed`, `size`, and `balanced` only have effect on the `high` optimization level and only one of them can be used as it is not possible to optimize for speed and size at the same time. It is also not possible to use preprocessor macros embedded in this pragma directive. Any such macro will not be expanded by the preprocessor.

Note: If you use the `#pragma optimize` directive to specify an optimization level that is higher than the optimization level you specify using a compiler option, the pragma directive is ignored.

Example

```
#pragma optimize=speed
int small_and_used_often()
{
    ...
}

#pragma optimize=size no_inline
int big_and_seldom_used()
{
    ...
}
```

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 alignment of structs 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 that accessing an object that is not aligned at its correct alignment requires code that is both larger and slower than the code needed to access the same kind of object when aligned correctly. If there are many accesses to such fields in the program, it is usually better to construct the correct values in a struct that is not packed, and access this instead.

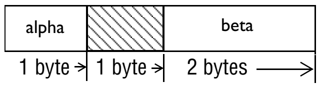
Also, special care is needed when creating and using pointers to misaligned fields. For direct access to misaligned fields in a packed struct, the compiler will emit the correct (but slower and larger) code when needed. However, when a misaligned field is accessed through a pointer to the field, the normal (smaller and faster) code for accessing the type of the field is used. In the general case, this will not work.

Example 1

This example declares a structure without using the `#pragma pack` directive:

```
struct First
{
  char alpha;
  short beta;
};
```

In this example, the structure `First` is not packed and has the following memory layout:



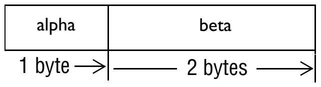
Note that one pad byte has been added.

Example 2

This example declares a similar structure using the `#pragma pack` directive:

```
#pragma pack(1)

struct FirstPacked
{
  char alpha;
  short beta;
};

#pragma pack()
```

In this example, the structure `FirstPacked` is packed and has the following memory layout:

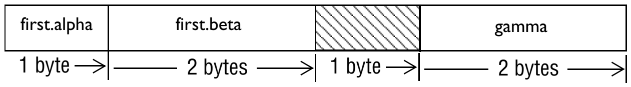


Example 3

This example declares a new structure, `Second`, that contains the structure `FirstPacked` declared in the previous example. The declaration of `Second` is not placed inside a `#pragma pack` block:

```
struct Second
{
  struct FirstPacked first;
  short gamma;
};
```

The following memory layout is used:



Note that the structure `FirstPacked` will use the memory layout, size, and alignment described in Example 2. The alignment of the member `gamma` is 2, which means that alignment of the structure `Second` will become 2 and one pad byte will be added.

required

Syntax

`#pragma required=`*`symbol`*

Parameters

`symbol`	Any statically linked function or variable.

Description

Use this pragma directive to ensure that a symbol which is needed by a second symbol is included in the linked output. The directive must be placed immediately before the second symbol.

Use the directive if the requirement for a symbol is not otherwise visible in the application, for example if a variable is only referenced indirectly through the segment it resides in.

Example

```
const char copyright[] = "Copyright by me";
...
#pragma required=copyright
int main[]
{...}
```

Even if the copyright string is not used by the application, it will still be included by the linker and available in the output.

rtmodel

Syntax

`#pragma rtmodel="`*`key`*`","`*`value`*`"`

Parameters

`"`*`key`*`"`	A text string that specifies the runtime model attribute.

`"value"`	A text string that specifies the value of the runtime model attribute. Using the special value * is equivalent to not defining the attribute at all.

Description

Use this pragma directive to add a runtime model attribute to a module, which can be used by the linker to check consistency between modules.

This pragma directive is useful for enforcing consistency between modules. All modules that are linked together and define the same runtime attribute key must have the same value for the corresponding key, or the special value `*`. It can, however, be useful to state explicitly that the module can handle any runtime model.

A module can have several runtime model definitions.

Note: The predefined compiler runtime model attributes start with a double underscore. In order to avoid confusion, this style must not be used in the user-defined attributes.

Example

```
#pragma rtmodel="I2C","ENABLED"
```

The linker will generate an error if a module that contains this definition is linked with a module that does not have the corresponding runtime model attributes defined.

See also

Checking module consistency, page 65.

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="MYHUGE" __near 4
```

See also

Important language extensions, page 152. 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 161.

Description

Use this pragma directive to specify IAR-specific *type attribute*s, which are not part of the ISO/ANSI C language standard. Note however, that a given type attribute may not be applicable to all kind of objects.

This directive affects the declaration of the identifier, the next variable, or the next function that follows immediately after the pragma directive.

Example

In the following example, an `int` object with the memory attribute `__near` is defined:

```
#pragma type_attribute=__near
int x;
```

The following declaration, which uses extended keywords, is equivalent:

```
__near int x;
```

See also

See the chapter *Extended keywords* for more details.

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.

Intrinsic functions summary

To use intrinsic functions in an application, include the header file `intrinsics.h`.

Note that the intrinsic function names start with double underscores, for example:

`__disable_interrupt`

The following table summarizes the intrinsic functions:

Intrinsic function	Description
`__disable_interrupt`	Disables interrupts
`__enable_interrupt`	Enables interrupts
`__get_status_register`	Returns the status register including interrupt state
`__halt`	Inserts a `HALT` instruction
`__no_operation`	Inserts a `NOP` instruction
`__pulse`	Inserts a `PULSE` instruction
`__set_status_register`	Sets the status register including interrupt state
`__stop`	Inserts a `STOP` instruction
`__trap_false`	Inserts a `TPF` instruction

Table 33: Intrinsic functions summary

Descriptions of intrinsic functions

The following section gives reference information about each intrinsic function.

__disable_interrupt

Syntax

```
void __disable_interrupt(void);
```

Description

Disables interrupts by setting the interrupt level to 7.

__enable_interrupt

Syntax

```
void __enable_interrupt(void);
```

Description

Enables interrupts by inserting the zeroes in the interrupt priority mask of the status register.

__get_status_register

Syntax

```
__istate_t __get_status_register(void);
```

Description

Returns the status register which includes the interrupt level. The return value can be used as an argument to the `__set_status_register` intrinsic function, which will restore the interrupt state.

Example

```
__istate_t s = __get_status_register();
__disable_interrupt();

  /* Do something */

__set_status_register(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.

__halt

Syntax

```
void __halt(void);
```

Description

Inserts a `HALT` instruction.

__no_operation

Syntax `void __no_operation(void);`

Description Inserts a `NOP` instruction.

__pulse

Syntax `void __pulse(void);`

Description Inserts a `PULSE` instruction.

__set_status_state

Syntax `void __set_status_register(__istate_t);`

Descriptions Restores the status register and interrupt level by setting the value returned by the `__get_status_register` function.

For information about the `__istate_t` type, see *__get_status_register*, page 186.

__stop

Syntax `void __stop(unsigned short);`

Description Inserts a `STOP` instruction.

__trap_false

Syntax `void __trap_false(void);`

Description Inserts a `TPF` 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 ColdFire IAR C/C++ Compiler adheres to the ISO/ANSI standard. The compiler also makes the following preprocessor-related features available to you:

- Predefined preprocessor symbols

  These symbols allow you to inspect the compile-time environment, for example the time and date of compilation. For details, see *Descriptions of predefined preprocessor symbols*, page 190.
- 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 121.
- 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 192.
- Preprocessor output

  Use the option `--preprocess` to direct preprocessor output to a named file, see *--preprocess*, page 138.

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 218.

Descriptions of predefined preprocessor symbols

The following table describes the predefined preprocessor symbols:

Predefined symbol	Identifies
`__BASE_FILE__`	A string that identifies the name of the base source file (that is, not the header file), being compiled. See also *`__FILE__`*, page 190, and *--no_path_in_file_macros*, page 133.
`__BUILD_NUMBER__`	A unique integer that identifies the build number of the compiler currently in use. The build number does not necessarily increase with a compiler that is released later.
`__CODE_MODEL__`	An integer that identifies the code model in use. The symbol reflects the `--code_model` option and is defined to `__CODE_MODEL_NEAR__` or `__CODE_MODEL_FAR__`. These symbolic names can be used when testing the `__CODE_MODEL__` symbol.
`__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_REL__` or `__DATA_MODEL_FAR__`. 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 2005`". *
`__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.*
`__FILE__`	A string that identifies the name of the file being compiled, which can be the base source file as well as any included header file. See also *`__BASE_FILE__`*, page 190, and *--no_path_in_file_macros*, page 133.*

Table 34: Predefined symbols

Predefined symbol	Identifies
`__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 126. See also *`__PRETTY_FUNCTION__`*, page 191.
`__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 126. See also *`__PRETTY_FUNCTION__`*, page 191.
`__IAR_SYSTEMS_ICC__`	An integer that identifies the IAR compiler platform. The current value is 6. Note that the number could be higher in a future version of the product. This symbol can be tested with `#ifdef` to detect whether the code was compiled by a compiler from IAR Systems.
`__ICCCF__`	An integer that is set to `1` when the code is compiled with the ColdFire IAR C/C++ Compiler.
`__ISA__`	An integer that identifies the code model in use. The symbol reflects the `--isa` option and is defined to `__ISA_A__`, `__ISA_A_PLUS__`, `__ISA_B__`, or `__ISA_C__`. These symbolic names can be used when testing the `__ISA__` symbol.
`__LINE__`	An integer that identifies the current source line number of the file being compiled, which can be the base source file as well as any included header file.*
`__LITTLE_ENDIAN__`	An integer that identifies the byte order of the microcontroller. For the ColdFire microcontroller families, the value of this symbol is defined to 0 (`FALSE`), which means that the byte order is big-endian.
`__PRETTY_FUNCTION__`	A string that identifies the function name, including parameter types and return type, of the function in which the symbol is used, for example `"void func(char)"`. This symbol is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see -e, page 126. See also *`__func__`*, page 191.

Table 34: Predefined symbols (Continued)

Predefined symbol	Identifies
`__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.*
`__SUBVERSION__`	An integer that identifies the version letter of the compiler version number, for example the C in 4.21C, as an ASCII character.
`__TIME__`	A string that identifies the time of compilation in the form `"hh:mm:ss"`.*
`__VER__`	An integer that identifies the version number of the IAR compiler in use. The value of the number is calculated in the following way: `(100 * the major version number + the minor version number)`. For example, for compiler version 3.34, 3 is the major version number and 34 is the minor version number. Hence, the value of `__VER__` is 334.

Table 34: Predefined symbols (Continued)

* **This symbol is required by the ISO/ANSI standard.**

Descriptions of miscellaneous preprocessor extensions

The following section gives reference information about the preprocessor extensions that are available in addition to the predefined symbols, pragma directives, and ISO/ANSI directives.

NDEBUG

Description

This preprocessor symbol determines whether any assert macros you have written in your application shall be included or not in the built application.

If this symbol is not defined, all assert macros are evaluated. If the symbol is defined, all assert macros are excluded from the compilation. In other words, if the symbol is:

- **defined**, the assert code will *not* be included
- **not defined**, the assert code will be included

This means that if you have written any assert code and build your application, you should define this symbol to exclude the assert code from the final application.

Note that the assert macro is defined in the `assert.h` standard include file.

In the IAR Embedded Workbench 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 has the equivalent effect of the `#pragma` directive.

Note: The `-e` option—enable language extensions—does not have to be specified.

Example

```
#if NO_OPTIMIZE
  #define NOOPT _Pragma("optimize=2")
#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
...
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 ColdFire IAR C/C++ Compiler comes with the IAR DLIB Library, which is a complete ISO/ANSI C and C++ library. This library also supports floating-point numbers in IEEE 754 format and it can be configured to include different levels of support for locale, file descriptors, multibyte characters, et cetera.

For detailed information about the library functions, see the online documentation supplied with the product. There is also keyword reference information for the DLIB library functions. To obtain reference information for a function, select the function name in the editor window and press F1.

For additional information about library functions, see the chapter *Implementation-defined behavior* in this guide.

HEADER FILES

Your application program gains access to library definitions through header files, which it incorporates using the `#include` directive. The definitions are divided into a number of different header files, each covering a particular functional area, letting you include just those that are required.

It is essential to include the appropriate header file before making any reference to its definitions. Failure to do so can cause the call to fail during execution, or generate error or warning messages at compile time or link time.

LIBRARY OBJECT FILES

Most of the library definitions can be used without modification, that is, directly from the library object files that are supplied with the product. For information about how to choose a runtime library, see *Basic settings for project configuration*, page 5. The linker will include only those routines that are required—directly or indirectly—by your application.

REENTRANCY

A function that can be simultaneously invoked in the main application and in any number of interrupts is reentrant. A library function that uses statically allocated data is therefore not reentrant.

Most parts of the DLIB library are reentrant, but the following functions and parts are not reentrant as they need static data:

- Heap functions—`malloc`, `free`, `realloc`, `calloc`, as well as 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.

In addition, some functions share the same storage for `errno`. These functions are not reentrant, since an `errno` value resulting from one of these functions can be destroyed by a subsequent use of the function before it has been read. Among these functions are:

```
exp, exp10, ldexp, log, log10, pow, sqrt, acos, asin, atan2,
cosh, sinh, strtod, strtol, strtoul
```

Remedies for this are:

- Do not use non-reentrant functions in interrupt service routines
- Guard calls to a non-reentrant function by a mutex, or a secure region, etc.

IAR DLIB Library

The IAR DLIB Library provides most of the important C and C++ library definitions that apply to embedded systems. These are of the following types:

- Adherence to a free-standing implementation of the ISO/ANSI standard for the programming language C. For additional information, see the chapter *Implementation-defined behavior* in this guide.
- Standard C library definitions, for user programs.
- Embedded C++ library definitions, for user programs.
- `CSTARTUP`, the module containing the start-up code. It is described in the chapter *The DLIB runtime environment* in this guide.
- Runtime support libraries; for example low-level floating-point routines.

- Intrinsic functions, allowing low-level use of ColdFire 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 200.

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
`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 35: Traditional standard C header files—DLIB

C++ HEADER FILES

This section lists the C++ header files.

Embedded C++

The following table lists the Embedded C++ header files:

Header file	Usage
`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 36: 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 37: 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 38: 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`

Table 39: New standard C header files—DLIB

Header file	Usage
`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.h`	Adds support for the `bool` data type in C.
`cstddef`	Defining several useful types and macros
`cstdint.h`	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.h`	Support for wide characters
`cwctype.h`	Classifying wide characters

Table 39: New standard C header files—DLIB (Continued)

LIBRARY FUNCTIONS AS INTRINSIC FUNCTIONS

Certain C library functions will under some circumstances be handled as intrinsic functions and will generate inline code instead of an ordinary function call, for example `memcpy`, `memset`, and `strcat`.

ADDED C FUNCTIONALITY

The IAR DLIB Library includes some added C functionality, 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.

Segment reference

The ColdFire IAR C/C++ Compiler places code and data into named segments which are referred to by the IAR XLINK Linker. Details about the segments are required for programming assembler language modules, and are also useful when interpreting the assembler language output from the compiler.

For 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 ColdFire IAR C/C++ Compiler:

Segment	Description
`CODE`	Holds the `__near_func` program code.
`CSTACK`	Holds the stack used by C or C++ programs.
`DIFUNCT`	Holds pointers to code, typically C++ constructors, that should be executed by the system startup code before `main` is called.
`EARLYDIFUNCT`	Holds the early dynamic initialization vector used by C++.
`FAR_AN`	Holds `__far` located uninitialized data.
`FAR_C`	Holds `__far` constant data.
`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.
`FCODE`	Holds the `__far_func` program code.
`HEAP`	Holds the heap used for dynamically allocated data.
`.iar.dynexit`	Holds the `atexit` table.
`INTVEC`	Contains the reset and interrupt vectors.
`NEAR_AN`	Holds `__near` located uninitialized data.
`NEAR_C`	Holds `__near` constant data.
`NEAR_I`	Holds `__near` static and global initialized variables.

Table 40: Segment summary

Segment	Description
NEAR_ID	Holds initial values for `__near` static and global variables in NEAR_I.
NEAR_N	Holds `__no_init __near` static and global variables.
NEAR_Z	Holds zero-initialized `__near` static and global variables.
NEARPID_I	Holds `__near_rel` static and global initialized variables.
NEARPID_ID	Holds initial values for `__near_rel` static and global variables in NEARPID_I.
NEARPID_N	Holds `__no_init __near_rel` static and global variables.
NEARPID_Z	Holds zero-initialized `__near_rel` static and global variables.
PIDBASE	Empty placeholder segment.
RCODE	Holds the startup code and other assembler support routines.

Table 40: Segment summary (Continued)

Descriptions of segments

This section gives reference information about each segment.

The segments are placed in memory by using the segment placement linker directives `-Z` and `-P`, for sequential and packed placement, respectively. Some segments cannot use packed placement, as their contents must be continuous.

In each description, the segment memory type—`CODE`, `CONST`, or `DATA`—indicates whether the segment should be placed in ROM or RAM memory; see Table 7, *XLINK segment memory types*, page 28.

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 28.

For detailed information about the extended keywords mentioned here, see the chapter *Extended keywords*.

CODE

Description	Holds `__near_func` program code, except the code for system initialization.
Segment memory type	`CODE`
Memory placement	This segment must be placed in one of the ranges `0x0–0x7FFF` and `0xFFFF8000–0xFFFFFFFF`.

Access type	Read-only

CSTACK

Description	Holds the internal data stack.
Segment memory type	`DATA`
Memory placement	This segment can be placed anywhere in memory.
Access type	Read/write
See also	*The stack*, page 33.

DIFUNCT

Description	Holds the dynamic initialization vector used by C++.
Segment memory type	`CONST`
Memory placement	This segment can be placed anywhere in memory.
Access type	Read-only

EARLYDIFUNCT

Description	Holds the early dynamic initialization vector used by C++.
Segment memory type	`CONST`
Memory placement	This segment can be placed anywhere in memory.
Access type	Read-only

FAR_AN

Description	Holds `__no_init __far` located data. Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive.

FAR_C

Description	Holds `__far` constant data.
Segment memory type	`CONST`
Memory placement	This segment can be placed anywhere in memory.
Access type	Read-only

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	`DATA`
Memory placement	This segment can be placed anywhere in memory.
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	`CONST`
Memory placement	This segment can be placed anywhere in memory.
Access type	Read-only

FAR_N

Description	Holds static and global `__no_init` `__far` variables.
Segment memory type	`DATA`
Memory placement	This segment can be placed anywhere in memory.
Access type	Read/write

FAR_Z

Description	Holds zero-initialized `__far` static and global variables. 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	This segment can be placed anywhere in memory.
Access type	Read/write

FCODE

Description	Holds `__far_func` program code, except the code for system initialization.
Segment memory type	`CODE`
Memory placement	This segment can be placed anywhere in memory.
Access type	Read-only

HEAP

Description	Holds the heap used for dynamically allocated data in memory, in other words data allocated by `malloc` and `free`, and in C++, `new` and `delete`.
Segment memory type	`DATA`

Memory placement	This segment must be placed in the first 64 Kbytes of memory.
Access type	Read/write
See also	*The heap*, page 35.

.iar.dynexit

Description	Holds entries for each dynamically initialized C++ object with a static life span and en entry for each call to the `atexit` function performed by your application.
Memory placement	This segment can be placed anywhere in memory.
See also	*Destruction and atexit() handling*, page 37.

INTVEC

Description	Holds the interrupt vector table.
Segment memory type	`CODE`
Memory placement	This segment must be placed where the interrupt vector table is located.
Access type	Read-only

NEAR_AN

Description	Holds `__no_init __near` located data. Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. *Located* means being placed at an absolute location using the `@` operator or the `#pragma location` directive.

NEAR_C

Description	Holds `__near` constant data.
Segment memory type	`CONST`
Memory placement	This segment must be placed in one of the ranges `0x0-0x7FFF` and `0xFFFF8000-0xFFFFFFFF`.

Access type

Read-only

NEAR_I

Description

Holds `__near` static and global initialized variables initialized by copying from the segment `NEAR_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

This segment must be placed in one of the ranges `0x0-0x7FFF` and `0xFFFF8000-0xFFFFFFFF`.

Access type

Read/write

NEAR_ID

Description

Holds initial values for `__near` static and global variables in the `NEAR_I` segment. These values are copied from `NEAR_ID` to `NEAR_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

This segment can be placed anywhere in memory.

Access type

Read-only

NEAR_N

Description

Holds static and global `__no_init __near` variables.

Segment memory type

`DATA`

Memory placement

This segment must be placed in one of the ranges `0x0-0x7FFF` and `0xFFFF8000-0xFFFFFFFF`.

Access type	Read/write

NEAR_Z

Description	Holds zero-initialized `__near` static and global variables. 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	This segment must be placed in one of the ranges `0x0–0x7FFF` and `0xFFFF8000–0xFFFFFFFF`.
Access type	Read/write

NEARPID_I

Description	Holds `__near_rel` static and global initialized variables initialized by copying from the segment `NEARPID_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	This segment can be placed anywhere in memory, but the segment must be located in the same 64-Kbyte memory block as the segments `PIDBASE`, `NEARPID_N`, and `NEARPID_Z`.
Access type	Read/write

NEARPID_ID

Description	Holds initial values for `__near_rel` static and global variables in the `NEARPID_I` segment. These values are copied from `NEARPID_ID` to `NEARPID_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	This segment can be placed anywhere in memory.
Access type	Read-only

NEARPID_N

Description	Holds static and global `__no_init` `__near_rel` variables.
Segment memory type	`DATA`
Memory placement	This segment can be placed anywhere in memory, but the segment must be located in the same 64-Kbyte memory block as the segments `PIDBASE`, `NEARPID_I`, and `NEARPID_Z`.
Access type	Read/write

NEARPID_Z

Description	Holds zero-initialized `__near_rel` static and global variables. 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	This segment can be placed anywhere in memory, but the segment must be located in the same 64-Kbyte memory block as the segments `PIDBASE`, `NEARPID_N`, and `NEARPID_Z`.
Access type	Read/write

PIDBASE

Description	An empty placeholder segment that is needed in front of the `NEARPID_`*`suffix`* segments.
Segment memory type	Any type.
Memory placement	This segment can be placed anywhere in memory, but the segment must be located in the same 64-Kbyte memory block as the segments `NEARPID_I`, `NEARPID_N`, and `NEARPID_Z`. Note that the `PIDBASE` segment must be placed first in line of these segments.
Access type	Read-only

RCODE

Description	Holds the startup code and other assembler support routines. 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 can be placed anywhere in memory.
Access type	Read-only

Implementation-defined behavior

This chapter describes how the ColdFire IAR C/C++ Compiler handles the implementation-defined areas of the C language.

ISO 9899:1990, the International Organization for Standardization standard - *Programming Languages - C* (revision and redesign of ANSI X3.159-1989, American National Standard), changed by the ISO Amendment 1:1994, *Technical Corrigendum 1,* and *Technical Corrigendum 2*, contains an appendix called *Portability Issues*. The ISO appendix lists areas of the C language that ISO leaves open to each particular implementation.

Note: The ColdFire IAR C/C++ Compiler adheres to a freestanding implementation of the ISO standard for the C programming language. This means that parts of a standard library can be excluded in the implementation.

Descriptions of implementation-defined behavior

This section follows the same order as the ISO appendix. Each item covered includes references to the ISO chapter and section (in parenthesis) that explains the implementation-defined behavior.

TRANSLATION

Diagnostics (5.1.1.3)

Diagnostics are produced in the form:

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

where *`filename`* is the name of the source file in which the error was encountered, *`linenumber`* is the line number at which the compiler detected the error, *`level`* is the level of seriousness of the message (remark, warning, error, or fatal error), *`tag`* is a unique tag that identifies the message, and *`message`* is an explanatory message, possibly several lines.

ENVIRONMENT

Arguments to main (5.1.2.2.2.1)

The function called at program startup is called `main`. There is no prototype declared for `main`, and the only definition supported for `main` is:

```
int main(void)
```

To change this behavior for the IAR DLIB runtime environment, see *Customizing system initialization*, page 54.

Interactive devices (5.1.2.3)

The streams `stdin` and `stdout` are treated as interactive devices.

IDENTIFIERS

Significant characters without external linkage (6.1.2)

The number of significant initial characters in an identifier without external linkage is 200.

Significant characters with external linkage (6.1.2)

The number of significant initial characters in an identifier with external linkage is 200.

Case distinctions are significant (6.1.2)

Identifiers with external linkage are treated as case-sensitive.

CHARACTERS

Source and execution character sets (5.2.1)

The source character set is the set of legal characters that can appear in source files. The default source character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the source character set will be the host computer's default character set.

The execution character set is the set of legal characters that can appear in the execution environment. The default execution character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the execution character set will be the host computer's default character set. The IAR DLIB Library needs a multibyte character scanner to support a multibyte execution character set.

See *Locale*, page 59.

Bits per character in execution character set (5.2.4.2.1)

The number of bits in a character is represented by the manifest constant `CHAR_BIT`. The standard include file `limits.h` defines `CHAR_BIT` as 8.

Mapping of characters (6.1.3.4)

The mapping of members of the source character set (in character and string literals) to members of the execution character set is made in a one-to-one way. In other words, the same representation value is used for each member in the character sets except for the escape sequences listed in the ISO standard.

Unrepresented character constants (6.1.3.4)

The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or in the extended character set for a wide character constant generates a diagnostic message, and will be truncated to fit the execution character set.

Character constant with more than one character (6.1.3.4)

An integer character constant that contains more than one character will be treated as an integer constant. The value will be calculated by treating the leftmost character as the most significant character, and the rightmost character as the least significant character, in an integer constant. A diagnostic message will be issued if the value cannot be represented in an integer constant.

A wide character constant that contains more than one multibyte character generates a diagnostic message.

Converting multibyte characters (6.1.3.4)

The only locale supported—that is, the only locale supplied with the IAR C/C++ Compiler—is the 'C' locale. If you use the command line option `--enable_multibytes`, the IAR DLIB Library will support multibyte characters if you add a locale with multibyte support or a multibyte character scanner to the library.

See *Locale*, page 59.

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 142, 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 145, 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 147, for information about `size_t`.

Conversion from/to pointers (6.3.4)

See *Casting*, page 146, for information about casting of data pointers and function pointers.

ptrdiff_t (6.3.6, 7.1.1)

See *ptrdiff_t*, page 147, 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 142, for information about the alignment requirement for data objects.

Sign of 'plain' bitfields (6.5.2.1)

A 'plain' `int` bitfield is treated as a signed `int` bitfield. All integer types are allowed as bitfields.

Allocation order of bitfields within a unit (6.5.2.1)

Bitfields are allocated within an integer from least-significant to most-significant bit.

Can bitfields straddle a storage-unit boundary (6.5.2.1)

Bitfields cannot straddle a storage-unit boundary for the chosen bitfield integer type.

Integer type chosen to represent enumeration types (6.5.2.2)

The chosen integer type for a specific enumeration type depends on the enumeration constants defined for the enumeration type. The chosen integer type is the smallest possible.

QUALIFIERS

Access to volatile objects (6.5.3)

Any reference to an object with volatile qualified type is an access.

DECLARATORS

Maximum numbers of declarators (6.5.4)

The number of declarators is not limited. The number is limited only by the available memory.

STATEMENTS

Maximum number of case statements (6.6.4.2)

The number of case statements (case values) in a switch statement is not limited. The number is limited only by the available memory.

PREPROCESSING DIRECTIVES

Character constants and conditional inclusion (6.8.1)

The character set used in the preprocessor directives is the same as the execution character set. The preprocessor recognizes negative character values if a 'plain' character is treated as a `signed` character.

Including bracketed filenames (6.8.2)

For file specifications enclosed in angle brackets, the preprocessor does not search directories of the parent files. A parent file is the file that contains the `#include` directive. Instead, it begins by searching for the file in the directories specified on the compiler command line.

Including quoted filenames (6.8.2)

For file specifications enclosed in quotes, the preprocessor directory search begins with the directories of the parent file, then proceeds through the directories of any grandparent files. Thus, searching begins relative to the directory containing the source file currently being processed. If there is no grandparent file and the file has not been found, the search continues as if the filename was enclosed in angle brackets.

Character sequences (6.8.2)

Preprocessor directives use the source character set, with the exception of escape sequences. Thus, to specify a path for an include file, use only one backslash:

```
#include "mydirectory\myfile"
```

Within source code, two backslashes are necessary:

```
file = fopen("mydirectory\\myfile","rt");
```

Recognized pragma directives (6.8.6)

In addition to the pragma directives described in the chapter *Pragma directives*, the following directives are recognized but will have no effect:

```
alignment
ARGSUSED
baseaddr
can_instantiate
codeseg
cspy_support
define_type_info
do_not_instantiate
function
hdrstop
instantiate
keep_definition
memory
module_name
none
no_pch
NOTREACHED
```

```
once
__printf_args
public_equ
__scanf_args
section
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 62.

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 58.

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 58.

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 58.

%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 61.

system() (7.10.4.5)

How the command processor works depends on how you have implemented the `system` function. See *Environment interaction*, page 61.

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 41: 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 63.

clock() (7.12.2.1)

From where the system clock starts counting depends on how you have implemented the `clock` function. See *Time*, page 63.