IAR C/C++ Compiler

Reference Guide

for Freescale's
HCS12 Microcontroller Family

CHCS12-2

IAR SYSTEMS

EDITION NOTICE

Second edition: February 2010

Part number: CHCS12-2

This guide applies to version 3.x of IAR Embedded Workbench® for HCS12.

Brief contents

Contents

Tables

Preface

Welcome to the IAR C/C++ Compiler Reference Guide for HCS12. The purpose of this guide is to provide you with detailed reference information that can help you to use the IAR C/C++ Compiler for HCS12 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 HCS12 microcontroller and need to get detailed reference information on how to use the IAR C/C++ Compiler for HCS12. In addition, you should have a working knowledge of the following:

- The architecture and instruction set of the HCS12 microcontroller. Refer to the documentation from Freescale for information about the HCS12 microcontroller
- The C or C++ programming language
- Application development for embedded systems
- The operating system of your host machine.

How to use this guide

When you start using the IAR C/C++ Compiler for HCS12, you should read *Part 1. Using the compiler* in this guide.

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

If you are new to using the IAR toolkit, we recommend that you first study the *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 and the IAR C-SPY Debugger, and corresponding reference information. The *IAR Embedded Workbench® IDE User Guide* also contains a glossary.

What this guide contains

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

Part 1. Using the compiler

- *Getting started* gives the information you need to get started using the IAR C/C++ Compiler for HCS12 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* describes the different ways code can be generated, and introduces the concept of code models and function memory type attributes. Special function types, such as interrupt functions, are also covered.
- *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 `cstartup` file, as well as how to use modules for locale, and file I/O.
- *Assembler language interface* contains information required when parts of an application are written in assembler language. This includes the calling convention.
- *Using C++* gives an overview of the two levels of C++ support: The industry-standard EC++ and IAR Extended EC++.
- *Efficient coding for embedded applications* gives hints about how to write code that compiles to efficient code for an embedded application.

Part 2. Compiler reference

- *Data representation* describes the available data types, pointers, and structure types. This chapter also gives information about type and object attributes.
- *Segment reference* gives reference information about the compiler's use of segments.
- *Compiler options* explains how to set the compiler options, gives a summary of the options, and contains detailed reference information for each compiler option.
- *Extended keywords* gives reference information about each of the HCS12-specific keywords that are extensions to the standard C language.
- *Pragma directives* gives reference information about the pragma directives.
- *The preprocessor* gives a brief overview of the preprocessor, including reference information about the different preprocessor directives, symbols, and other related information.
- *Intrinsic functions and extended operators* gives reference information about the functions that can be used for accessing HCS12-specific low-level features.
- *Library functions* gives an introduction to the C or C++ library functions, and summarizes the header files.

- *Implementation-defined behavior* describes how the IAR C/C++ Compiler for HCS12. handles the implementation-defined areas of the C language standard.
- *IAR language extensions* describes the IAR extensions to the ISO/ANSI standard for the C programming language.
- *Diagnostics* describes how the compiler's diagnostic system works.

Other documentation

The complete set of IAR Systems development tools for the HCS12 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 IAR Assembler for HCS12, refer to the *IAR Assembler Reference Guide for HCS12*
- 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, refer to the online help system
- Porting application code and projects created with a previous 68HC12 IAR Embedded Workbench IDE, refer to *IAR Embedded Workbench Migration Guide for HCS12*.

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

FURTHER READING

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

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

We recommend that you visit the following websites:

- The Freescale website, **www.Freescale.com**, contains information and news about the HCS12 microcontroller.
- The IAR website, **www.iar.com**, holds application notes and other product information.
- Finally, the Embedded C++ Technical Committee website, **www.caravan.net/ec2plus**, contains information about the Embedded C++ standard.

Document conventions

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

TYPOGRAPHIC CONVENTIONS

This guide uses the following typographic conventions:

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

Table 1: Typographic conventions used in this guide

Part 1. Using the compiler

This part of the IAR C/C++ Compiler Reference Guide for HCS12 includes the following chapters:

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

Getting started

This chapter gives the information you need to get started using the IAR C/C++ Compiler for HCS12 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 HCS12 microcontroller. In the following chapters, these techniques will be studied in more detail.

IAR language overview

There are two high-level programming languages available for use with the IAR C/C++ Compiler for HCS12:

- C, the most widely used high-level programming language used in the embedded systems industry. Using the IAR C/C++ Compiler for HCS12, 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++*.
  - Extended EC++, with additional features such as full template support, namespace support, the new cast operators, as well as the Standard Template Library (STL).

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

It is also possible to implement parts of the application, or the whole application, in assembler language. See the *IAR Assembler Reference Guide for HCS12*.

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

Supported HCS12 derivatives

The IAR C/C++ Compiler for HCS12 supports all derivatives based on the standard Freescale HCS12 microcontroller and the classic 68HC12 microcontroller.

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 IAR C/C++ Compiler for HCS12 or the IAR Assembler for HCS12.

A library is a collection of object files. A typical example of a library is the compiler library containing the runtime environment and the C/C++ standard library. Libraries can also be built using the IAR XAR Library Builder, the IAR XLIB Librarian, or be provided by external suppliers.

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

Below, the process for building an application on the command line is described. For information about how to build an application using the IAR Embedded Workbench IDE, see the *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.r12` using the default settings:

```
icchcs12 myfile.c
```

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

LINKING

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

- A number of object files and possibly certain libraries
- The standard library containing the runtime environment and the standard language functions
- A program start label

- A linker command file that describes the memory layout of the target system
- Information about the output format.

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

```
xlink myfile.r12 myfile2.r12 -s __program_start -f lnkhcs12.xcl
dlhcs12bdn.r12 -o aout.a12 -Fmotorola
```

In this example, `myfile.r12` and `myfile2.r12` are object files, `lnkhcs12.xcl` is the linker command file, and `dlhcs12bdn.r12` is the runtime library. The option -s specifies the label where the application starts. The option `-o` specifies the name of the output file, and the option `-F` must be used for specifying the output format.

The IAR XLINK Linker produces output after 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 PROM programmer—in which case you need output without debug information, such as Intel-hex or Motorola S-records.

Basic settings for project configuration

This section gives an overview of the basic settings for the project setup that are needed to make the compiler generate the best code for the HCS12 device you are using. You can specify the options either from the command line interface or in the IAR Embedded Workbench IDE. For details about how to set options, see *Setting command line options*, page 145, and the *IAR Embedded Workbench® IDE User Guide*, respectively.

The basic settings available for the HCS12 microcontroller are:

- Code model
- Size of `double` floating-point type
- Optimization settings
- Runtime environment.

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

CODE MODEL

The IAR C/C++ Compiler for HCS12 supports *code models* that you can set on file or function level to control which function calls are generated. The following code models are available:

- The *Normal* code model has an upper limit of 64 Kbytes
- The *Banked* code model supports the inbuilt code banking via the `PPAGE` register, that is, an 8-bit bank and a bank window inside the 64 Kbyte memory area.

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

SIZE OF DOUBLE FLOATING-POINT TYPE

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

OPTIMIZATION FOR SPEED AND SIZE

The IAR C/C++ Compiler for HCS12 is a state-of-the-art compiler with an optimizer that performs, among other things, dead-code elimination, constant propagation, inlining, common sub-expression elimination, and precision reduction. It also performs loop optimizations, such as unrolling and induction variable elimination.

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

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

For details about compiler optimizations, see *Controlling compiler optimizations*, page 106. 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

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

Note that for the DLIB library there are two different configurations—Normal and Full—which include different levels of support for locale, file descriptors, multibyte characters, et cetera. See *Library configurations*, page 49, for more information.

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

Choosing a runtime library from the command line

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

Command line	Description
`-I\hcs12\inc`	Specifies the include paths
`-I\hcs12\inc\{clib|dlib}`	Specifies the library-specific include path. Use `clib` or `dlib` depending on which library you are using.
`libraryfile``.r12`	Specifies the library object file
`--dlib_config` <*`install_dir`*>`\`*`configfile`*`.h`	Specifies the library configuration file

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

For a list of all prebuilt library object files for the IAR DLIB Library, see Table 12, *Prebuilt libraries*, page 51. The table also shows how the object files correspond to the dependent project options, and the corresponding configuration files. Make sure to use the object file that matches your other project options.

Setting library and runtime environment options

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

- The formatters used by the functions `printf`, `scanf`, and their variants, see *Choosing formatters for printf and scanf*, page 52.

- The size of the stack and the heap, see *The stack*, page 38, and *The heap*, page 39, respectively.

Special support for embedded systems

This section briefly describes the extensions provided by the IAR C/C++ Compiler for HCS12 to support specific features of the HCS12 microcontroller.

EXTENDED KEYWORDS

The IAR C/C++ Compiler for HCS12 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.

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

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

PRAGMA DIRECTIVES

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

The pragma directives are always enabled in the IAR C/C++ Compiler for HCS12. 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 models.

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

SPECIAL FUNCTION TYPES

The special hardware features of the HCS12 microcontroller are supported by the compiler's special function types: interrupt, monitor, and task. You can write a complete application without having to write any of these functions in assembler language.

For detailed information, see *Special function types*, page 21.

HEADER FILES FOR I/O

Standard peripheral units are defined in device-specific I/O header files with the filename extension `h`. The product package supplies I/O files for all devices that are available at the time of the product release. You can find these files in the `hcs12/inc` directory. Make sure to include the appropriate include file in your application source files. If you need additional I/O header files, they can easily be created using one of the provided ones as a template.

For an example, see *Accessing special function registers*, page 116.

ACCESSING LOW-LEVEL FEATURES

For hardware-related parts of your application, accessing low-level features is essential. The IAR C/C++ Compiler for HCS12 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 79.

Data storage

This chapter gives a brief introduction to the memory layout of the HCS12 microcontroller and the fundamental ways data can be stored in memory: on the stack, in static (global) memory, or in heap memory. For efficient memory usage, the IAR C/C++ Compiler for HCS12 provides a set of data models and data memory attributes, allowing you to fine-tune the access methods, resulting in smaller code size. The concepts of 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 HCS12 microcontroller can address 64 Kbytes of continuous memory, ranging from `0x0000` to `0xFFFF`. Different types of physical memory can be placed in the memory range. A typical application will have both read-only memory (ROM) and read/write memory (RAM). In addition, some parts of the memory range contain processor control registers and peripheral units.

The IAR C/C++ Compiler for HCS12 can access memory in two different ways. The access methods range from a generic method that can access the full memory space—data16—to a sometimes cheaper method that can only access the zero page area—data8.

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.

Memory types

This section describes the concept of *memory types* used for accessing data by the IAR C/C++ Compiler for HCS12. It also discusses pointers in the presence of multiple memory types. For each memory type, the capabilities and limitations are discussed.

The IAR C/C++ Compiler for HCS12 uses two different memory types—data8 and data16—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. Each memory type corresponds to one memory access method. By mapping different memories—or part of memories—to memory types, the compiler can generate code that can access data efficiently.

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

It is possible to specify—for individual variables or pointers—different memory types. This makes it possible to create an application that can contain a large amount of data, and at the same time make sure that variables that are used often are placed in memory that can be efficiently accessed.

DATA16

The data16 memory consists of the full 64 Kbytes of data memory space. In hexadecimal notation, this is the address range `0x0000-0xFFFF`.

A data16 object can be placed anywhere in non-banked memory, and the size of such an object is limited to 64 Kbytes-1. Default pointer is always data16.

DATA8

The data8 memory consists of the low 256 bytes of data memory space. In hexadecimal notation, this is the address range `0x00-0xFF`. The size of an object located in data8 memory is limited to 255 bytes.

A data8 pointer occupies one byte and a data16 pointer occupies 2 bytes, which means memory is saved when using data8 pointers compared to data16 pointers. However, the drawback is that dereferencing data8 pointers is more expensive in terms of code size than using data16 pointers. Dereferencing a data8 pointer typically costs 2 bytes more, because a data8 pointer has to be loaded via an accumulator and then transferred to an index register. Arithmetics on data8 pointers can be somewhat cheaper than arithmetics on data16 pointers, especially when indexing arrays which have an element size greater than 1 byte. Direct-addressing a data8 object saves one byte per instruction.

USING DATA MEMORY ATTRIBUTES

The IAR C/C++ Compiler for HCS12 provides two *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 the data8 memory instead of in the default memory. This 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	Address range	Pointer size
Data8	`__data8`	`0-0xFF`	8 bits
Data16	`__data16`	`0-0xFFFF`	16 bits

Table 3: Memory types and their corresponding keywords

The keywords are only available if language extensions are enabled in the IAR C/C++ Compiler for HCS12.

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

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

For reference information about each keyword, see *Descriptions of extended keywords*, page 175.

Syntax

The keywords follow the same syntax as the type qualifiers `const` and `volatile`. The memory attributes are *type attributes* and therefore they must be specified both when they are defined and in the declaration, see *Type and object attributes*, page 131.

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

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

Note that the keyword affects both identifiers. If no memory type is specified, the default memory type is used, which is always data16.

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

Type definitions

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

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

and

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

POINTERS AND MEMORY TYPES

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

In the IAR C/C++ Compiler for HCS12, a pointer also points to some type of memory. The memory type is specified using a keyword before the asterisk. For example, a pointer that points to an integer stored in data8 memory is declared by:

```
int __data8 * p;
```

Note that the location of the pointer variable `p` is not affected by the keyword. In the following example, however, the pointer variable `p2` is also placed in data8 memory. Like `p`, `p2` points to a character in data8 memory.

```
char __data8 * __data8 p2;
```

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

Differences between pointer types

A pointer must contain information needed to specify a memory location of a certain memory type. This means that the pointer sizes are different for different memory types. For the IAR C/C++ Compiler for HCS12, the size of the `data8` and `data16` pointers are 8 and 16 bits, respectively.

In the IAR C/C++ Compiler for HCS12, it is illegal, with one exception, to convert pointers between different types without explicit casts. The exception is when casting from a small pointer type to a large. Because the pointer size is 8 bits for data8 pointers and 16 bits for data16 pointers, and data8 resides in the data16 area, it is legal to cast a data8 pointer to a data16 pointer without an explicit cast.

STRUCTURES AND MEMORY TYPES

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

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

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

The following declaration is incorrect:

```
struct MySecondStruct
{
  int blue;
  __data8 int green;   /* Error! */
};
```

MORE EXAMPLES

The following is a series of examples with descriptions. First, some integer variables are defined and then pointer variables are introduced. Finally, a function accepting a pointer to an integer in data16 memory is declared. The function returns a pointer to an integer in data20 memory. It makes no difference whether the memory attribute is placed before or after the data type. In order to read the following examples, start from the left and add one qualifier at each step:

`int a;`	A variable defined in default memory.
`int __data16 b;`	A variable in data16 memory.
`__data8 int c;`	A variable in data8 memory.
`int * d;`	A pointer stored in default memory. The pointer points to an integer in default memory.
`int __data8 * e;`	A pointer stored in default memory. The pointer points to an integer in data8 memory.
`int __data8 * __data16 f;`	A pointer stored in data16 memory pointing to an integer stored in data8 memory.
`int __data8 * myFunction(` `  int __data16 *);`	A declaration of a function that takes a parameter which is a pointer to an integer stored in data16 memory. The function returns a pointer to an integer stored in data8 memory.

C++ and memory types

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

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

Also note that for non-static member functions—unless class memory is used, see *Classes*, page 97—the `this` pointer will be of the default data16 pointer type. This means that it must be possible to convert a pointer to the object to the data16 pointer type. The restrictions that apply to the data16 pointer type also apply to the `this` pointer.

Example

In the example below, an object, named `delta`, of the type `MyClass` is defined in data16 memory. The class contains a static member variable that is stored in data8 memory.

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

  __data8 static int gamma;
};

// Definitions needed (should be placed in a source file):
__data8 int MyClass::gamma;

// A variable definition:
__data16 MyClass delta;
```

The stack and auto variables

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

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

The stack can contain:

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

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

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

Advantages

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

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

Potential problems

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

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

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

Dynamic memory on the heap

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

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

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

The IAR C/C++ Compiler for HCS12 supports dynamic memory allocation—the heap—in both data8 memory and data16 memory. For more information about this, see *The heap*, page 39.

Potential problems

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

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

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

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

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

Functions

This chapter contains information about functions. First, the different ways normal code can be generated and the concept of code models are introduced. Then, the special function types interrupt, monitor, and banked are described, including how to declare Embedded C++ member functions by using special function types. The last section describes how to use extended keywords when defining functions, and how to place functions into named segments.

Code models

The HCS12 microcontroller can be used in two modes: normal mode and banked mode. The IAR C/C++ Compiler for HCS12 supports these modes by means of code models.

The code model controls how code is generated for an application. All object files of an application must be compiled using the same code model. The supported code models are:

Code model	Description
Normal (default)	Non-banked function calls
Banked	Banked function calls

Table 4: Code models

For further information about the banked code model, see *Banked functions*, page 25.

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

Use the `--code_model` option to specify the code model for your project; see *--code_model*, page 150.

OVERRIDING DEFAULT CODE MODEL

It is possible to override the default code model by specifying for an individual function how it will be called, in other words, the assembler instruction used when calling the function. You specify this by using the appropriate *function memory attribute*. You can also specify a function memory attribute for each function by using the `#pragma type_attribute` directive. These attributes must be specified both when the function is declared and when it is defined. For more information about attributes, see *Type and object attributes*, page 131.

The following function memory attributes are available for overriding default code model:

Function memory attribute	Address range	Pointer size	Default in code model
`__non_banked`	`0-0xFFFF`	2 bytes	Normal
`__banked`	`0-0xFFFFFF`	3 bytes	Banked

Table 5: 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 between non-related types*, page 129.

Syntax

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

```
__banked void alpha(void);
```

The keywords that are *type* attributes must be specified both when they are defined and in the declaration.

FUNCTION DIRECTIVES

Note: The directives described in this section are primarily intended to support static overlay, a feature which is useful in some smaller microcontrollers. The IAR C/C++ Compiler for HCS12 does not use static overlay, as it has no use for it.

The function directives `FUNCTION`, `ARGFRAME`, `LOCFRAME`, and `FUNCALL` are generated by the IAR C/C++ Compiler for HCS12 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 *IAR Assembler Reference Guide for HCS12*.

Special function types

This section describes the special function types interrupt, simple, and monitor. The IAR C/C++ Compiler for HCS12 allows an application to take full advantage of these HCS12 features, without forcing you to implement anything in assembler language.

INTERRUPT FUNCTIONS

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

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

The HCS12 microcontroller supports many interrupt sources. For each interrupt source, an interrupt routine can be written. Each interrupt routine is associated with a vector number, alternatively multiple vector numbers, which is specified in the HCS12 microcontroller documentation from the chip manufacturer. The interrupt vector is specified with an absolute address.

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

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

```
#pragma vector=MIWU1  /* Symbol defined in I/O header file */
__interrupt void my_interrupt_routine(void)
{
  /* Do something */
}
```

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

If a vector is specified in the definition of an interrupt function, the processor interrupt vector table is populated. It is also possible to define an interrupt function without a vector. This is useful if an application is capable of populating or changing the interrupt vector table at runtime. See the chip manufacturer's HCS12 microcontroller documentation for more information about the interrupt vector table.

Controlling enter and leave sequences for interrupts

If you need to control the enter and leave sequences for interrupts, you should implement your own prologue and epilogue library routines. You can do this using the `#pragma context_handler` directive. A typical example when this can be useful is if you have an operating system with several processes, each having its own stack. Every interruptible process would need to reserve room for the worst case interrupt scenario. If you have many processes, this can be a substantial amount of memory.

You can find an example of this in the source file `stack_swap.s12`, located in the product installation directory.

For reference information, see *#pragma context_handler*, page 181.

NORMAL AND SIMPLE CALLING CONVENTIONS

The IAR C/C++ Compiler for HCS12 supports two different calling conventions—the default calling convention *Normal* and the *Simple* calling convention.

For more information about the two calling conventions, see *Calling convention*, page 85.

The Normal calling convention is used unless you explicitly declare a function to use the simple calling convention by using the `__simple` keyword:

```
__simple void f(void);
```

For more information about the `__simple` keyword, see *__simple*, page 178.

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 176.

Example of implementing a semaphore in C

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

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

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

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


/* release_lock -- Unlock the lock. */

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

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

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

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

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

Example of implementing a semaphore in C++

In C++, it is common to implement small methods with the intention that they should be inlined. However, the IAR C/C++ Compiler for HCS12 does not support inlining of functions and methods that are declared using the `__monitor` keyword.

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

```
#include <intrinsics.h>

volatile long tick_count = 0;

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

  ~Mutex ()
  {
    __set_interrupt_state(_state);
  }

private:
  __istate_t _state;
};


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

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

    tick = tick_count;
  }

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

C++ AND SPECIAL FUNCTION TYPES

C++ member functions can be declared using special function types. However, there is one restriction:

> 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.

Banked functions

This section introduces the banking technique. It is important to know when to use it, what it does, and how it works. The section also describes how to write and partition source code and ISRs (interrupt service routines) for banked applications, including the associated compiler and linker command file configuration. Finally, the section also discusses linker output formats suitable for banked addresses, and methods of downloading to multiple PROMs.

Note: When you read this section, you should be familiar with the basic functions of the compiler described in *Part 1. Using the compiler* in this book.

INTRODUCTION TO THE BANKING SYSTEM

If you are using a HCS12 microcontroller with a natural address range of 64 Kbytes of memory, it has a 16-bit addressing capability. *Banking* is a technique for extending the amount of memory that can be accessed by the processor beyond the limit set by the size of the natural addressing scheme of the processor. In other words, more code can be accessed.

Banked memory is used in projects which require such a large amount of executable code that the natural 64 Kbytes address range of the HCS12 microcontroller is not sufficient to contain it all.

The banked code model

The banked code model allows the code memory area of the HCS12 microcontroller to be extended with up to 256 banks of additional ROM memory. Your microcontroller provides the physical memory banks, which determine the limit for the number of banks.

As the processor cannot handle more than 64 Kbytes of memory at any single time, the extended memory range introduced by banking implies that special care must be taken. Only one bank at a time is visible. However, the bank switch is done automatically at the function calls and exits, which means a new bank is brought into the 64-Kbyte address range.

How banking works

The following memory map shows an example of a HCS12 banked code memory area:

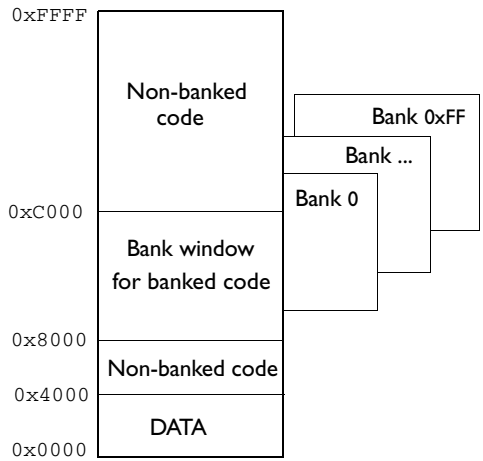


Figure 1: Banked code memory layout (example)

The bank window is normally 16 Kbytes (`0x8000–0xBFFF`). You can place the banked area anywhere in code memory, but there must always be a *non-banked* area for holding the system startup code. It is practical to place the non-banked area at address `FFFF` and downwards.

To access code residing in one of the memory banks, the compiler keeps track of the bank number of a banked function by maintaining a 3-byte pointer to it, which has the following form:

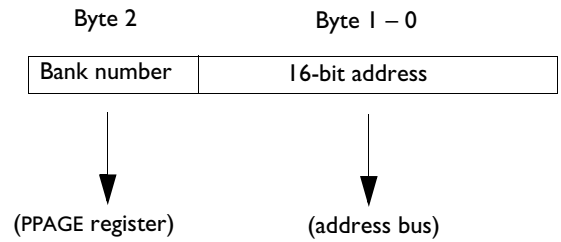


Figure 2: The 3-byte banked address

Code that cannot be banked

Code banking is achieved by dividing the address space used for program code into two parts: *non-banked* and *banked* code. There must always be a certain amount of code that is non-banked. For example, interrupt service routines must always be available, as interrupts can occur at any time. The following selected parts must be located in non-banked memory:

- Interrupt vectors (the segment `INTVEC`)
- Interrupt service routines (the segment `CODE`)
- The system startup code (the segment `CODE`)
- Functions declared `__non_banked` (the segment `CODE`)
- Constants (the segments `DIFUNCT`, `DATA8_C`, and `DATA16_C`)
- Selected parts of the runtime library, such as support routines for switch statements (the segment `CODE`).

Code located in non-banked memory will always be available to the processor, and will always be located at the same internal address.

WRITING SOURCE CODE FOR BANKED MEMORY

Writing code to be used in banked memory is not much different from writing code for standard memory, but there are a few issues to be aware of. These primarily concern partitioning your code into source modules so that they can be most efficiently placed into banks by the linker, and the distinction between banked versus non-banked code.

To read about banked data, see *Banked constant data*, page 118.

C/C++ language considerations

From the C/C++ language standpoint, any arbitrary C/C++ program can be compiled for banked memory. The only restriction is the size of the function, since it cannot be larger than the size of a bank.

Bank size and code size considerations

Each banked C/C++ source function that you compile will generate a separate *segment part*, and all segment parts generated from banked functions will be located in the `BANKED_CODE` segment. The code contained in a segment part is an *indivisible unit*. The linker cannot break up a code contained in indivisible units and place part of it in one bank and part of it in another bank. Thus, the code produced from a banked function must always be smaller than the bank size.

This means that you have to consider the size of each C/C++ source file so that the generated code will fit into your banks. If any of your code segments is larger than the specified bank size, the linker will issue an error.

Some optimizations—Cross call—on high optimization level will form indivisible units of several segment parts, or in other words, several segment parts might be connected into one larger indivisible unit. This might lead to indivisible units not fitting your bank size. To solve this, you can either disable the optimization or repartition your code.

If you need to specify the location of any code individually, you can rename the code segment for each function to a distinct name that will allow you to refer to it individually. To assign a function to a specific segment, use the `@` operator, the `#pragma location` directive, or the `#pragma segment` directive.

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

Banked versus non-banked function calls

Differentiating between the non-banked versus banked function calls is important because non-banked function calls are somewhat faster and take up less code space than banked function calls. Therefore, it is useful to be familiar with which types of function declarations that result in non-banked function calls.

In this context, a *function call* is the sequence of machine instructions generated by the compiler whenever a function in your C/C++ source code calls another C/C++ function or library routine. This also includes saving the return address and then sending the new execution address to the hardware.

There are two function call sequences:

- Non-banked function calls: The return address and new execution address are 16-bit (2 bytes) values. Non-banked function calls are default in the standard code model.
- Banked function calls: The return address and new execution address are 24-bit (3 bytes) values (default in the banked code model).

Functions declared using the `__non_banked` memory attribute are located in the non-banked code area, which means the functions can be called with non-banked function calls. For functions that are frequently called, it can be recommended to place them in the non-banked code area.

Example of a non-banked function call in the banked code model

In this example we assume that `f1()` will call `f2()`. They are defined in separate source modules. Then the definition of `f2()` would look like:

```
__non_banked void f2(void) /* simple void function example */
{
  /* code here...*/
}
```

The function prototype for `f2()` in the module where `f1()` will call `f2()` would be:

```
extern __non_banked void f2(void);
```

The actual call to `f2()` from within `f1()` would be exactly as any other function, for example:

```
void f1(void)
{
  f2();
}
```

Interrupt functions in banked mode

The code for interrupt service routines are by default non-banked. This means that these functions are located in the `CODE` segment. The compiler handles this automatically.

Calling banked functions from assembler language

In an assembler language program, calling a C/C++ function located in another bank requires using the same protocol as the compiler. For information about this protocol, see *Calling convention*, page 85. To generate an example of a banked function call, use the technique described in the section *Calling assembler routines from C*, page 82.

USING THE BANKED CODE MODEL

When you create a banked application, you must be aware of the associated compiler and linker command file configuration.

Compiler options for banked mode

To compile your modules for banked mode, use the Banked code model option (`--code_model banked`). To place an individual function in non-banked memory when using the banked code model, declare the function with the `__non_banked` attribute. To place an individual function in banked memory when using the Normal code model, declare the function with the `__banked` attribute.

Linker options for banked mode

When linking an application, you must place your code segments into banks corresponding to the available physical banks in your hardware. To achieve the most efficient memory usage, use the `-P` linker directive to control the placement of segments.

For an example, see *Banked code*, page 41.

For an example about how to place data initializers in banked memory, see *Banked data initializers*, page 117.

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 IAR C/C++ Compiler for HCS12 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 *Segment reference* chapter in *Part 2. Compiler reference*.

Segment memory type

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

By default, the IAR C/C++ Compiler for HCS12 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 6: XLINK segment memory types

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

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

Placing segments in memory

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

In particular, the linker command file specifies:

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

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

CUSTOMIZING THE LINKER COMMAND FILE

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

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

Range	Description
0x00–0xFF	Data8 RAM memory
0x1000–0x3FFF	Data16 RAM memory
0x4000–0x7FFF	Data16 ROM memory
0x8000–0xBFFF	Data16 ROM memory for eight 16-Kbyte banks
0xC000–0xFFFF	Data16 ROM memory

Table 7: Memory layout of a target system (example)

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

The contents of the linker command file

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

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

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

- The CPU used:
  `-chcs12`

  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.

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

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

Using the -Z command for sequential placement

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

The following illustrates how to use the `-Z` command to place the segment `MYSEGMENTA` followed by the segment `MYSEGMENTB` in `CONST` memory (that is, ROM) in the memory range `0x4000-0x7FFF`.

```
-Z(CONST)MYSEGMENTA,MYSEGMENTB=4000-7FFF
```

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=4000-7FFF
-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=4000-40FF
-Z(CONST)MYLARGESEGMENT=4000-7FFF
```

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,8000-8FFF
```

If your application has an additional RAM area in the memory range `0x3000-0x3FFF`, you just add that to the original definition:

```
-P(DATA)MYDATA=0-1FFF,3000–3FFF,8000-8FFF
```

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

Using the -P command for banked placement

The `-P` command is also useful for banked segment placement, that is, code that should be divided into several different memory banks. For an example, see *Banked code*, page 41.

Data segments

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

To get a clear understanding about how the data segments work, you must be familiar with the different memory types and the different data models available in the IAR C/C++ Compiler for HCS12. If you need to refresh these details, see the chapter *Data storage*.

STATIC MEMORY SEGMENTS

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

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

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

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

Segment naming

The names of the segments consist of two parts—the segment group name and a *suffix*—for instance, `DATA16_I`. There is a segment group for each memory type, where each segment in the group holds different categories of declared data. The names of the segment groups are derived from the memory type and the corresponding keyword, for example `DATA16` and `__data16`. The following table summarizes the memory types and the corresponding segment groups:

Memory type	Segment group	Memory range
Data8	`DATA8`	`0x00-0xFF`
Data16	`DATA16`	`0x0000-0xFFFF`

Table 8: Memory types with corresponding segment groups

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

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

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

Table 9: Segment name suffixes

For a summary of all supported segments, see *Summary of segments*, page 135.

Examples

Assume the following examples:

```
__data16 int j;
__data16 int i = 0;
```

The data16 variables that are to be initialized to zero when the system starts will be placed in the segment `DATA16_Z`.

`__no_init __data16 int j;`	The data16 non-initialized variables will be placed in the segment `DATA16_N`.
`__data16 int j = 4;`	The data16 non-zero initialized variables will be placed in the segment `DATA16_I`, and initializer data in segment `DATA16_ID`.

Initialized data

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

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

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

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

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

  The `ID` segment can, for all segment groups, be placed anywhere within the visible 64-Kbyte memory area. Note that the gap between the ranges will also be copied.

  For more information about how to place data initializers in banked memory, see *Banked data initializers*, page 117.

- 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 RAM are defined:
-Z(DATA)DATA8_Z,DATA8_I=00-FF
-Z(DATA)DATA16_Z,DATA16_I=1000-3FFF

//Then, the ROM data segments are placed in memory:
//-Z(CONST)DATA8_C=00-FF
-Z(CONST)DATA16_C,DATA8_ID,DATA16_ID=C000-FFFF
```

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

THE STACK

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

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

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

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

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

Stack size allocation from the command line

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

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

```
-D_CSTACK_SIZE=100
```

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=1000-3FFF
```

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

Stack size and placement 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 program failure. Either variable storage will be overwritten, leading to undefined behavior, or the stack will fall outside of the memory area, also leading to undefined behavior. Because the second alternative might be easier to detect, you should consider placing your stack so that there is unwritable memory at the first location outside the allocated stack.

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

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

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=100
```

Placement of heap segment

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

```
-Z(DATA)HEAP+HEAP_SIZE=1000-3FFF
```

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

Heap size and standard I/O

If you have excluded `FILE` descriptors from the DLIB runtime environment, like in the normal configuration, there are no input and output buffers at all. Otherwise, like in the full configuration, be aware that the size of the input and output buffers is set to 512 bytes in the `stdio` library header file. If the heap is too small, I/O will not be buffered, which is considerably slower than when I/O is buffered. If you execute the application using the simulator driver of the IAR C-SPY Debugger, you are not likely to notice the speed penalty, but it is quite noticeable when the application runs on an HCS12 microcontroller. If you use the standard I/O library, you should set the heap size to a value which accommodates the needs of the standard I/O buffer.

LOCATED DATA

A variable that has been explicitly placed at an address, for example by using the compiler `@` syntax, will be placed in either the `DATA16_AC` or the `DATA16_AN` segment. The former is used for constant-initialized data, and the latter for items declared as `__no_init`. The individual segment part of the segment knows its location in the memory space, and it does not have to be specified in the linker command file.

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

Code segments

This section contains descriptions of the segments used for storing code, and the interrupt vector table. For a complete list of all segments, see *Summary of segments*, page 135.

NON-BANKED CODE

Non-banked code—that is, all user-written code when you use the Normal code model, `__interrupt` declared functions, and some assembler run-time library routines—is placed in the `CODE` segment.

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

```
-Z(CODE)CODE=C000-FFFD
```

BANKED CODE

When you use the Banked code model, all user-written code is located in the `BANKED_CODE` segment. Here, the `-P` linker directive is used for distributing the banked code across the different code banks. This is useful here, because the memory range is non-consecutive.

In the linker command file it can look like this:

```
-P(CODE)BANKED_CODE=[_CODEBANK_START-_CODEBANK_END]
                    *_CODEBANK_BANKS+10000
```

Note the use of the linker defines `_CODEBANK_START`, `_CODEBANK_END`, and `_CODEBANK_BANKS` to define the bank start address, bank end address, and the number of banks, respectively. These defines could, for example, look like this:

```
-D_CODEBANK_START=8000
-D_CODEBANK_END=BFFF
-D_CODEBANK_BANKS=6
```

INTERRUPT VECTORS

The interrupt vector table contains pointers to interrupt routines, including the reset routine. The table is placed in the absolute segment `INTVEC`. As the segment is an absolute segment, it should not be defined in the linker command file.

The `__interrupt` declared functions are located in the code segment `CODE`, see *Non-banked code*, page 41.

C++ dynamic initialization

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

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

For example:

```
-Z(CONST)DIFUNCT=C000-FFFD
```

For additional information, see *DIFUNCT*, page 142.

Efficient usage of segments and memory

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

CONTROLLING DATA AND FUNCTION PLACEMENT

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

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

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

Note: Variables and functions can also be placed into named segments using the `--segment` option.

Data placement at an absolute location

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

Example

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

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

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

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

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

Data placement into named segments

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

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

Example

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

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

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

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

Function placement into named segments

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

Example

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

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

Declaring located variables extern

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

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

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

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

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

CREATING USER-DEFINED SEGMENTS

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

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

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

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

Verifying the linked result of code and data placement

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

SEGMENT TOO LONG ERRORS AND RANGE ERRORS

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

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

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

LINKER MAP FILE

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

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

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

Normally, XLINK will not generate an output file if there are any errors, such as range errors, during the linking process. Use the option **Range checks disabled** in the Embedded Workbench, 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 demands of an embedded application on the runtime environment depend both on the application itself and on the target hardware. The IAR DLIB runtime environment can be used as is together with the IAR C-SPY Debugger. However, to be able to run the application on hardware, you must adapt the runtime environment.

This section gives an overview of:

- The runtime environment and its components
- Library selection.

RUNTIME ENVIRONMENT FUNCTIONALITY

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

The runtime library is delivered both as prebuilt libraries and as source files, and you can find them in the product subdirectories `hcs12\lib` and `hcs12\src`, respectively.

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

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

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

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

LIBRARY SELECTION

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

The 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.s12`, with your own customized versions
- Choosing the level of support for certain standard library functionality, for example, locale, file descriptors, and multibytes, by choosing a *library configuration*: normal or full.

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

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

SITUATIONS THAT REQUIRE LIBRARY BUILDING

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

You must build your own library when:

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

For information about how to build a customized library, see *Building and using a customized library*, page 56.

LIBRARY CONFIGURATIONS

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

The following DLIB library configurations are available:

Library configuration	Description
Normal DLIB	No locale interface, C locale, no file descriptor support, no multibyte characters in `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 56.

The prebuilt libraries are based on the default configurations, see Table 12, *Prebuilt libraries*, page 51. There is also a ready-made library project template that you can use if you want to rebuild the runtime library.

DEBUG SUPPORT IN THE RUNTIME LIBRARY

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

The following table describes the different levels of debugging support:

Debugging support	Linker option in IAR Embedded Workbench	Linker command line option	Description
Basic debugging	Debug information for C-SPY	`-Fubrof`	Debug support for C-SPY without any runtime support
Runtime debugging	With runtime control modules	`-r`	The same as `-Fubrof`, but also includes debugger support for handling program abort, exit, and assertions.
I/O debugging	With I/O emulation modules	`-rt`	The same as `-r`, but also includes debugger support for I/O handling, which means that `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 69.

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

Using a prebuilt library

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

- Code model
- Size of `double` floating-point
- Library configuration—Normal or Full.

For the IAR C/C++ Compiler for HCS12, this means there is a prebuilt runtime library for each combination of these options. The following table shows the mapping of the library file, code models, `double` size, and library configurations:

Library	Code model	Size of double	Library configuration
`dlhcs12bdn.r12`	Banked	64 bits	Normal
`dlhcs12bdf.r12`	Banked	64 bits	Full
`dlhcs12bfn.r12`	Banked	32 bits	Normal
`dlhcs12bff.r12`	Banked	32 bits	Full
`dlhcs12ndn.r12`	Normal	64 bits	Normal
`dlhcs12ndf.r12`	Normal	64 bits	Full
`dlhcs12nfn.r12`	Normal	32 bits	Normal
`dlhcs12nff.r12`	Normal	32 bits	Full

Table 12: Prebuilt libraries

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

`<type><target><code_model><size_of_double><library_configuration>.r12`

where

- `<type>` is `dl` for the IAR DLIB runtime environment
- `<cpu_variant>` is `hcs12`
- `<code_model>` is one of `n` or `b` for normal and banked code, respectively
- `<size_of_double>` is one of `f`, or `d`, for 32-bit doubles or 64-bit doubles, respectively
- `<library_configuraton>` 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 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.

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

- Specify which library object file to use on the XLINK command line, for instance:
  `dlhcs12bdn.r12`
- Specify the include paths for the compiler and assembler:
  `-I hcs12\inc`
- Specify the library configuration file for the compiler:
  `--dlib_config <install_dir>\dlhcs12bdn.h`

You can find the library object files and the library configuration files in the subdirectory `hcs12\lib`.

CUSTOMIZING A PREBUILT LIBRARY WITHOUT REBUILDING

The prebuilt libraries delivered with the IAR C/C++ Compiler for HCS12 can be used as is. However, it is possible to customize parts of a library without rebuilding it. There are two different methods:

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

The following items can be customized:

Items that can be customized	Described on page
Formatters for printf and scanf	*Choosing formatters for printf and scanf*, page 52
Startup and termination code	*System startup and termination*, page 58
Low-level input and output	*Standard streams for input and output*, page 61
File input and output	*File input and output*, page 64
Low-level environment functions	*Environment interaction*, page 67
Low-level signal functions	*Signal and raise*, page 68
Low-level time functions	*Time*, page 68
Size of heaps, stacks, and segments	*Data segments*, page 35

Table 13: Customizable items

For a description about how to override library modules, see *Overriding library modules*, page 54.

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 63.

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 (default)	_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
Approximate relative size	100%	90%	30%	10%

Table 14: Formatters for printf

*** Depends on which library configuration is used.**

For information about how to fine-tune the formatting capabilities even further, see *Configuration symbols for printf and scanf*, page 63.

Specifying the print formatter in the IAR Embedded Workbench

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

Specifying printf formatter from the command line

To use any other variant than the default (`_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 (default)	_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
Approximate relative size	100%	60%	50%

Table 15: Formatters for scanf

*** Depends on which library configuration that is used.**

For information about how to fine-tune the formatting capabilities even further, see *Configuration symbols for printf and scanf*, page 63.

Specifying scanf formatter in the IAR Embedded Workbench

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

Specifying scanf formatter from the command line

To use any other variant than the default (`_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 `hcs12\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

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

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

Overriding library modules from the command line

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

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

   ```
   icchcs12 library_module
   ```

   This creates a replacement object module file named *`library_module`*`.r12`.

   Note: The code model, size of `double`, include paths, and the library configuration file must be the same for *`library_module`* as for the rest of your code.
4. Add *`library_module`*`.r12` to the XLINK command line, either directly or by using an extended linker command file, for example:

   ```
   xlink library_module dlhcs12bdn.r12
   ```

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

   Run XLINK to rebuild your application.

This will use your version of *library_module*.r12, 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 48, 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 *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 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 49.

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

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

MODIFYING THE LIBRARY FUNCTIONALITY

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

The library functionality is determined by a set of *configuration symbols*. The default values of these symbols are defined in the Dlib_defaults.h file. This read-only file describes the configuration possibilities. In addition, your library has its own library configuration file dlhcs12Custom.h, which sets up that specific library with full library configuration. For more information, see Table 13, *Customizable items*, page 52.

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 `dlhcs12Custom.h` file 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 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 performs during startup and termination of applications. The following figure gives a graphical overview of the startup and exit sequences:

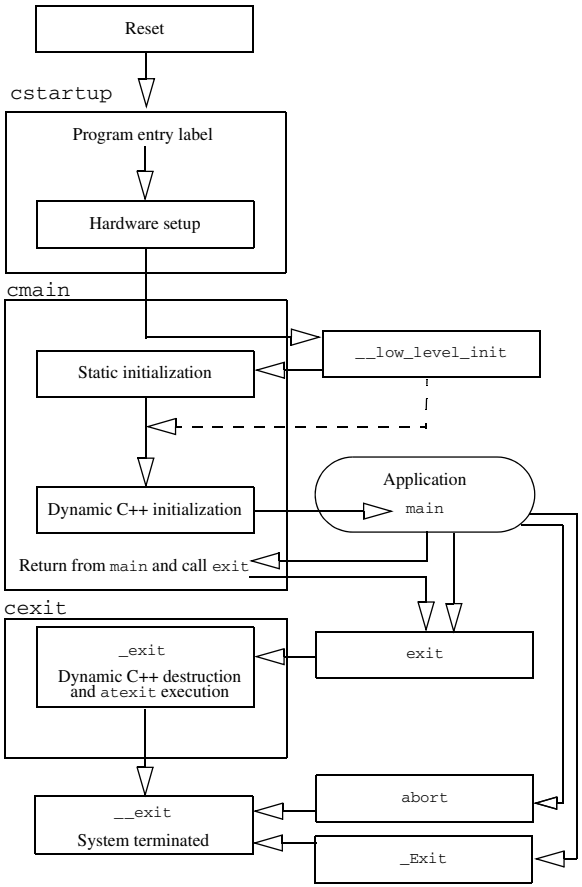


Figure 3: Startup and exit sequences

The code for handling startup and termination is located in the following source files:

Source files	Directory
cstartup.s12	hcs12\src\lib

Table 16: Code for startup and termination

Source files	Directory
cmain.s12	hcs12\src\lib\dlib
cexit.s12	hcs12\src\lib\dlib
low_level_init.c	hcs12\src\lib
low_level_init.s12	hcs12\src\lib

Table 16: Code for startup and termination (Continued)

SYSTEM STARTUP

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

- When the cpu is reset it will jump to the program entry label `__program_start` in the system startup code.
- The function `__low_level_init` is called, giving the application a chance to perform early initializations
- Static variables are initialized; this includes clearing zero-initialized memory and copying the ROM image of the RAM memory of the rest of the initialized variables depending on the return value of `__low_level_init`
- Static C++ objects are constructed
- The `main` function is called, which starts the application.

SYSTEM TERMINATION

An application can terminate normally in two different ways:

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

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

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

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

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

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

C-SPY interface to system termination

If your project is linked with the XLINK options **With runtime control modules** or **With I/O emulation modules**, the normal `__exit` and `abort` functions are replaced with special ones. C-SPY will then recognize when those functions are called and can take appropriate actions to simulate program termination. For more information, see *C-SPY Debugger runtime interface*, page 69.

Customizing system initialization

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

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

The code for handling system startup is located in the source files `cstartup.s12`, `low_level_init.c`, and `low_level_init.s12` located in the `hcs12\src` directory.

Note: Normally there is no need for customizing the `cmain.s12` file or the `cexit.s12` file.

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 56.

Note: Regardless of whether you modify the `__low_level_init` routine or the `cstartup.s12` file, you do not have to rebuild the library.

__LOW_LEVEL_INIT

Two skeleton low-level initialization files are supplied with the product—a C source file, `low_level_init.c` and an alternative assembler source file, `low_level_init.s12`. A prebuilt variant is included in the library. 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 CSTARTUP.S12 FILE

As noted earlier, you should not modify the `cstartup.s12` file if a customized version of `__low_level_init` is enough for your needs. However, if you do need to modify the `cstartup.s12` file, 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 54.

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 `hcs12\src` directory.

If you intend to rebuild the library, the source files are available in the template library project, see *Building and using a customized library*, page 56. 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 69.

Example of using __write and __read

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

```
__no_init volatile unsigned char LCD_IO @ address;

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

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

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

```
__no_init volatile unsigned char KB_IO @ 0xD2;

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

For information about the @operator, see *Controlling data and function placement*, page 42.

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 52.

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 `DLIB_Defaults.h` file.

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
`_DLIB_PRINTF_CHAR_BY_CHAR`	Output char by char or buffered

Table 17: Descriptions of printf configuration symbols

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

Scanf configuration symbols	Includes support for
`_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 18: Descriptions of scanf configuration symbols

CUSTOMIZING FORMATTING CAPABILITIES

To customize the formatting capabilities, you need to set up a library project, see *Building and using a customized library*, page 56. 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 49. 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 19: Low-level I/O files

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

Note: If you link your library with I/O debugging support, C-SPY variants of the low-level I/O functions will be linked for interaction with C-SPY. For more information, see *Debug support in the runtime library*, page 49.

Locale

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

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

The DLIB library can be used in two major modes:

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

LOCALE SUPPORT IN PREBUILT LIBRARIES

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

- All prebuilt libraries supports the C locale only
- 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 encoding during runtime.
- Libraries with *normal library configuration* do not have support for the locale interface.

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

CUSTOMIZING THE LOCALE SUPPORT

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

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

Locale configuration symbols

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

If you want to customize the locale support, you simply define the locale configuration symbols required by your application. For more information, see *Building and using a customized library*, page 56.

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

Building a library without support for locale interface

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

Building a library with support for locale interface

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

CHANGING LOCALES AT RUNTIME

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

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

`lang_REGION`

or

`lang_REGION.encoding`

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

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

Example

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

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

Environment interaction

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

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

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

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

```
key=value\0
```

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

For example:

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

If you need a more sophisticated environment variable handling, you should implement your own `getenv`, and possibly `putenv` function. This does not require that you rebuild the library. You can find source templates in the files `getenv.c` and `environ.c` in the `hcs12\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 54.

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 56.

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 49.

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 `hcs12\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 54.

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 56.

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 `hcs12\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 54.

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 56.

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 69.

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 56. 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 `hcs12\src` directory. For further information, see *Building and using a customized library*, page 56. To turn off assertions, you must define the symbol `NDEBUG`.

In the IAR Embedded Workbench IDE, the 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 49. 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 *
`__exit`	C-SPY notifies that the end of the application has been reached *
`__read`	`stdin`, `stdout`, and `stderr` will be directed to the Terminal I/O window; all other files will read the associated host file
`__write`	`stdin`, `stdout`, and `stderr` will be directed to the Terminal I/O window, all other files will write to the associated host file
`__open`	Opens a file on the host computer
`__close`	Closes the associated host file on the host computer
`__seek`	Seeks in the associated host file on the host computer
`remove`	Writes a message to the Debug Log window and returns -1
`rename`	Writes a message to the Debug Log window and returns -1
`time`	Returns the time on the host computer
`clock`	Returns the clock on the host computer
`system`	Writes a message to the Debug Log window and returns -1
`_ReportAssert`	Handles failed asserts *

Table 20: 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 49. 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.

Checking module consistency

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

When developing an application, it is important to ensure that incompatible modules are not used together. For example, in the IAR C/C++ Compiler for HCS12, it is possible to specify the size of the `double` floating-point type. If you write a routine that only works for 64-bit doubles, it is possible to check that the routine is not used in an application built using 32-bit doubles.

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

RUNTIME MODEL ATTRIBUTES

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

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

Example

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

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

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

Table 21: 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="__code_model", "normal"
```

For detailed syntax information, see *#pragma rtmodel*, page 187.

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

```
  RTMODEL "__code_model", "normal"
```

For detailed syntax information, see the *IAR Assembler Reference Guide for HCS12*.

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

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

PREDEFINED RUNTIME ATTRIBUTES

The table below shows the predefined runtime model attributes that are available for the IAR C/C++ Compiler for HCS12. These can be included in assembler code or in mixed C or C++ and assembler code.

Runtime model attribute	Value	Description
`__rt_version`	`1`	This runtime key is always present in all modules generated by the IAR C/C++ Compiler for HCS12. If a major change in the runtime characteristics occurs, the value of this key changes. For an example, see *Hints for using the Normal calling convention*, page 86.
`__code_model`	`normal` or `banked`	Corresponds to the code model used in the project.
`__double_size`	`32` or `64`	Corresponds to the code model used in the project.

Table 22: Predefined runtime model attributes

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

If you are using assembler routines in the C or C++ code, refer to the chapter *Assembler directives* in the *IAR Assembler Reference Guide for HCS12.*

Examples

For an example of using the runtime model attribute `__rt_version` for checking module consistency on used calling convention, see *Hints for using the Normal calling convention*, page 86.

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

Note that the compiler sets this attribute to `free`, unless the register is locked.

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

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

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

USER-DEFINED RUNTIME MODEL ATTRIBUTES

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

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

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

Implementation of system startup code

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

Note:

- Do not modify the `cstartup.s12` file unless required by your application. Your first option should be to use a customized version of `__low_level_init` for initialization code.
- Normally, there is no need for customizing the `cmain.s12` file or the `cexit.s12` file.

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

For information about assembler source files, see the *IAR Assembler Reference Guide for HCS12*.

MODULES AND SEGMENT PARTS

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

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

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

Segment parts, REQUIRE, and the falling-through trick

The system startup code has been designed to use segment parts so that as little as possible of unused code will be included in the linked application. The following trick is used in the files `cmain.s12` and `cexit.s12`, which are files you should not modify.

For example, every piece of code used for initializing one type of memory is stored in a segment part of its own. If a variable is stored in a certain memory type, the corresponding initialization code will be referenced by the code generated by the compiler, and included in your application. Should no variables of a certain type exist, the code is simply discarded.

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

The segment parts defined in the system startup code are guaranteed to be placed immediately after each other. XLINK will not change the order of the segment parts or modules, because the segments holding the system startup code are placed using the `-Z` option.

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

```
    MODULE  doSomething

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

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


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

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


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

?end_of_test:
    JUMP    (RA)   // This is included if ?do_something above
                   // is included.
    ENDMOD
```

Added C functionality

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

The following include files provide these features:

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

STDINT.H

This include file provides integer characteristics.

STDBOOL.H

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

MATH.H

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

STDIO.H

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

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

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

`__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 functions have been added:

`_exit`	Exits without closing files et cetera.
`__qsortbbl`	A `qsort` function that uses the bubble sort algorithm. Useful for applications that have limited stack.

PRINTF, SCANF AND STRTOD

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

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 HCS12 microcontroller that require precise timing and special instruction sequences.

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

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

Mixing C and assembler

The IAR C/C++ Compiler for HCS12 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 complete information, and can 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 and extended operators*.

MIXING C AND ASSEMBLER MODULES

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

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

The first issue is discussed in this section. The following two are covered in the section *Calling convention*, page 85.

The section on memory access methods 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 93.

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

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

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

The recommended method for mixing C or C++ and assembler modules is described in *Calling assembler routines from C*, page 82, and *Calling assembler routines from C++*, page 84, 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("MOV flag,PIND");
  }
}
```

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

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

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

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

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, 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**.

Use the following options to compile the skeleton code:

```
icchcs12 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 `s12`. Also remember to specify the code model you are using as well as a low level of optimization.

The result is the assembler source output file `skeleton.s12`.

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 by using the option `-lB` instead. Note that `CFI` information must be included in the source code to make the C-SPY Call Stack window work.

The output file

The output file contains the following important information:

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

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

Calling assembler routines from C++

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

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

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

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

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

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

```
class X;

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

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

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

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

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

Calling convention

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

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

The IAR C/C++ Compiler for HCS12 provides two calling conventions—Normal, which is used by default, and Simple. This section describes these calling conventions. The following items are looked upon:

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

CHOOSING A CALLING CONVENTION

There are two calling conventions to choose between:

- The Simple calling convention offers a simple assembler interface. It is compatible with the 68HC12 IAR Embedded Workbench IDE version 2.x. This calling convention is recommended for use with assembler code as it will remain valid over time
- The Normal calling convention is default. It is more efficient than the Simple calling convention, but also more complex to understand and can be subject to change in later versions of the compiler.

The Normal calling convention is used by default. For individual functions, you can override the default calling convention by using the `__simple` function attribute, for example:

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

Hints for using the Normal calling convention

The Normal calling convention is very complex and if you intend to use it for your assembler routines, you should create a list file and see how the compiler assigns the different parameters to the available registers. For an example, see *Creating skeleton code*, page 82.

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

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

The parameter *value* should have the same value as used internally by the compiler. For information about what value to use, see the generated list file. If the calling convention changes in future compiler versions, the runtime model value used internally by the compiler will also change. Using this method gives a module consistency check as the linker will produce an error if there is a mismatch between the values.

For more information about checking module consistency, see *Checking module consistency*, page 70.

FUNCTION DECLARATIONS

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

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

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

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

C AND C++ LINKAGE

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

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

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

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

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

   int f(int);

#ifdef __cplusplus
}
#endif
```

PRESERVED VERSUS SCRATCH REGISTERS

The general HCS12 CPU registers are divided into three separate sets, which are described in this section.

Scratch registers

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

The register `D`, as well as the `CCR` status register, can always be used as a scratch register by the function. In the Normal calling convention, the `Y` register is also a scratch register.

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 register `X` must always be a preserved register. In the Simple calling convention, the `Y` register must also be preserved.

Note: For a `__task` declared function, no registers are preserved.

Special registers

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

- 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.

- A `__monitor` declared function will push the `CCR` status register on the stack and disable interrupts using the `SEI` instruction. Previous interrupt status will be restored just prior to function return.

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

Hidden parameters

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

If the function returns a structure or a 64-bit scalar (`long long` or 64-bit `double`), the memory location where the return value is to be stored is passed in a register:

- In the Simple calling convention, the hidden parameter is added first in the parameter list, which means the parameter will be passed in the register `D`
- In the Normal calling convention, the hidden parameter will be added last in the parameter list.

Register parameters

In the Normal calling convention the registers available for passing parameters are `A`, `B`, `D`, and `Y`. In the Simple calling convention, the register `D` is available for passing parameters.

Parameters	Passed in registers Normal calling convention	Passed in registers Simple calling convention
8-bit values	`A`, `B`	`A`, `B`
16-bit values	`D`, `Y`	`D`
24-bit values	`B:Y`	`--`
32-bit values	`D:Y`	`--`

Table 23: Registers used for passing parameters

In the Simple calling convention, the assignment of registers to parameters is a straightforward process. Traversing the parameters from left to right, only the first parameter is assigned to the available register or registers. Should there be no more available registers, the parameter and any additional parameters are passed on the stack. In case the first parameter is not placed in register, none of any remaining parameters will be placed in register, even though there should be free registers available.

In the Normal calling convention, the assignment of registers to parameters is a complex process. Traversing the parameters from left to right, suitable parameters will be picked from any position in the parameter list.

The following table shows some of the possible combinations, and in the listed examples, `b` denotes an 8-bit data type, `w` denotes a 16-bit data type, `d` denotes a 24-bit or 32-bit parameter, `*` denotes a pointer type.

Parameters *	Parameter 1	Parameter 2	Parameter 3	Assignment
`f1(w1,w2)` 16-bit parameters but no 8-bit parameters	`D`	`Y`	`--`	First parameter in `D`, if there is a second parameter, it will be placed in `Y`.
`f2(w1,b1,w2)` At least two 16-bit parameters and at most one 8-bit parameter	`D`	stack	`Y`	First 16-bit parameter in `D`, the second parameter in `Y`, the 8-bit parameter on the stack.
`f3(* w1,b1,w2)` At least two 16-bit parameters where the first is a pointer type, and at most one 8-bit parameter	`Y`	stack	`D`	First 16-bit pointer type parameter in Y, the second 16-bit parameter in D, the 8-bit pointer on the stack.
`f4(b1,b2)` 8-bit parameters but no 16-bit parameters	`B`	`A`	`--`	The first parameter in B, the second, if any, in A.
`f5(b1,b2,w1)`	`B`	`A`	`Y`	First 8-bit parameter in `B`, the second, if any, in `A`. Any 16-bit parameter in `Y`.
`f6(d1,d2)` Any mix of 24-bit and 32-bit parameters	`B:Y` or `D:Y`	stack	`--`	The first parameter in `B:Y` or `D:Y` respectively, and the second parameter on the stack.

Table 24: Assignment of register parameters

Table 24, *Assignment of register parameters* only lists some examples of combinations of register assignment. However, if you intend to use the Normal calling convention, you should create a list file and see how the compiler assigns the different parameters to the available registers. For an example, see *Creating skeleton code*, page 82.

Stack layout

Stack parameters—any parameters not passed in register—are stored in the main memory, starting at the location pointed to by the stack pointer. Below the stack pointer, which grows towards low addresses, 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, and so on.

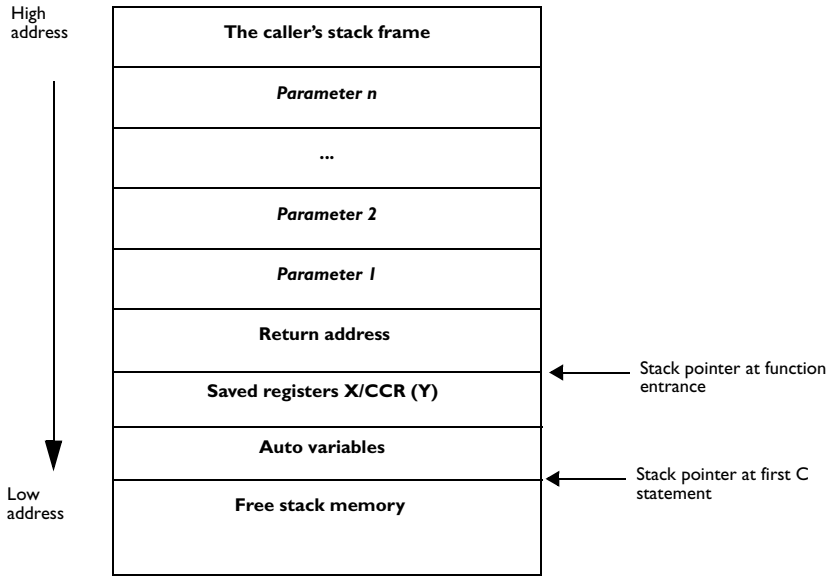


Figure 4: Storing stack parameters in memory

FUNCTION EXIT

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

The return value of a function, if any, can be scalar (such as integers and pointers), floating-point, or a structure.

Registers used for returning values

The registers available for returning values are:

- In the Normal calling convention: B, D, Y, BY, and DY
- In the Simple calling convention: B, D, BY, and DY

Return values	Returned in registers Normal calling convention	Returned in registers Simple calling convention
8-bit values	B	B
16-bit pointer	Y	D
16-bit integral type	D	D
24-bit values	BY	BY
32-bit values	DY	DY
64-bit values and non-scalars	Returned as a hidden pointer in register Y that is passed as the last argument	Returned as a hidden pointer in register D that is passed as the last argument

Table 25: Registers used for returning values

Stack layout

It is the responsibility of the caller to clean the stack from parameters after the called function has returned. This is done by adjusting the stack pointer, either by using the LEAS instruction (2-byte operation code), or by using a PUL instruction (1-byte operation code). The stack cleaning can be delayed by the caller, which means several cleanups can be combined. After function exit, the stack pointer must point at the same location as at the function entrance.

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.

A function returns by using one of the following instructions:

- For non_banked functions, the instruction RTS is used, which corresponds to the JSR instruction
- For banked functions, the instruction RTC is used, which corresponds to the CALL instruction
- For interrupt functions, the instruction RTI is used.

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 D, and the return value is passed back to its caller in the register D.

The following assembler routine is compatible with the declaration for a function in non-banked memory; it will return a value that is one number higher than the value of its parameter:

```
ADDD      #1
RTS
```

Example 2

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

```
struct a_struct { int k; int l; };
int a_function(struct a_struct m, int n);
```

The calling function must reserve four bytes on the top of the stack and copy the contents of the `struct` to that location. In the Normal calling convention, the integer parameter `n` is passed in the register D. In the Simple calling convention, the parameter is passed on the stack as the previous parameter is not passed via any register. The return value is passed back to its caller in the register D.

Example 3

The function below will return a `struct`.

```
struct a_struct { int k; };
struct a_struct  a_function(int m);
```

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 Y. The caller assumes that this register remains untouched. The parameter *x* is passed in D.

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

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

In this case, the return value is a scalar, so there is no hidden parameter. The parameter `m` is passed in D, and the return value is returned in Y.

Call frame information

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

If you want the call stack to be available when debugging a routine written in assembler language, you must supply equivalent debug information in your assembler source using the assembler directive CFI. This directive is described in detail in the IAR Assembler Reference Guide for HCS12.

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
A, B, X, Y, D	Normal registers
SP	The stack pointer
CCR	The status register—special processor registers
?RET	The return address

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

Using C++

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

Overview

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

STANDARD EMBEDDED C++

The following C++ features are supported:

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

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

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

- Templates
- Multiple inheritance
- Exception handling
- Runtime type information

- New cast syntax (the operators `dynamic_cast`, `static_cast`, `reinterpret_cast`, and `const_cast`)
- Namespaces.

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

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

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

To support Extended EC++, this product includes a version of the standard template library (STL), in other words, the C++ standard chapters utilities, containers, iterators, algorithms, and some numerics. This STL 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 IAR C/C++ Compiler for HCS12, the default language is C. To be able to compile files written in Embedded C++, you must use the `--ec++` compiler option. See *--ec++*, page 156. You must also use the IAR DLIB runtime library.

To take advantage of *Extended* Embedded C++ features in your source code, you must use the `--eec++` compiler option. See *--eec++*, page 157.

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

Feature descriptions

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

CLASSES

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

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

For further information about attributes, see *Type and object attributes*, page 131.

Example

```
class A {
  public:
   static __data8 int i @ 60;   //Located in data8 at address 60
   static __banked void f();    //Located in banked memory
   __banked void g();           //Located in banked memory
   virtual __banked void h();   //Located in banked memory
};
```

The `this` pointer used for referring to a class object will by default have the data memory attribute for the default data pointer type. This means that such a class object can only be defined to reside in memory from which pointers can be implicitly casted to a default data pointer.

Example

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

Class memory

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

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

Example

```
class __data8 C {
  public:
    void f();       // Has a this pointer of type C __data8 *
    void f() const; // Has a this pointer of type
                    // C __data8 const *
    C();            // Has a this pointer pointing into data8
                    // memory
    C(C const &);  // Takes a parameter of type C __data8 const&
                    // (also true of generated copy constructor)
    int i;
};
C Ca;               // Resides in data8 memory instead of the
                    // default memory
C __data16 Cb;      // Restricted usage; __data16 pointer can't
                    // be implicitly cast into a __data8
                    // pointer
void h()
{
  C Cc;             // Resides on the stack; restricted usage due
                    // to data8 this pointer
}
C * Cp;             // Creates a pointer to data8 memory
C __data16 * Cp;    // Creates a pointer to data16 memory;
                    // restricted usage due to data8 this pointer
```

Note: Whenever a class type associated with a class memory type, like `C`, must be declared, the class memory type must be mentioned as well:

```
class __data16 C;
```

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

There is a built-in operator that returns the class memory type associated with a class, `__memory_of(`*`class`*`)`. For instance, `__memory_of(C)` returns `__data8`.

When inheriting, the rule is that it must be possible to convert implicitly a pointer to a subclass into a pointer to its base class. This means that a subclass can have a *more* restrictive class memory than its base class, but not a *less* restrictive class memory.

```
class __data8 D : public C { // OK, same class memory
  public:
    void g();
    int j;
};

class __data16 E : public C { // Not OK, data16 memory isn't
                              // inside data8 memory
  public:
    void g();
    int j;
};

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

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

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

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

For more information about memory types, see *Memory types*, page 12.

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);         // An 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.

Templates and data memory attributes

For data memory attributes to work as expected in templates, two elements of the standard C++ template handling have been changed—class template partial specialization matching and function template parameter deduction.

In Extended Embedded C++, the class template partial specialization matching algorithm works like this:

> When a pointer or reference type is matched against a pointer or reference to a template parameter type, the template parameter type will be the type pointed to, stripped of any data memory attributes, if the resulting pointer or reference type is the same.

Example

```
// data16 is the memory type of the default pointer.
template<typename> class Z;
template<typename T> class Z<T *>;

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

In Extended Embedded C++, the function template parameter deduction algorithm works like this:

When function template matching is performed and an argument is used for the deduction; if that argument is a pointer to a memory that can be implicitly converted to a default pointer, do the parameter deduction as if it was a default pointer.

When an argument is matched against a reference, do the deduction as if the argument and the parameter were both pointers.

Example

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

fun((int __data16 *) 0); // T = int
fun((int          *) 0); // T = int
fun((int __data8  *) 0); // T = int
```

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

For templates that are matched using this modified algorithm, it is impossible to get automatic generation of special code for pointers to *small* memory types. For *large* and "other" memory types (memory that cannot be pointed to by a default pointer) it is possible. In order to make it possible to write templates that are fully memory-aware—in the rare cases where this is useful—use the `#pragma basic_template_matching` directive in front of the template function declaration. That template function will then match without the modifications described above.

Example

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

fun((int __data16 *) 0); // T = int __data16
```

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 96.

VARIANTS OF CASTS

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

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

MUTABLE

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

NAMESPACE

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

THE STD NAMESPACE

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

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

POINTER TO MEMBER FUNCTIONS

A pointer to a member function can only contain a default function pointer, or a function pointer that can implicitly be casted to a default function pointer. To use a pointer to a member function, make sure that all functions that should be pointed to reside in the default memory or a memory contained in the default memory.

Example

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

USING INTERRUPTS AND EC++ DESTRUCTORS

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

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

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

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

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

To avoid interrupts, place a call to the intrinsic function `__disable_interrupts()` before the call to `_exit()`.

Efficient coding for embedded applications

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

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

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

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

Taking advantage of the compilation system

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

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

CONTROLLING COMPILER OPTIMIZATIONS

The IAR C/C++ Compiler for HCS12 allows you to specify whether generated code should be optimized for size or for speed, at a selectable optimization level. The purpose of optimization is to reduce the code size and to improve the execution speed. When only one of these two goals can be reached, the compiler prioritizes according to the settings you specify. Note that one optimization sometimes enables other optimizations to be performed, and an application may become smaller even when optimizing for speed rather than size.

The following table describes the optimization levels:

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

Table 27: Compiler optimization levels

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

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

Both compiler options and pragma directives are available for specifying the preferred type and level of optimization. The chapter *Compiler options* contains reference information about the command line options used for specifying optimization type and level. Refer to the *IAR Embedded Workbench® IDE User Guide* for information about the compiler options available in the IAR Embedded Workbench. Refer to *#pragma optimize*, page 185, for information about the pragma directives that can be used for specifying optimization type and level.

FINE-TUNING ENABLED TRANSFORMATIONS

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

- Common subexpression elimination
- Loop unrolling
- Function inlining
- Code motion
- Type-based alias analysis
- Cross call.

Common subexpression elimination

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

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

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

Loop unrolling

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

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

The compiler heuristically decides which loops to unroll. Different heuristics are used when optimizing for speed and size.

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

To read more about the command line option, see *--no_unroll*, page 164.

Function inlining

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

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

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

To read more about the command line option, see *--no_inline*, page 163.

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**.

To read more about the command line option, see *--no_code_motion*, page 162.

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 163.

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. Thereby creating a non-conforming application.

Cross call

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

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

To read more about related command line options, see *--no_cross_call*, page 162 and *--cross_call_passes*, page 151.

Selecting data types and placing data in memory

For efficient treatment of data, you should consider the data types used and the most efficient placement of the variables.

USING EFFICIENT DATA TYPES

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

- Use small and unsigned data types, (`unsigned char` and `unsigned short`) unless your application really requires signed values.
- Try to avoid 64-bit data types, such as `double` and `long long`.
- Bitfields with sizes other than 1 bit should be avoided because they will result in less efficient code compared to bit and byte operations.
- When using arrays, it is more efficient if the type of the index expression matches the index type of the memory of the array. For data16 this is `int` and for data8 this is `signed char`.

- 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.
- Pointer arithmetics is best performed in 16-bit mode. Pointer expressions declared `__data8` can result in less efficient code compared to `__data16` pointer expressions.

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 IAR C/C++ Compiler for HCS12 supports two floating-point formats—32 and 64 bits. The 32-bit floating-point type `float` is more efficient in terms of code size and execution speed. However, the 64-bit format `double` supports higher precision and larger numbers.

In the IAR C/C++ Compiler for HCS12, the floating-point type `float` always uses the 32-bit format. The format used by the `double` floating-point type depends on the setting of the double size compiler option `--double={32|64}`.

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.

If you are using 64-bit doubles—that is, using the option `--double=64`—innocent-looking expressions with `float` variables can be evaluated in `double` precision. In the example below `a` is converted from a `float` to a `double`, `1` is added and the result is converted back to a `float`:

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

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

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

DATA MEMORY ATTRIBUTES

The IAR C/C++ Compiler for HCS12 provides two different data memory type attributes—`__data8` and `__data16`. Efficient usage of memory type attributes can significantly reduce the application size.

For most applications it is sufficient to use the default memory for the data objects. However, for individual objects it might be necessary to specify the `__data8` memory attributes in certain cases, for example:

- An application where some global variables are accessed in a large number of locations. In this case they can be declared to be placed in data8 memory
- Data that must be placed at a specific memory location.

For details about the memory types, see *Memory types*, page 12.

ANONYMOUS STRUCTS AND UNIONS

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

Anonymous structures are part of the C++ language; however, they are not part of the C standard. In the IAR C/C++ Compiler for HCS12 they can be used in C if language extensions are enabled.

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

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

Example

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

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

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

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

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

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

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

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

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

Writing efficient code

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

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

- Local variables—auto variables and parameters—are preferred over static or global variables. The reason is that the optimizer must assume, for example, that called functions may modify non-local variables. When the life spans for local variables end, the previously occupied memory can then be reused. Globally declared variables will occupy data memory during the whole program execution. Read about stack considerations in section *Stack size and placement considerations*, page 39.
- 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 163.
- 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 79.

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 parameters, such as structures, as parameters or as 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;
{
  .....
}
```

CODE MODEL AND FUNCTION MEMORY ATTRIBUTES

For most applications it is sufficient to use the code model feature to specify the default memory for functions. However, for individual functions it might be necessary to specify other memory attributes in certain cases, for example:

- Functions that have speed requirements should be placed in the fastest memory
- For functions part of an external interface that dictates a certain memory.

INTEGER TYPES AND BIT NEGATION

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

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

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

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

PROTECTING SIMULTANEOUSLY ACCESSED VARIABLES

Variables that are accessed from multiple threads, for example from `main` or an interrupt, must be properly marked and have adequate protection, the only exception to this is a variable that is always *read-only*.

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

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

ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for a number of HCS12 derivatives are included in the IAR C/C++ Compiler for HCS12 delivery. The header files are named `io`*`chip`*`.h` and define the processor-specific special function registers (SFRs).

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

SFRs with bitfields are declared in the header file. The following example is from `r9S12x512.h`:

```
__no_init volatile union
{
  unsigned char INITRM;
  struct
  {
    unsigned char RAMHAL:     1;
    unsigned char       :     2;
    unsigned char RAM11 :     1;
    unsigned char RAM12 :     2;
    unsigned char RAM13 :     1;
    unsigned char RAM14 :     1;
    unsigned char RAM15 :     1;
  } INITRM_bit;
} @ 0x0010;
```

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

```
// whole register access
INITRM = 0x1234;

// Bitfield accesses
INITRM_bit.RAM11 = 1;
INITRM_bit.RAM14 = 3;
```

You can also use the header files as templates when you create new header files for other HCS12 derivatives. For details about the @ operator, see *Located data*, page 40.

NON-INITIALIZED VARIABLES

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

The compiler supports the declaration of variables that will not be initialized, using the `__no_init` type modifier. They can be specified either as a keyword or using the `#pragma object_attribute` directive. The compiler places such variables in separate segments, according to the specified memory keyword. See the chapter *Placing code and data* for more information.

For `__no_init`, the `const` keyword implies that an object is read-only, rather than that the object is stored in read-only memory. It is not possible to give a `__no_init` object an initial value.

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

For information about the `__no_init` keyword, see page 177. Note that to use this keyword, language extensions must be enabled; see *-e*, page 156. For information about the `#pragma object_attribute`, see page 185.

BANKED DATA INITIALIZERS

If you use the banked code model, you can also place data initializers in banked memory. To achieve this, make sure to place the following segments in the same bank:

- The `*_ID` segments; segments that hold data initializers
- The `INITTAB` segment; holds information about addresses and sizes of segments to be initialized at system startup
- The `BANKED_CODE_SEGMENT_INIT` segment; holds the function that performs the copying of data initializers from the `*_ID` segments to their corresponding `*_I` segments at system startup.

In the linker command file, this could, for example look like this:

```
-Z(CODE)BANKED_CODE_SEGMENT_INIT,INITTAB,DATA16_ID=18000-1BFFF
```

This example places the segments in bank 1, which is assumed to be located in the `8000-BFFF` bank window.

Banked constant data

To save non-banked memory, it can be useful to place data constants in banked memory. To achieve this, you should:

- Define your constant data and place it in a separate segment to be placed in banked memory, see *Defining constant data for banked memory*, page 118
- Define the function that will access the banked data—the access function—so that the bank mechanism is considered.

Typically, for a banked application you use the banked code model, which means your functions will be located in banked memory. The bank mechanism can only have one bank window visible at a time. This means that the bank mechanism will not work if the banked constant data is located in a different bank compared to the function that will access the constant data. There are two ways to solve this problem, you must make sure that any function that will access the constant data is located either in the same bank as the constants—see *Placing function in same bank as banked data*, page 119—or in non-banked memory—see *Placing function in non-banked memory*, page 120. Which solution you choose depends on your requirements.

DEFINING CONSTANT DATA FOR BANKED MEMORY

The `#pragma location` directive is used for placing the constants in a separate segment, which you should locate to a banked address area.

This example defines your constant data:

```
#pragma location="MY_BANKED_CONSTANTS"
const char banked_data[] = {1,2,3,4,5};
```

Converting a 16-bit address to a banked address

At compile time, the banked constant data will have 16-bit addresses. However, the bank mechanism requires a 24-bit address. To solve this problem, the compiler provides an intrinsic function `__address_24_of`. It takes an address of a 16-bit object as its argument and returns the banked address as an unsigned long. The lower 16-bits, of the returned address, is the address in the 64-Kbyte bank window while bit 16–23 is the bank number to put in the bank switch register.

At link time, the objects will have 24-bit addresses if you make sure to locate the segment in banked memory, that is, memory with 24-bit addresses. This is achieved by editing the linker command file.

PLACING FUNCTION IN SAME BANK AS BANKED DATA

If you place the access function and the banked constant data in the same bank, the code bank register `PPAGE` will automatically be handled by the compiler leading to slightly smaller code size and no non-banked memory will be occupied for the function. However, the drawbacks are that all banked data to be accessed by the function must reside within the same bank as the function itself, limiting the amount of data accessible by the function. You will also get a slightly more complicated linker command file. Compare these pros and cons with the pros and cons for having the access function in non-banked memory, see *Placing function in non-banked memory*, page 120.

Note: The access function must be careful not to pass a pointer, which points to a banked constant object, to a function located in another bank.

The access function in banked memory

When you define an access function to be placed in banked memory, you must:

- Use the `#pragma location` directive to place the access function in a separate segment
- In the linker command file, define the segment to a location in the banked memory area. You must also make sure the banked function is placed in the same bank as the constant data.

For a banked application, you typically use the Banked code model, which means the compiler will *automatically* call functions via a banked function call.

This example defines an access function to be placed in banked memory:

```
#pragma location="MY_BANKED_CODE"
void memcpy_from_bank (char* dest,
                       unsigned int count,
                       char* address)
{
  /* Copy bytes from code bank to ram */
  while (count--)
    *dest++ = *address++;
}
```

Note that this example assumes Banked code model, otherwise the `__banked` keyword is needed on the function.

The function call

In this example, the intrinsic function `__address_24_of` is used when accessing the banked constant data. However, because the function that accesses the constant data and the constant data itself are both located in the same bank, a 24-bit address is not required. Thus, the parameter `banked_data` is converted to a `char*`, in other words a pointer to a 16-bit address.

```
/* Make __address_24_of intrinsic function visible for the
   compiler */
#include <intrinsics.h>

void main(void)
{
  /* Make a buffer on the stack */
  char str[16];

  /* Calling the access function - copying from banked code
     memory to RAM */
  memcpy_from_bank(str, 5,
                   (char*)(int) __address24_of(banked_data));
}
```

If you do not use the `__address_24_of` function in this case, the linker will issue a range error. The reason for the explicit `int` cast is to restrain a compiler warning.

In the linker command file

Because the access function and the constant data itself must both be located in the same bank, you must explicitly define the segments to the same banked memory area. This means that you must add a segment control directive to the linker command file with the definitions of your banks, for example:

```
-P(CODE)MY_BANKED_CODE,MY_BANKED_CONSTANTS=3A8000-3ABFFF
```

PLACING FUNCTION IN NON-BANKED MEMORY

If you place the access function in non-banked memory, the same function can be used for accessing data in different banks and the data is not required to be placed in the same bank as the function accessing the data. However, the drawbacks are that you must explicitly save and restore the `PPAGE` register leading to slightly bigger code size, and non-banked memory will be occupied for storing the function. Compare these pros and cons with the pros and cons for having the access function and the constant data in the same bank, see *Placing function in same bank as banked data*, page 119.

Note: The access function must be careful not to pass a pointer, which points to a banked constant object, to a function located in another bank.

The access function in non-banked memory

When you define a function to be placed in non-banked memory, you must:

- Use the `__non_banked` keyword to make the function accessible via a non-banked function call
- Save and restore the `PPAGE` register.

This example defines an access function to be placed in non-banked memory:

```
/* Make the PPAGE register visible to the compiler */
#include <hcs12a4.h>

/* Make __address_24_of intrinsic function visible for the
   compiler */
#include <intrinsics.h>

__non_banked void memcpy_from_bank (char* dest,
                                    unsigned int count,
                                    unsigned long address)
{
  /* Save current code banke page */
  char save = PPAGE;

  /* Make the code bank that contains the desired data constants
     visible in the 64K bank window */
  PPAGE = address >> 16;

  /* Create a char pointer to the object to copy from */
  char* p = (char*) (int) address;

  /* Copy bytes from code bank to RAM */
  while (count--)
    *dest++ = *p++;

  /* Restore previous code bank */
  PPAGE = save;
}
```

The function call

Also in this example, the intrinsic function `__address_24_of` is used when accessing the banked constant data. Because the access function is located in non-banked memory and the constant data itself is located in banked memory, the parameter—the address of the banked constant data—must be the complete banked address containing all 24 bits.

```
/* Make __address_24_of intrinsic function visible for the
   compiler */
#include <intrinsics.h
```

```
void main(void)
{
  /* Make a buffer on the stack */
  char str[16];

  /* Calling the access function - copying from banked code
     memory to RAM */
  memcpy_from_bank(str, 5, __address24_of(banked_data));
}
```

In the linker command file

Because a non-banked access function is not dependent on in which bank the banked constant data is located, the banked constant data can be located in any arbitrary bank by the linker. This means that you simply have to extend the segment control directive in the linker command file with the name of your segment holding the constant data.

The default segment control directive looks like this:

```
-P(CODE)BANKED_CODE=[8000-BFFF]*7+10000
```

Extend this line with the name of your segments for constant data, for example:

```
-P(CODE)BANKED_CODE,MY_BANKED_CONSTANTS=[8000-BFFF]*7+10000
```

Your non-banked function will automatically be placed in the segment `CODE`, which is located in non-banked memory.

Banked variable data (data in RAM)

In general, you can utilize banked RAM data for two purposes.

First, you can use banked RAM data in a similar way as described for banked constant data, see *Banked constant data*, page 118, but for RAM objects, typically for buffers.

Second, if you have an operating system with separate stacks for different processes, you can place the stacks in data banks. When there is a context switch, you have to set the data bank register to make the appropriate stack visible in the 64-Kbyte area. Typically, you have to be careful with sharing data objects, which have been allocated on the stack, between processes.

In both cases, you have to manage the data bank register manually in similar explicit fashion as for banked constants. Banked RAM data is linked in a similar way as banked code. However, the addresses are different as the bank window for data typically is located at address `1000-1FFF`.

Part 2. Compiler reference

This part of the IAR C/C++ Compiler Reference Guide for HCS12 contains the following chapters:

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

Data representation

This chapter describes the data types, pointers, and structure types supported by the IAR C/C++ Compiler for HCS12.

See the chapter *Efficient coding for embedded applications* for information about which data types and pointers provide the most efficient code for your application.

Alignment

Every C data object has an alignment that controls how the object can be stored in memory. Should an object have an alignment of, for example, four, it must be stored on an address that is divisible by four.

The reason for the concept of alignment is that some processors have hardware limitations for how the memory can be accessed.

Assume that a processor can read 4 bytes of memory using one instruction, but only when the memory read is placed on an address divisible by 4. Then, 4-byte objects, such as `long` integers, will have alignment 4.

Another processor might only be able to read 2 bytes at a time; in that environment, the alignment for a 4-byte `long` integer might be 2.

A structure type will inherit the alignment from its components.

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

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

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

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

ALIGNMENT IN THE IAR C/C++ COMPILER FOR HCS12

The HCS12 microcontroller can access memory using 8- or 16-bit operations, and there are no alignment requirements.

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
bool	8 bits	0 to 1
char	8 bits	0 to 255
signed char	8 bits	-128 to 127
unsigned char	8 bits	0 to 255
signed short	16 bits	-32768 to 32767
unsigned short	16 bits	0 to 65535
signed int	16 bits	-32768 to 32767
unsigned int	16 bits	0 to 65535
signed long	32 bits	-2^{31} to $2^{31}-1$
unsigned long	32 bits	0 to $2^{32}-1$
signed long long	64 bits	-2^{63} to $2^{63}-1$
unsigned long long	64 bits	0 to $2^{64}-1$

Table 28: 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 `stdbool.h` file. This will also enable the boolean values `false` and `true`.

The enum type

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

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

The char type

The `char` type is by default unsigned in the compiler, but the `--char_is_signed` compiler option allows you to make it signed. Note, however, that the library is compiled with the `char` type as unsigned.

The wchar_t type

The `wchar_t` data type is an integer type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locals.

The `wchar_t` data type is supported by default in the C++ language. To use the `wchar_t` type also in C source code, you must include the `stddef.h` file from the runtime library.

Bitfields

In ISO/ANSI C, `int` and `unsigned int` can be used as the base type for integer bitfields. In the IAR C/C++ Compiler for HCS12, any integer type can be used as the base type when language extensions are enabled.

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

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

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

FLOATING-POINT TYPES

In the IAR C/C++ Compiler for HCS12, floating-point values are represented in standard IEEE 754 format.

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

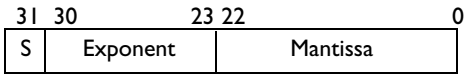
Type	Size	Range (+/-)	Exponent	Mantissa
float	32 bits	±1.18E-38 to ±3.39E+38	8 bits	23 bits
double *	32 bits (default)	±1.18E-38 to ±3.39E+38	8 bits	23 bits
double *	64 bits	±2.23E-308 to ±1.79E+308	11 bits	52 bits
long double *	32 bits	±1.18E-38 to ±3.39E+38	8 bits	23 bits
long double *	64 bits	±2.23E-308 to ±1.79E+308	11 bits	52 bits

Table 29: Floating-point types

*** Depends on whether the `--double` option is used, see *--double*, page 156. The type `long double` uses the same precision as `double`.**

32-bit floating-point format

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



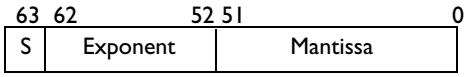
The value of the number is:

```
(-1)S * 2(Exponent-127) * 1.Mantissa
```

The precision of the float operators (`+`, `-`, `*`, and `/`) is approximately 7 decimal digits.

64-bit floating-point format

The representation of a 64-bit floating-point number as an integer is:



The value of the number is:

```
(-1)S * 2(Exponent-1023) * 1.Mantissa
```

The precision of the float operators (`+`, `-`, `*`, and `/`) is approximately 15 decimal digits.

Special cases

The following applies to both 32-bit and 64-bit floating-point formats:

- Zero is represented by zero mantissa and exponent. The sign bit signifies positive or negative zero.
- Infinity is represented by setting the exponent to the highest value and the mantissa to zero. The sign bit signifies positive or negative infinity.
- Not a number (`NaN`) is represented by setting the exponent to the highest positive value and the mantissa to a non-zero value. The value of the sign bit is ignored.
- Denormalized numbers are used to represent values smaller than what can be represented by normal values. The drawback is that the precision will decrease with smaller values. The exponent is set to 0 to signify that the number is denormalized, even though the number is treated as if the exponent would have been 1. Unlike normal numbers, denormalized numbers do not have an implicit 1 as MSB of the mantissa. The value of a denormalized number is:

  ```
  (-1)S * 2(1-BIAS) * 0.Mantissa
  ```

  where `BIAS` is 127 and 1023 for 32-bit and 64-bit floating-point values, respectively.

Pointer types

The IAR C/C++ Compiler for HCS12 has two basic types of pointers: code pointers and data pointers.

CODE POINTERS

The following table lists the available code pointers:

Pointer attribute	Pointer size	Default in code model	Description
`__non_banked`	2 bytes	Normal	Can address the entire 64 Kbyte memory space.
`__banked`	3 bytes	Banked	Must be used for banked functions.

Table 30: Code pointers

Casting

- Casting from `__non_banked` to `__banked` is performed through zero extension
- Casting from `__banked` to `__non_banked` is an illegal operation.

DATA POINTERS

The following table lists the available data pointers:

Pointer attribute	Pointer size	Address range
`__data8`	1 byte	`0x00-0xFF`
`__data16`	2 bytes (default)	`0x0000-0xFFFF`

Table 31: Data pointers

Casting

- Casting from `__data8` to `__data16` is performed through zero extension
- Casting from `__data16` to `__data8` is performed by truncation.

CASTING BETWEEN NON-RELATED TYPES

Casts between pointers have the following characteristics:

- Casting a *value* of an integer type to a pointer of a smaller type is performed by truncation
- Casting a *value* of an unsigned integer type to a pointer of a larger type is performed by zero extension

- Casting a *value* of a signed integer type to a pointer of a larger type is performed in two steps. In the first step, the value is sign extended to int size. In the second step, the int value is zero-extended to pointer size. In practice, *both* steps are performed only when casting from a signed char to a banked function pointer.
- Casting a *pointer type* to a smaller integer type is performed by truncation
- Casting a *pointer type* to a larger integer type is performed by zero extension
- Casting a *data pointer* to a function pointer and vice versa is illegal
- Casting a *function pointer* to an integer type gives an undefined result.

size_t

`size_t` is the unsigned integer type required to hold the maximum size of an object. In the IAR C/C++ Compiler for HCS12, the size of `size_t` is 16 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 IAR C/C++ Compiler for HCS12, the size of `ptrdiff_t` is 16 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 IAR C/C++ Compiler for HCS12, the size of `intptr_t` is 16 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

There are no alignment requirements for `struct` and `union` types.

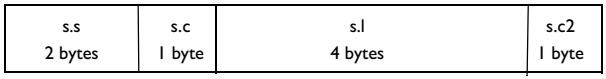
GENERAL LAYOUT

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 3, 4, 5, and 6 */
  char c2;  /* stored in byte 7 */
} s;
```

The following diagram shows the layout in memory:



Type and object attributes

The IAR C/C++ Compiler for HCS12 provides a set of attributes that support specific features of the HCS12 microcontroller. There are two basic types of attributes—*type attributes* and *object attributes*.

Type attributes affect the *external functionality* of the data object or function. For instance, how an object is placed in memory, or in other words, how it is accessed.

Object attributes affect the *internal functionality* of the data object or function.

To understand the syntax rules for the different attributes, it is important to be familiar with the concepts of the type attributes and the object attributes.

For information about how to use attributes to modify data, see the chapter *Data storage*. For information about how to use attributes to modify functions, see the chapter *Functions*. For detailed information about each attribute, see *Descriptions of extended keywords*, page 175.

TYPE ATTRIBUTES

Type attributes define how a function is called, or how a data object is accessed. This means that type attributes must be specified both when they are defined and in the declaration.

You can either place the type attributes directly in your source code, or use the pragma directive `#pragma type_attribute`.

Type attributes can be further divided into *memory attributes* and *general type attributes*.

Memory attributes

A memory attribute corresponds to a certain logical or physical memory in the microcontroller.

- Available *function memory attributes*: `__banked` and `__non_banked`
- Available *data memory attributes*: `__data8`, and `__data16`

For each level of indirection, you can only specify one memory attribute.

General type attributes

The following general type attributes are available:

- *Function type attributes* change the calling convention of a function: `__interrupt`, `__simple`, and `__task`
- *Data type attributes*: `const`, and `volatile`

For each level of indirection, you can specify as many type attributes as required.

OBJECT ATTRIBUTES

Object attributes affect functions and data objects, but not how the function is called or how the data is accessed. This means that an object attribute does not need to be present in the declaration of an object.

The following object attributes are available:

- Object attributes that can be used for variables: `__no_init`
- Object attributes that can be used for functions and variables: `location`, `@`, and `__root`
- Object attributes that can be used for functions: `__intrinsic`, `__monitor`, `__noreturn`, and `vector`.

Note: The `__intrinsic` attribute is reserved for compiler internal use only.

You can specify as many object attributes as required.

DECLARING OBJECTS IN SOURCE FILES

When declaring objects, note that he IAR-specific attributes work exactly like `const`. One exception to this is attributes that are declared in front of the type specifier apply to all declared objects.

See *More examples*, page 15.

DECLARING OBJECTS VOLATILE

There are three main reasons for declaring an object `volatile`:

- Shared access; the object is shared between several tasks in a multitasking environment
- Trigger access; as for a memory-mapped SFR where the fact that an access occurs has an effect
- Modified access; where the contents of the object can change in ways not known to the compiler.

Definition of access to volatile objects

The ISO/ANSI standard defines an abstract machine, which governs the behavior of accesses to `volatile` declared objects. The IAR C/C++ Compiler for HCS12 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.

Rules for accesses

Accesses to `volatile` declared objects are subject to the following rules:

1 All accesses are preserved

2 All accesses are complete, that is, the whole object is accessed

3 All accesses are performed in the same order as given in the abstract machine

4 All accesses are atomic, that is, non-interruptible.

The IAR C/C++ Compiler for HCS12 adheres to these rules for the following combinations of memory types and data types:

`__data8`	`char`, `short`, `int`, `__data8*`, `__data16*`
`__data16`	`char`, `short`, `int`, `__data8*`, `__data16*`

Data types in C++

In C++, all plain C data types are represented in the same way as described earlier in this chapter. However, if any Embedded C++ features are used for a type, no assumptions can be made concerning the data representation. This means, for example, that it is not legal to write assembler code that accesses class members.

Segment reference

The IAR C/C++ Compiler for HCS12 places code and data into named segments which are referred to by the IAR XLINK Linker™. Details about the segments are required for programming assembler language modules, and are also useful when interpreting the assembler language output from the compiler.

For information about how to define segments in the linker command file, see *Customizing the linker command file*, page 32.

Summary of segments

The table below lists the segments that are available in the IAR C/C++ Compiler for HCS12. Note that *located* denotes absolute location using the `@` operator or the `#pragma location` directive. The XLINK segment memory type `CODE`, `CONST`, or `DATA` indicates whether the segment should be placed in ROM or RAM memory areas; see Table 6, *XLINK segment memory types*, page 32.

Segment	Description	Type
`BANKED_CODE`	Holds banked program code.	`CODE`
`BANKED_CODE_SEGMENT_INIT`	Holds the segment initialization function.	`CODE`
`CODE`	Holds all code except for banked code.	`CODE`
`CSTACK`	Holds the stack used by C or C++ programs.	`DATA`
`DATA8_AC`	Holds located constant data.	`CONST` or `DATA`
`DATA8_AN`	Holds located `__data8` uninitialized data.	`CONST` or `DATA`
`DATA8_C`	Holds `__data8` declared constant data, including string literals.	`CONST`
`DATA8_I`	Holds initialized `__data8` declared data.	`DATA`
`DATA8_ID`	Holds `__data8` declared data that is copied to `DATA8_I` by `cstartup`.	`CONST`
`DATA8_N`	Holds uninitialized `__data8` declared data.	`DATA`
`DATA8_Z`	Holds zero-initialized `__data8` declared data.	`DATA`

Table 32: Segment summary

Segment	Description	Type
DATA16_AC	Holds located constant data.	CONST or DATA
DATA16_AN	Holds located uninitialized data.	CONST or DATA
DATA16_C	Holds __data16 declared constant data, including string literals.	CONST
DATA16_I	Holds initialized __data16 declared data.	DATA
DATA16_ID	Holds __data16 declared data that is copied to DATA16_I by cstartup.	CONST
DATA16_N	Holds uninitialized __data16 declared data.	DATA
DATA16_Z	Holds zero-initialized __data16 declared data.	DATA
DIFUNCT	Holds pointers to code, typically C++ constructors, which should be executed by the system startup code before main is called.	CONST
HEAP	Holds the heap data used by malloc and free.	DATA
INITTAB	Holds information about addresses and sizes of segments to be initialized at system startup.	CONST
INTVEC	Contains the reset and interrupt vectors.	CONST

Table 32: Segment summary (Continued)

Descriptions of segments

The following section gives reference information about each segment. For detailed information about the extended keywords mentioned here, see the chapter *Extended keywords*.

BANKED_CODE

Holds banked program code.

This segment holds banked code, which is:

- All code not declared __non_banked in the banked code model
- All code declared __banked in the normal code model.

Place this segment in memory using the -P segment control directives.

XLINK segment memory type

CODE

Memory range

This segment can be placed anywhere in banked memory.

Access type

Read-only

`BANKED_CODE_SEGMENT_INIT` In the Banked code model, this segment holds the function that performs the following at system startup:

- Copies the data initializers from the `*_ID` segments to their corresponding `*_I` segments
- Clears the `*_Z` segments.

The segment is used if you place data initializers in banked memory. For more information, see *Banked data initializers*, page 117. Note that you must not remove this segment from the linker command file when you are using the Banked code model, even if you do not intend to place data initializers in banked memory.

XLINK segment memory type

`CODE`

Memory range

This segment can be placed anywhere in banked memory.

Access type

Read-only

`CODE` Holds program code.

This segment holds code, that is, all code except for banked code. Place this segment in memory using the `-Z` or `-P` segment control directives.

XLINK segment memory type

`CODE`

Memory range

This segment can be placed anywhere within the first 64 Kbytes of memory.

Access type

Read-only

CSTACK Holds the internal data stack. This segment and its length are normally defined in the linker command file with the following command:

```
-Z(DATA)CSTACK+nn=start-end
```

where *nn* is the size of the stack specified as a hexadecimal number, and *start* and end specifies the available memory range for the stack.

XLINK segment memory type

DATA

Memory range

This segment can be placed anywhere within the first 64 Kbytes of memory.

Access type

Read/write

DATA8_AC Holds __data8 located constants.

This segment holds constants declared using the IAR absolute location placement extension (the @ operator). These constants do not need to—and must not—be given a location using segment placements options to the linker.

DATA8_AN Holds __data8 located non-initialized data.

This segment holds non-initialized data declared using the IAR absolute location placement extension (the @ operator). This data does not need to—and must not—be given a location using segment placements options to the linker.

DATA8_C Holds __data8 constant data, including string literals.

XLINK segment memory type

CONST

Memory range

This segment can be placed anywhere within the first 256 bytes of memory.

DATA8_I

Holds static and global initialized __data8 variables that have been declared with non-zero initial values. When you define this segment in the linker command file, the -Z segment control directive must be used and not the -P directive. The -P directive does not guarantee a fixed order between the segment parts when they are copied from the *_ID segments to their corresponding *_I segments.

XLINK segment memory type

DATA

Memory range

This segment can be placed anywhere within the first 256 bytes of memory.

DATA8_ID

Holds initial values for the variables located in the DATA8_I segment. These values are copied from DATA8_ID to DATA8_I during system initialization.When you define this segment in the linker command file, the -Z segment control directive must be used and not the -P directive. The -P directive does not guarantee a fixed order between the segment parts when they are copied from the *_ID segments to their corresponding *_I segments.

XLINK segment memory type

CONST

Memory range

This segment can be placed anywhere within the first 64 Kbytes of memory.

DATA8_N

Holds static and global __data8 variables that will not be initialized at system startup. Variables defined using the __no_init object attribute will be placed in this segment.

XLINK segment memory type

DATA

Memory range

This segment can be placed anywhere within the first 256 bytes of memory.

DATA8_Z Holds static and global `__data8` variables that have been declared without an initial value or with a zero-initialized value.

XLINK segment memory type

`DATA`

Memory range

This segment can be placed anywhere within the first 256 bytes of memory.

DATA16_AC Holds `__data16` located constants.

This segment holds constants declared using the IAR absolute location placement extension (the `@` operator). These constants do not need to—and must not—be given a location using segment placements options to the linker.

DATA16_AN Holds `__data16` located non-initialized data.

This segment holds non-initialized data declared using the IAR absolute location placement extension (the @ operator). This data does not need to—and must not—be given a location using segment placements options to the linker.

DATA16_C Holds `__data16` constant data, including string literals.

XLINK segment memory type

`CONST`

Memory range

This segment can be placed anywhere within the first 64 Kbytes of memory.

DATA16_I Holds static and global initialized `__data16` variables that have been declared with non-zero initial values. When you define this segment in the linker command file, the `-Z` segment control directive must be used and not the `-P` directive. The `-P` directive does not guarantee a fixed order between the segment parts when they are copied from the `*_ID` segments to their corresponding `*_I` segments.

XLINK segment memory type

`DATA`

Memory range

This segment can be placed anywhere within the first 64 Kbytes of memory.

DATA16_ID

Holds initial values for the variables located in the DATA16_I segment. These values are copied from DATA16_ID to DATA16_I during system initialization. When you define this segment in the linker command file, the -Z segment control directive must be used and not the -P directive. The -P directive does not guarantee a fixed order between the segment parts when they are copied from the *_ID segments to their corresponding *_I segments.

XLINK segment memory type

CONST

Memory range

This segment can be placed anywhere within the first 64 Kbytes of memory.

DATA16_N

Holds static and global __data16 variables that will not be initialized at system startup. Variables defined using the __no_init object attribute will be placed in this segment.

XLINK segment memory type

DATA

Memory range

This segment can be placed anywhere within the first 64 Kbytes of memory.

DATA16_Z

Holds static and global __data16 variables that have been declared without an initial value or with a zero-initialized value.

XLINK segment memory type

DATA

Memory range

This segment can be placed anywhere within the first 64 Kbytes of memory.

DIFUNCT Holds the dynamic initialization vector used by C++.

XLINK segment memory type

CONST

Memory range

This segment can be placed anywhere within the first 64 Kbytes of memory.

HEAP This segment holds dynamically allocated data, in other words data used by `malloc` and `free`, and in C++, `new` and `delete`.

This segment and its length is normally defined in the linker command file by the command:

`-Z(DATA)HEAP+`*`nn`*`=`*`start`*

where *`nn`* is the length and *`start`* is the location.

For more information about dynamically allocated data and the heap, see *The heap*, page 39.

XLINK segment memory type

DATA

Memory range

This segment can be placed anywhere within the first 64 Kbytes of memory.

INITTAB Holds the table containing addresses and sizes of segments that need to be initialized at startup.

XLINK segment memory type

CONST

Memory range

This segment can be placed anywhere within the first 64 Kbytes of memory.

INTVEC Holds the interrupt vector table generated by the use of the `__interrupt` extended keyword in combination with the `#pragma vector` directive.

XLINK segment memory type

`CODE`

Memory range

This segment is an absolute segment and should not be defined in the linker command file.

Compiler options

This chapter explains how to set the compiler options from the command line, and gives detailed reference information about each option.

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

Setting command line options

To set compiler options from the command line, include them on the command line after the `icchcs12` command, either before or after the source filename. For example, when compiling the source `prog.c`, use the following command to generate an object file with debug information:

```
icchcs12 prog --debug
```

Some options accept a filename, included after the option letter with a separating space. For example, to generate a listing to the file `list.lst`:

```
icchcs12 prog -l list.lst
```

Some other options accept a string that is not a filename. The string is included after the option letter, but without a space. For example, to define a symbol:

```
icchcs12 prog -DDEBUG=1
```

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

Note that a command line option has a *short* name and/or a *long* name:

- A short option name consists of one character, with or without parameters. You specify it with a single dash, for example `-e`
- A long name consists of one or several words joined by underscores, and it may have parameters. You specify it with double dashes, for example `--char_is_signed`.

SPECIFYING PARAMETERS

When a parameter is needed for an option with a short name, it can be specified either immediately following the option or as the next command line argument.

For instance, an include file path of `\usr\include` can be specified either as:

```
-I\usr\include
```

or as:

```
-I \usr\include
```

Additionally, output file options can take a parameter that is a directory name. The output file will then receive a default name and extension.

When a parameter is needed for an option with a long name, it can be specified either immediately after the equal sign (=) or as the next command line argument, for example:

```
--diag_suppress=Pe0001
```

or

```
--diag_suppress Pe0001
```

The option `--preprocess`, however, is an exception, as the filename must be preceded by a space. In the following example, comments are included in the preprocessor output:

```
--preprocess=c prog
```

Options that accept multiple values may be repeated, and may also have comma-separated values (without a space), for example:

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

The current directory is specified with a period (`.`), for example:

```
icchcs12 prog -l .
```

A file specified by `'-'` is standard input or output, whichever is appropriate.

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

```
icchcs12 prog -l ---r
```

SPECIFYING ENVIRONMENT VARIABLES

Compiler options can also be specified in the `QCCHCS12` environment variable. The compiler automatically appends the value of this variable to every command line, so it provides a convenient method of specifying options that are required for every compilation.

The following environment variables can be used with the IAR C/C++ Compiler for HCS12:

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\hcs12\inc;c:\headers`
QCCHCS12	Specifies command line options; for example: `QCCHCS12=-lA asm.lst -z9`

Table 33: Environment variables

ERROR RETURN CODES

The IAR C/C++ Compiler for HCS12 returns status information to the operating system which can be tested in a batch file.

The following command line error codes are supported:

Code	Description
0	Compilation successful, but there may have been warnings.
1	There were warnings, provided that the option `--warnings_affect_exit_code` was used.
2	There were non-fatal errors or fatal compilation errors making the compiler abort.
3	There were fatal errors.

Table 34: Error return codes

Options summary

The following table summarizes the compiler command line options:

Command line option	Description
`--char_is_signed`	Treats `char` as signed
`--code_model={banked|normal}`	Specifies the code model
`--cross_call_passes=`*`N`*	Cross-call optimization
`-D`*`symbol`*`[=`*`value`*`]`	Defines preprocessor symbols
`--debug`	Generates debug information
`--dependencies[=[i][m]] {`*`filename`*`|`*`directory`*`}`	Lists file dependencies
`--diag_error=`*`tag`*`,`*`tag`*`,...`	Treats these as errors

Table 35: Compiler options summary

Command line option	Description
`--diag_remark=`*tag*`,`*tag*`,...`	Treats these as remarks
`--diag_suppress=`*tag*`,`*tag*`,...`	Suppresses these diagnostics
`--diag_warning=`*tag*`,`*tag*`,...`	Treats these as warnings
`--diagnostics_tables {`*filename*`|`*directory*`}`	Lists all diagnostic messages
`-dlib_config` *filename*	Determines library configuration file
`--double={32|64}`	Forces the compiler to use 32-bit or 64-bit doubles
`--do_cross_call`	Forces cross-call optimization
`-e`	Enables language extensions
`--ec++`	Enables Embedded C++ syntax
`--eec++`	Enables Extended Embedded C++ syntax
`--enable_multibytes`	Enables support for multibyte characters
`--error_limit=`*n*	Specifies the allowed number of errors before compilation stops
`-f` *filename*	Extends the command line
`--force_switch_type={0|1|2}`	Sets the switch type
`--header_context`	Lists all referred source files
`-I`*path*	Specifies include file path
`-l[a|A|b|B|c|C|D][N][H] {`*filename*`|`*directory*`}`	Creates a list file
`--library_module`	Creates a library module
`--migration_preprocessor_extensions`	Extends the preprocessor
`--module_name=`*name*	Sets object module name
`--no_code_motion`	Disables code motion optimization
`--no_cross_call`	Disables cross-call optimization
`--no_cse`	Disables common subexpression elimination

Table 35: Compiler options summary (Continued)

Command line option	Description
`--no_inline`	Disables function inlining
`--no_tbaa`	Disables type-based alias analysis
`--no_typedefs_in_diagnostics`	Disables the use of typedef names in diagnostics
`--no_ubrof_messages`	Minimizes object file size
`--no_unroll`	Disables loop unrolling
`--no_warnings`	Disables all warnings
`--no_wrap_diagnostics`	Disables wrapping of diagnostic messages
`-o {`*`filename`*`|`*`directory`*`}`	Sets object filename
`--omit_types`	Excludes type information
`--only_stdout`	Uses standard output only
`--preinclude` *`includefile`*	Includes an include file before reading the source file
`--preprocess[=[c][n][l]] {`*`filename`*`|`*`directory`*`}`	Generates preprocessor output
`--public_equ` *`symbol`*`[=`*`value`*`]`	Defines a global named assembler label
`-r`	Generates debug information
`--remarks`	Enables remarks
`--require_prototypes`	Verifies that prototypes are proper
`--root_functions`	Treats all functions as `__root`
`--root_variables`	Treats all variables as `__root`
`-s[2|3|6|9]`	Optimizes for speed
`--segment` *`memory_attr=segment_name`*	Changes segment name base
`--silent`	Sets silent operation
`--strict_ansi`	Checks for strict compliance with ISO/ANSI C
`--warnings_affect_exit_code`	Warnings affects exit code

Table 35: Compiler options summary (Continued)

Command line option	Description
`--warnings_are_errors`	Warnings are treated as errors
`-z[2|3|6|9]`	Optimizes for size

Table 35: Compiler options summary (Continued)

Descriptions of options

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

--char_is_signed

`--char_is_signed`

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

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

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

--code_model

`--code_model={banked|b|normal|n}`

The HCS12 microcontroller can be used in two modes: normal mode and banked mode. The IAR C/C++ Compiler for HCS12 supports these modes by means of code models.

The code model controls how code is generated for an application. All object files of an application must be compiled using the same code model. The supported code models are:

Code model	Description
`normal` (default)	Non-banked function calls
`banked`	Banked function calls

Table 36: Available code models

If you do not include any of the code model options, the compiler uses the normal code model as default.

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

Example

For example, use the following command to specify the banked code model:

```
--code_model banked
```

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>General Options>Target**.

`--cross_call_passes`

`--cross_call_passes=`*`N`*

Use this option to decrease the `STACK` usage by running the cross-call optimizer *`N`* times, where *`N`* can be 1–5. The default is to run it until no more improvements are possible.

For additional information, see *--no_cross_call*, page 162.

This option is related to the **Optimizations** options in the **Project>Options>C/C++ Compiler>Optimization** category in the IAR Embedded Workbench.

`-D`

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

Use this option to define a preprocessor symbol with the name *`symbol`* and the value *`value`*. If no value is specified, `1` is used.

The option `-D` has the same effect as a `#define` statement at the top of the source file:

`-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=
```

This option can be used one or more times on the command line.

Example

You may want to arrange your source to produce either the test or production version of your program, depending on whether the symbol `TESTVER` was defined. To do this, you would use include sections such as:

```
#ifdef  TESTVER
  ...   additional code lines for test version only
#endif
```

Then, you would select the version required on the command line as follows:

```
Production version:   icchcs12 prog
Test version:         icchcs12 prog -DTESTVER
```

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

`--debug, -r`

```
--debug
-r
```

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

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

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

`--dependencies`

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

Use this option to make the compiler write information to a file about each source code file opened by the compiler. The following modifiers are available:

Option modifier	Description
`i`	Lists only the names of files (default)
`m`	Lists in makefile style

Table 37: Generating a list of dependencies (--dependencies)

If a *`filename`* is specified, the compiler stores the output in that file.

If a *`directory`* is specified, the compiler stores the output in that directory, in a file with the extension `i`. The filename will be the same as the name of the compiled source file, unless a different name has been specified with the option `-o`, in which case that name will be used.

To specify the working directory, replace *`directory`* with a period (`.`).

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

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

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

```
foo.r12: c:\iar\product\include\stdio.h
foo.r12: d:\myproject\include\foo.h
```

Example 1

To generate a listing of file dependencies to the file `listing.i`, use:

```
icchcs12 prog --dependencies=i listing
```

Example 2

To generate a listing of file dependencies to a file called `listing.i` in the `mypath` directory, you would use:

```
icchcs12 prog --dependencies mypath\listing
```

Example 3

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

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

```
%.r12 : %.c
      $(ICC) $(ICCFLAGS) $< --dependencies=m $*.d
```

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

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

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

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

`--diag_error`

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

Use this option to classify diagnostic messages as errors. An error indicates a violation of the C or C++ language rules, of such severity that object code will not be generated, and the exit code will be non-zero.

Example

The following example classifies warning `Pe117` as an error:

```
--diag_error=Pe117
```

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

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

Use this option to classify diagnostic messages as remarks. A remark is the least severe type of diagnostic message and indicates a source code construct that may cause strange behavior in the generated code.

Example

The following example classifies the warning `Pe177` as a remark:

```
--diag_remark=Pe177
```

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

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

Use this option to suppress diagnostic messages.

Example

The following example suppresses the warnings `Pe117` and `Pe177`:

```
--diag_suppress=Pe117,Pe177
```

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

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

Use this option to classify diagnostic messages as warnings. A warning indicates an error or omission that is of concern, but which will not cause the compiler to stop before compilation is completed.

Example

The following example classifies the remark `Pe826` as a warning:

```
--diag_warning=Pe826
```

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

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

Use this option to list all possible diagnostic messages in a named file. This can be very convenient, for example, if you have used a pragma directive to suppress or change the severity level of any diagnostic messages, but forgot to document why.

This option cannot be given together with other options.

If a *`filename`* is specified, the compiler stores the output in that file.

If a *`directory`* is specified, the compiler stores the output in that directory, in a file with the name `diagnostics_tables.txt`. To specify the working directory, replace *`directory`* with a period (`.`).

Example 1

To output a list of all possible diagnostic messages to the file `diag.txt`, use:

```
--diagnostics_tables diag
```

Example 2

If you want to generate a table to a file `diagnostics_tables.txt` in the working directory, you could use:

```
--diagnostics_tables .
```

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

Each runtime library has a corresponding library configuration file. Use the `--dlib_config` option to specify the library configuration file for the compiler. Make sure that you specify a configuration file that corresponds to the library you are using.

All prebuilt runtime libraries are delivered with corresponding configuration files. You can find the library object files and the library configuration files in the directory `hcs12\lib`. For examples and a list of prebuilt runtime libraries, see *Using a prebuilt library*, page 50.

If you have built your own customized runtime library, you should also have created a corresponding customized library configuration file, which must be specified to the compiler. For more information, see *Building and using a customized library*, page 56.

Note: This option only applies to the IAR DLIB runtime environment.

To set the related option in the IAR Embedded Workbench, select **Project>Options>General Options>Library Configuration**.

`--do_cross_call` `--do_cross_call`

Using this option forces the compiler to run the cross-call optimizer, regardless of the optimization level. The cross-call optimizer is otherwise only run at high size optimization (`-z9`).

This option is related to the **Optimizations** options in the **Project>Options>C/C++ Compiler>Optimization** category in the IAR Embedded Workbench.

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

Use this option to force the compiler to use 64-bit doubles or 32-bit doubles. By default, the compiler uses 32-bit doubles. For additional information, see *Floating-point types*, page 127.

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>General Options>Target**.

`-e` `-e`

In the command line version of the IAR C/C++ Compiler for HCS12, language extensions are disabled by default. If you use language extensions such as HCS12-specific keywords and anonymous structs and unions in your source code, you must enable them by using this option.

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

For additional information, see *Special support for embedded systems*, page 8.

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

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

In the IAR C/C++ Compiler for HCS12, the default language is C. If you use Embedded C++, you must use this option to set the language the compiler uses to Embedded C++.

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

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

In the IAR C/C++ Compiler for HCS12, 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 *Extended Embedded C++*, page 96.

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

--enable_multibytes `--enable_multibytes`

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

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

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

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

Use the `--error_limit` option to specify the number of errors allowed before the compiler stops the compilation. By default, 100 errors are allowed. *n* must be a positive number; `0` indicates no limit.

-f `-f` *filename*

Reads command line options from the named file, with the default extension `xcl`.

By default, the compiler accepts command parameters only from the command line itself and the `QCCHCS12` environment variable. To make long command lines more manageable, and to avoid any operating system command line length limit, you can use the `-f` option to specify a command file, from which the compiler reads command line items as if they had been entered at the position of the option.

In the command file, you format the items exactly as if they were on the command line itself, except that you may use multiple lines, because the newline character acts just as a space or tab character.

Both C and C++ style comments are allowed in the file. Double quotes behave as in the Microsoft Windows command line environment.

Example

For example, you could replace the command line:

```
icchcs12 prog -r "-DUsername=John Smith" -DUserid=463760
```

with

```
icchcs12 prog -r -f userinfo
```

if the file `userinfo.xcl` contains:

```
"-DUsername=John Smith"
-DUserid=463760
```

`--force_switch_type` `--force_switch_type={0|1|2}`

Use this option to override compiler heuristics in deciding the method for implementing switch tables.

Choose between:

Option modifier	Switch type
0	Library call with switch table containing pairs of values and labels
1	Inline code with jump table containing addresses
2	Inline compare/jump logic

Table 38: Specifying switch type

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

`--header_context` `--header_context`

Occasionally, to find the cause of a problem it is necessary to know which header file was included from which source line. Use this option to list, for each diagnostic message, not only the source position of the problem, but also the entire include stack at that point.

`-I` `-I`*`path`*
`-I path`

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

Following is the full description of the compiler's `#include` file search procedure:

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

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

  it searches the following directories for the file to include:

  1 The directories specified with the `-I` option, in the order that they were specified.

  2 The directories specified using the `C_INCLUDE` environment variable, if any.

- If the compiler encounters the name of an `#include` file in double quotes, for example:

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

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

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

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

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

  ```
  icchcs12 ..\src\src.c -I..\include -I..\debugconfig
  ```

  Then the following directories are searched in the order listed below for the `config.h` file, which in this example is located in the `dir\debugconfig` directory:

`dir\include`	Current file.
`dir\src`	File including current file.
`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.

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

`-l` `-l[a|A|b|B|c|C|D][N][H] {filename|directory}`

By default, the compiler does not generate a listing. Use this option to generate a listing to a file.

The following modifiers are available:

Option modifier	Description
`a`	Assembler list file
`A`	Assembler file with C or C++ source as comments
`b`	Basic assembler list file. This file has the same contents as a list file produced with `-la`, except that none of the extra compiler generated information (runtime model attributes, call frame information, frame size information) is included *
`B`	Basic assembler list file. This file has the same contents as a list file produced with `-lA`, except that none of the extra compiler generated information (runtime model attributes, call frame information, frame size information) is included *
`c`	C or C++ list file
`C` (default)	C or C++ list file with assembler source as comments
`D`	C or C++ list file with assembler source as comments, but without instruction offsets and hexadecimal byte values
`N`	No diagnostics in file
`H`	Include source lines from header files in output. Without this option, only source lines from the primary source file are included

Table 39: Generating a compiler list file (-l)

*** This makes the list file less useful as input to the assembler, but more useful for reading by a human.**

If a *`filename`* is specified, the compiler stores the output in that file.

If a *`directory`* is specified, the compiler stores the output in that directory, in a file with the extension `lst`. The filename will be the same as the name of the compiled source file, unless a different name has been specified with the option `-o`, in which case that name will be used.

To specify the working directory, replace *`directory`* with a period (`.`).

Example 1

To generate a listing to the file `list.lst`, use:

```
icchcs12 prog -l list
```

Example 2

If you compile the file `mysource.c` and want to generate a listing to a file `mysource.lst` in the working directory, you could use:

```
icchcs12 mysource -l .
```

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

`--library_module` `--library_module`

Use this option to make the compiler generate a library module rather than a program module. A program module is always included during linking. A library module will only be included if it is referenced in your program.

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

`--migration_preprocessor_extensions` `--migration_preprocessor_extensions`

If you need to migrate code from an earlier IAR C or C/C++ compiler, you may want to use this option. With this option, the following can be used in preprocessor expressions:

- Floating-point expressions
- Basic type names and `sizeof`
- All symbol names (including typedefs and variables).

Note: If you use this option, not only will the compiler accept code that is not standard conformant, but it will also reject some code that *does* conform to the standard.

Important! Do not depend on these extensions in newly written code, as support for them may be removed in future compiler versions.

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

`--module_name` `--module_name=`*`name`*

Normally, the internal name of the object module is the name of the source file, without a directory name or extension. Use this option to specify an object module name.

To set the object module name explicitly, use the option `--module_name=`*`name`*, for example:

```
icchcs12 prog --module_name=main
```

This option is useful when several modules have the same filename, because the resulting duplicate module name would normally cause a linker error; for example, when the source file is a temporary file generated by a preprocessor.

Example

The following example—in which `%1` is an operating system variable containing the name of the source file—will give duplicate name errors from the linker:

```
preproc %1.c temp.c                ; preprocess source,
                                   ; generating temp.c
icchcs12 temp.c                    ; module name is
                                   ; always 'temp'
```

To avoid this, use `--module_name=`*name* to retain the original name:

```
preproc %1.c temp.c                ; preprocess source,
                                   ; generating temp.c
icchcs12 temp.c --module_name=%1   ; use original source
                                   ; name as module name
```

Note: In this example, `preproc` is an external utility.

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

`--no_code_motion` `--no_code_motion`

Use this option to disable optimizations that move code. These optimizations, which are performed at optimization levels 6 and 9, normally reduce code size and execution time. However, the resulting code may be difficult to debug.

Note: This option has no effect at optimization levels below 6.

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

`--no_cross_call` `--no_cross_call`

Use this option to disable the cross-call optimization.

This optimization is performed at size optimization, level 7–9. Notice that, although it can drastically reduce the code size, this option increases the execution time.

For additional information, see *--cross_call_passes*, page 151.

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

--no_cse --no_cse

Use --no_cse to disable common subexpression elimination.

At optimization levels 6 and 9, the compiler avoids calculating the same expression more than once. This optimization normally reduces both code size and execution time. However, the resulting code may be difficult to debug.

Note: This option has no effect at optimization levels below 6.

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

--no_inline --no_inline

Use --no_inline to disable function inlining.

Function inlining means that a simple function, whose definition is known at compile time, is integrated into the body of its caller to eliminate the overhead of the call.

This optimization, which is performed at optimization level 9, normally reduces execution time and increases code size. The resulting code may also be difficult to debug.

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

Note: This option has no effect at optimization levels below 9.

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

--no_tbaa --no_tbaa

Use --no_tbaa to disable type-based alias analysis. When this options is not used, the compiler is free to assume that objects are only accessed through the declared type or through unsigned char. See *Type-based alias analysis*, page 108 for more information.

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

--no_typedefs_in_diagnostics --no_typedefs_in_diagnostics

Normally, when a type is mentioned in a message from the compiler, most commonly in a diagnostic message of some kind, the typedef names that were used in the original declaration are used whenever they make the resulting text shorter. For example,

```
typedef int (*MyPtr)(char const *);
MyPtr p = "foo";
```

will give an error message like the following:

```
Error[Pe144]: a value of type "char *" cannot be used to
initialize an entity of type "MyPtr"
```

If the `--no_typedefs_in_diagnostics` option is specified, the error message will be like this:

```
Error[Pe144]: a value of type "char *" cannot be used to
initialize an entity of type "int (*)(char const *)"
```

--no_unroll --no_unroll

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.

This optimization, which is performed at optimization level 9, normally reduces execution time, but increases code size. The resulting code may also be difficult to debug.

The compiler heuristically decides which loops to unroll. Different heuristics are used when optimizing for speed and size.

Note: This option has no effect at optimization levels below 9.

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

--no_ubrof_messages --no_ubrof_messages

By default, range error messages are embedded in the UBROF output object file. These messages can contain tiny fragments of your source code.

Use this option if you do not want the UBROF output file to contain this type of information. The drawback is that the range error messages will be less helpful.

For information about a related option, see *--omit_types*, page 166.

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

`--no_warnings` `--no_warnings`

By default, the compiler issues warning messages. Use this option to disable all warning messages.

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

`--no_wrap_diagnostics` `--no_wrap_diagnostics`

By default, long lines in compiler diagnostic messages are broken into several lines to make the message easier to read. Use this option to disable line wrapping of diagnostic messages.

`-o` `-o {`*`filename`*`|`*`directory`*`}`

Use the `-o` option to specify an output file for object code.

If a *`filename`* is specified, the compiler stores the object code in that file.

If a *`directory`* is specified, the compiler stores the object code in that directory, in a file with the same name as the name of the compiled source file, but with the extension `r12`. To specify the working directory, replace *`directory`* with a period (`.`).

Example 1

To store the compiler output in a file called `obj.r12` in the `mypath` directory, you would use:

```
icchcs12 mysource -o mypath\obj
```

Example 2

If you compile the file `mysource.c` and want to store the compiler output in a file `mysource.r12` in the working directory, you could use:

```
icchcs12 mysource -o .
```

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>General Options>Output**.

`--omit_types` `--omit_types`

By default, the compiler includes type information about variables and functions in the object output.

Use this option if you do not want the compiler to include this type information in the output. The object file will then only contain type information that is a part of a symbol's name. This means that the linker cannot check symbol references for type correctness, which is useful when you build a library that should not contain type information.

For information about a related option, see *--no_ubrof_messages*, page 164.

`--only_stdout` `--only_stdout`

Use this option to make the compiler use the standard output stream (`stdout`) also for messages that are normally directed to the error output stream (`stderr`).

`--preinclude` `--preinclude` *`includefile`*

Use this option to make the compiler include the specified include file before it starts to read the source file. This is useful if you want to change something in the source code for the entire application, for instance if you want to define a new symbol.

`--preprocess` `--preprocess[=[c][n][l]] {`*`filename`*`|`*`directory`*`}`

Use this option to direct preprocessor output to a named file.

The following table shows the mapping of the available preprocessor modifiers:

Command line option	Description
`--preprocess=c`	Preserve comments
`--preprocess=n`	Preprocess only
`--preprocess=l`	Generate `#line` directives

Table 40: Directing preprocessor output to file (--preprocess)

If a *`filename`* is specified, the compiler stores the output in that file.

If a *`directory`* is specified, the compiler stores the output in that directory, in a file with the extension `i`. The filename will be the same as the name of the compiled source file, unless a different name has been specified with the option `-o`, in which case that name will be used.

To specify the working directory, replace *directory* with a period (.).

Example 1

To store the compiler output with preserved comments to the file `output.i`, use:

```
icchcs12 prog --preprocess=c output
```

Example 2

If you compile the file `mysource.c` and want to store the compiler output with `#line` directives to a file `mysource.i` in the working directory, you could use:

```
icchcs12 mysource --preprocess=l .
```

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

`--public_equ`

```
--public_equ symbol[=value]
```

This option is equivalent to defining a label in assembler language by using the `EQU` directive and exporting it using the `PUBLIC` directive.

`-r, --debug`

```
-r
--debug
```

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

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

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

`--remarks`

```
--remarks
```

The least severe diagnostic messages are called remarks (see *Severity levels*, page 231). A remark indicates a source code construct that may cause strange behavior in the generated code.

By default, the compiler does not generate remarks. Use this option to make the compiler generate remarks.

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

--require_prototypes `--require_prototypes`

This option forces the compiler to verify that all functions have proper prototypes. Using this option means that code containing any of the following will generate an error:

- A function call of a function with no declaration, or with a Kernighan & Ritchie C declaration
- A function definition of a public function with no previous prototype declaration
- An indirect function call through a function pointer with a type that does not include a prototype.

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

--root_functions `--root_functions`

Use this option to apply the `__root` extended keyword to all functions. This will make sure that the functions are not removed by the IAR XLINK Linker.

Notice that the `--root_functions` option is always available, even if you do not specify the compiler option `-e`, language extensions.

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

--root_variables `--root_variables`

Use this option to apply the `__root` extended keyword to all global and static variables. This will make sure that the variables are not removed by the IAR XLINK Linker.

Notice that the `--root_variables` option is always available, even if you do not specify the compiler option `-e`, language extensions.

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

-s `-s[2|3|6|9]`

Use this option to make the compiler optimize the code for maximum execution speed.

If no optimization option is specified, the compiler will use the size optimization `-z3` by default. If the `-s` option is used without specifying the optimization level, speed optimization at level 3 is used by default.

The following table shows how the optimization levels are mapped:

Option modifier	Optimization level
2	None* (Best debug support)
3	Low*
6	Medium
9	High (Maximum optimization)

Table 41: Specifying speed optimization (-s)

***The most important difference between `-s2` and `-s3` is that at level 2, all non-static variables will live during their entire scope.**

A low level of optimization makes it relatively easy to follow the program flow in the debugger, and, conversely, a high level of optimization makes it relatively hard.

Note: The `-s` and `-z` options cannot be used at the same time.

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

`--segment` `--segment` *`memory_attribute=segment_name`*

The IAR C/C++ Compiler for HCS12 places code and data into named segments which are referred to by the IAR XLINK Linker. Use the `--segment` option to place any part of your application into separate non-default segments. This is useful if you want to control placement of your code or data to different address ranges and you find the `@` notation insufficient.

By default, *data objects* are placed in segments named *`BASENAME`*`_*`, where *`BASENAME`* is derived from the corresponding memory attributes `__data8` and `__data16`, and `*` is any of the suffixes `AC`, `AN`, `C`, `I`, `ID`, `N`, or `Z`. By default, *non-banked code* is placed in the `CODE` segment and *banked code* is placed in the `BANKED_CODE` segment. In addition to the memory attributes, you can change the name of the segments that corresponds to the following strings: `__intvec`, `__difunct`, `__cstack`, and `__inittab`.

Note that any changes to the segment names require corresponding modification in the linker command file.

For detailed information about segments, see *Placing segments in memory*, page 32.

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

--silent `--silent`

By default, the compiler issues introductory messages and a final statistics report. Use `--silent` to make the compiler operate without sending these messages to the standard output stream (normally the screen).

This option does not affect the display of error and warning messages.

--strict_ansi `--strict_ansi`

By default, the compiler accepts a relaxed superset of ISO/ANSI C (see the chapter *IAR language extensions*). Use `--strict_ansi` to ensure that the program conforms to the ISO/ANSI C standard.

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

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

--warnings_affect_exit_code `--warnings_affect_exit_code`

By default, the exit code is not affected by warnings, as only errors produce a non-zero exit code. With this option, warnings will generate a non-zero exit code.

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

--warnings_are_errors `--warnings_are_errors`

Use this option to make the compiler treat all warnings as errors. If the compiler encounters an error, no object code is generated. Warnings that have been changed into remarks are not treated as errors.

Note: Any diagnostic messages that have been reclassified as warnings by the compiler option `--diag_warning` or the `#pragma diag_warning` directive will also be treated as errors when `--warnings_are_errors` is used.

For additional information, see *--diag_warning*, page 154 and *#pragma diag_warning*, page 183.

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

`-z` `-z[2|3|6|9]`

Use this option to make the compiler optimize the code for minimum size. If no optimization option is specified, `-z3` is used by default.

The following table shows how the optimization levels are mapped:

Option modifier	Optimization level
2	None* (Best debug support)
3	Low*
6	Medium
9	High (Maximum optimization)

Table 42: Specifying size optimization (-z)

***The most important difference between `-z2` and `-z3` is that at level 2, all non-static variables will live during their entire scope.**

A low level of optimization makes it relatively easy to follow the program flow in the debugger, and, conversely, a high level of optimization makes it relatively hard.

Note: The `-s` and `-z` options cannot be used at the same time.

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

Extended keywords

This chapter describes the extended keywords that support specific features of the HCS12 microcontroller, the general syntax rules for the keywords, and a detailed description of each keyword.

For information about the address ranges of the different memory areas, see the chapter *Segment reference*.

Using extended keywords

This section covers how extended keywords can be used when declaring and defining data and functions. The syntax rules for extended keywords are also described.

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

The keywords and the `@` operator are only available when language extensions are enabled in the IAR C/C++ Compiler for HCS12.

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

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

EXTENDED KEYWORDS FOR DATA

The extended keywords that can be used for data can be divided into two groups that control the following:

- The memory type of objects and pointers. Keywords of this group must be specified both when the object is declared and when it is defined: `__data8`, and `__data16`
- Other characteristics of objects: `@`, `__root` and `__no_init`.

See the chapter *Data storage* in *Part 1. Using the compiler* for more information about how to use data memory types. To read more about `__no_init`, see *Non-initialized variables*, page 116.

EXTENDED KEYWORDS FOR FUNCTIONS

The extended keywords that can be used when functions are declared can be divided into two groups:

- Keywords that control the type of the functions. Keywords of this group must be specified both when the function is declared and when it is defined:`__banked`, `__non_banked`, `__interrupt`, `__simple`, and `__task`.
- Keywords that only control the defined function but not the function call: `@`, `__monitor`, `__noreturn`, and `__root`.

See the chapter *Functions* in *Part 1. Using the compiler* for more information about how to use function memory types.

Summary of extended keywords

Some extended keywords are used on data, some on functions, and some can be used on both data and functions.

The following table summarizes the extended keywords that can be used on functions:

Extended keywords for functions	Description
`__banked`	Controls the storage of functions; specifies a function to be placed in banked memory
`__interrupt`	Supports interrupt functions
`__intrinsic`	Reserved for compiler internal use only
`__monitor`	Supports atomic execution of a function
`__non_banked`	Specifies a function to be non-banked and called by a standard `CALL` instruction
`__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
`__simple`	Specifies old calling convention; available for backward compatibility
`__task`	Allows functions to exit without restoring registers

Table 43: Summary of extended keywords for functions

The following table summarizes the extended keywords that can be used on data:

Extended keywords for data	Description
`__data8`	Controls the storage of variables

Table 44: Summary of extended keywords for data

Extended keywords for data	Description
__data16	Controls the storage of variables
__no_init	Supports non-volatile memory
__root	Ensures that a function or variable is included in the object code even if unused

Table 44: Summary of extended keywords for data (Continued)

Note: Some of the keywords can be used on both data and functions.

Descriptions of extended keywords

The following sections give detailed information about each extended keyword.

__banked

The __banked keyword is used on a function to make it banked, which means a 3-byte CALL instruction will be used at function calls. The function can be called from any code bank. All functions are by default banked in the banked code model. This keyword can be specified using the #pragma type_attribute directive.

__data8

The __data8 extended keyword places variables and constants in data8 memory, and determines how objects are accessed. The maximum object size is 255 bytes.

This keyword can be specified using the #pragma type_attribute directive.

Maximum memory size

256 bytes.

__data16

The __data16 extended keyword places variables and constants in data16 memory, and determines how objects are accessed. The maximum object size is 64 Kbytes-1.

This keyword can be specified using the #pragma type_attribute directive.

Maximum memory size

64 Kbytes.

__interrupt

The `__interrupt` keyword specifies interrupt functions. The `#pragma vector` directive can be used for specifying the interrupt vector(s), see *#pragma vector*, page 188. An interrupt function must have a `void` return type and cannot have any parameters.

The following example declares an interrupt function with an interrupt vector with the absolute address `0xFFF4` in the `INTVEC` segment:

```
#pragma vector=0xFFF4
__interrupt void my_interrupt_handler(void);
```

An interrupt function cannot be called directly from a C program. It can only be executed as a response to an interrupt request.

It is possible to define an interrupt function without a vector, but then the compiler will not generate an entry in the interrupt vector table. For additional information, see *INTVEC*, page 143.

The range of the interrupt vectors depends on the device used.

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

For additional information about interrupt functions, see *Interrupt functions*, page 21.

__intrinsic

The `__intrinsic` keyword is reserved for compiler internal use only.

__monitor

The `__monitor` keyword causes interrupts to be disabled during execution of the function. This allows atomic operations to be performed, such as operations on semaphores that control access to resources by multiple processes. A function declared with the `__monitor` keyword is equivalent to any other function in all other respects. This keyword can be specified using the `#pragma object_attribute` directive.

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

For additional information, see the intrinsic functions *__disable_interrupt*, page 199, and *__enable_interrupt*, page 199.

Read more about monitor functions in *Monitor functions*, page 22.

__non_banked

The `__non_banked` keyword is used on a function to make it non-banked, which means a standard 2-byte `JSR` instruction will be used at function calls. The function can be called from any code bank. All functions are by default non-banked in the Normal code model. For a banked application, this keyword can be used for functions that must reside in non-banked memory, typically for efficiency reasons.

This keyword can be specified using the `#pragma type_attribute` directive.

`__no_init` The `__no_init` keyword is used for suppressing initialization of a variable at system startup.

The `__no_init` keyword is placed in front of the type. In this example, `myarray` is placed in a non-initialized segment:

```
__no_init int myarray[10];
```

The `#pragma object_attribute` directive can also be used. The following declaration is equivalent to the previous one:

```
#pragma object_attribute=__no_init
int myarray[10];
```

Note: The `__no_init` keyword cannot be used in combination with the `typedef` keyword.

`__noreturn` The `__noreturn` keyword can be used on a function to inform the compiler that the function will not return. If you use this keyword on such functions, the compiler can optimize more efficiently. Examples of functions that do not return are `abort` and `exit`.

The `__noreturn` keyword is an object attribute, which means the `#pragma object_attribute` directive can be used for specifying it. For more information about object attributes, see *Object attributes*, page 132.

`__root` The `__root` attribute can be used on a function or a variable to ensure that, when the module containing the function or variable is linked, the function or variable is also included, whether or not it is referenced by the rest of the program.

By default, only the part of the runtime library calling `main` and any interrupt vectors are root. All other functions and variables are included in the linked output only if they are referenced by the rest of the program.

The `__root` keyword is placed in front of the type, for example to place `myarray` in non-volatile memory:

```
__root int myarray[10];
```

The `#pragma object_attribute` directive can also be used. The following declaration is equivalent to the previous one:

```
#pragma object_attribute=__root
int myarray[10];
```

Note: The `__root` keyword cannot be used in combination with the `typedef` keyword.

`__simple` The `__simple` keyword is available for backward compatibility of the interface for calling assembler routines from C. It makes a function use the calling convention of the 68HC12 IAR C Compiler version 2.x instead of the default calling convention, both which are described in *Calling convention*, page 85.

This calling convention is preferred when calling assembler functions from C.

`__task` Allows functions to exit without restoring registers. This keyword is typically used for the `main` function.

By default, functions save the contents of used non-scratch registers (permanent registers) on the stack upon entry, and restore them at exit. Functions declared as `__task` do not save any registers, and therefore require less stack space. Such functions should only be called from assembler routines.

The function `main` may be declared `__task`, unless it is explicitly called from the application. In real-time applications with more than one task, the root function of each task may be declared `__task`.

The keyword is placed in front of the return type, for instance:

```
__task void my_handler(void);
```

The `#pragma type_attribute` directive can also be used. The following declaration of `my_handler` is equivalent with the previous one:

```
#pragma type_attribute=__task
void my_handler(void);
```

The `__task` keyword must be specified both in the function declaration and when the function is defined.

Pragma directives

This chapter describes the pragma directives of the IAR C/C++ Compiler for HCS12.

The pragma directives control the behavior of the compiler, for example how it allocates memory, whether it allows extended keywords, and whether it outputs warning messages. The pragma directives are preprocessed, which means that macros are substituted in a pragma directive.

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

Summary of pragma directives

The following table shows the pragma directives of the compiler:

Pragma directive	Description
`#pragma basic_template_matching`	Makes a template function fully memory-aware
`#pragma bitfields`	Controls the order of bitfield members
`#pragma constseg`	Places constant variables in a named segment
`#pragma context_handler`	Controls the enter and leave sequences of an interrupt function
`#pragma data_alignment`	Gives a variable a higher (more strict) alignment
`#pragma dataseg`	Places variables in a named segment
`#pragma diag_default`	Changes the severity level of diagnostic messages
`#pragma diag_error`	Changes the severity level of diagnostic messages
`#pragma diag_remark`	Changes the severity level of diagnostic messages
`#pragma diag_suppress`	Suppresses diagnostic messages
`#pragma diag_warning`	Changes the severity level of diagnostic messages
`#pragma include_alias`	Specifies an alias for an include file
`#pragma inline`	Inlines a function
`#pragma language`	Controls the IAR language extensions

Table 45: Pragma directives summary

Pragma directive	Description
`#pragma location`	Specifies the absolute address of a variable
`#pragma message`	Prints a message
`#pragma object_attribute`	Changes the definition of a variable or a function
`#pragma optimize`	Specifies type and level of optimization
`#pragma required`	Ensures that a symbol which is needed by another symbol is present in the linked output
`#pragma rtmodel`	Adds a runtime model attribute to the module
`#pragma segment`	Declares a segment name to be used by intrinsic functions
`#pragma type_attribute`	Changes the declaration and definitions of a variable or function
`#pragma vector`	Specifies the vector of an interrupt function

Table 45: Pragma directives summary (Continued)

Note: For portability reasons, some old-style pragma directives are recognized but will give a diagnostic message. It is important to be aware of this if you need to port existing code that contains any of those pragma directives. For additional information, see the *IAR Embedded Workbench Migration Guide for HCS12*.

Descriptions of pragma directives

This section gives detailed information about each pragma directive.

All pragma directives using = for value assignment should be entered like:

```
#pragma pragmaname=pragmavalue
```

or

```
#pragma pragmaname = pragmavalue
```

#pragma basic_template_matching

```
#pragma basic_template_matching
```

Use this pragma directive in front of a template function declaration to make the function fully memory-aware, in the rare cases where this is useful. That template function will then match the template without the modifications described in *Templates and data memory attributes*, page 100.

Example

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

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

#pragma bitfields

```
#pragma bitfields={reversed|default}
```

The `#pragma bitfields` directive controls the order of bitfield members.

By default, the IAR C/C++ Compiler for HCS12 places bitfield members from the least significant bit to the most significant bit in the container type. Use the `#pragma bitfields=reversed` directive to place the bitfield members from the most significant to the least significant bit. This setting remains active until you turn it off again with the `#pragma bitfields=default` directive.

#pragma constseg

The `#pragma constseg` directive places constant variables in a named segment. Use the following syntax:

```
#pragma constseg=MY_CONSTANTS
const int factorySettings[] = {42, 15, -128, 0};
#pragma constseg=default
```

The segment name cannot be a predefined segment; see the chapter *Segment reference* for more information.

The memory in which the segment resides is optionally specified using the following syntax:

```
#pragma constseg=__data8 MyOtherSeg
```

All constants defined following this directive will be placed in the segment `MyOtherSeg` and accessed using data8 addressing.

#pragma context_handler

```
#pragma context_handler=prefix
```

Use this pragma directive in front of an interrupt function declaration. The pragma directive causes a library call to the routines *prefix*`_PROLOGUE` and *prefix*`_EPILOGUE` at the function enter and leave sequence, respectively. Implement these library routines, for example, to make the interrupt routine use a different stack, which could save memory in certain situations. An example of such a situation is an operating system with one stack per process.

Example

```
#pragma context_handler=switch_stack
__interrupt void myInterrupt(void)
{
...
}
```

This will insert a call to the subroutine `switch_stack_PROLOGUE` before the first statement in the interrupt routine. After the final statement at the end of the interrupt routine, there will be a call to the subroutine `switch_stack_EPILOGUE`.

`#pragma data_alignment`

`#pragma data_alignment=`*`expression`*

Use this pragma directive to give a variable a higher (more strict) alignment than it would otherwise have. It can be used on variables with static and automatic storage duration.

The value of the constant *`expression`* must be a power of two (1, 2, 4, etc.).

When you use `#pragma data_alignment` on variables with automatic storage duration, there is an upper limit on the allowed alignment for each function, determined by the calling convention used.

`#pragma dataseg`

The `#pragma dataseg` directive places variables in a named segment. Use the following syntax:

```
#pragma dataseg=MY_SEGMENT
__no_init char myBuffer[1000];
#pragma dataseg=default
```

The segment name cannot be a predefined segment, see the chapter *Segment reference* for more information. The variable `myBuffer` will not be initialized at startup, and can for this reason not have any initializer.

The memory in which the segment resides is optionally specified using the following syntax:

```
#pragma dataseg=__data8 MyOtherSeg
```

All variables in `MyOtherSeg` will be accessed using data8 addressing.

#pragma diag_default

```
#pragma diag_default=tag,tag,...
```

Changes the severity level back to default, or as defined on the command line for the diagnostic messages with the specified tags. For example:

```
#pragma diag_default=Pe117
```

See the chapter *Diagnostics* for more information about diagnostic messages.

#pragma diag_error

```
#pragma diag_error=tag,tag,...
```

Changes the severity level to `error` for the specified diagnostics. For example:

```
#pragma diag_error=Pe117
```

See the chapter *Diagnostics* for more information about diagnostic messages.

#pragma diag_remark

```
#pragma diag_remark=tag,tag,...
```

Changes the severity level to `remark` for the specified diagnostics. For example:

```
#pragma diag_remark=Pe177
```

See the chapter *Diagnostics* for more information about diagnostic messages.

#pragma diag_suppress

```
#pragma diag_suppress=tag,tag,...
```

Suppresses the diagnostic messages with the specified tags. For example:

```
#pragma diag_suppress=Pe117,Pe177
```

See the chapter *Diagnostics* for more information about diagnostic messages.

#pragma diag_warning

```
#pragma diag_warning=tag,tag,...
```

Changes the severity level to `warning` for the specified diagnostics. For example:

```
#pragma diag_warning=Pe826
```

See the chapter *Diagnostics* for more information about diagnostic messages.

#pragma include_alias

```
#pragma include_alias "orig_header" "subst_header"
#pragma include_alias <orig_header> <subst_header>
```

The `#pragma include_alias` directive makes it possible to provide an alias for a header file. This is useful for substituting one header file with another, and for specifying an absolute path to a relative file.

The parameter *subst_header* is used for specifying an alias for *orig_header*. This pragma directive must appear before the corresponding `#include` directives and `subst_header` must match its corresponding `#include` directive exactly.

Example

```
#pragma include_alias <stdio.h> <C:\MyHeaders\stdio.h>
#include <stdio.h>
```

This example will substitute the relative file `stdio.h` with a counterpart located according to the specified path.

`#pragma inline`

```
#pragma inline[=forced]
```

The `#pragma inline` directive advises the compiler that the function whose declaration follows immediately after the directive should be inlined—that is, expanded into the body of the calling function. Whether the inlining actually takes place is subject to the compiler's heuristics.

This is similar to the C++ keyword `inline`, but has the advantage of being available in C code.

Specifying `#pragma inline=forced` disables the compiler's heuristics and forces the inlining. If the inlining fails for some reason, for example if it cannot be used with the function type in question (like `printf`), an error message is emitted.

`#pragma language`

```
#pragma language={extended|default}
```

The `#pragma language` directive is used for turning on the IAR language extensions or for using the language settings specified on the command line:

`extended`	Turns on the IAR language extensions and turns off the `--strict_ansi` command line option.
`default`	Uses the settings specified on the command line.

`#pragma location`

```
#pragma location=address
```

The `#pragma location` directive specifies the location—the absolute address—of the variable whose declaration follows the pragma directive. For example:

```
#pragma location=0xFF20
char PORT1; /* PORT1 is located at address 0xFF20 */
```

The directive can also take a string specifying the segment placement for either a variable or a function, for example:

```
#pragma location="foo"
```

For additional information and examples, see *Located data*, page 40.

#pragma message

```
#pragma message(message)
```

Makes the compiler print a message on `stdout` when the file is compiled. For example:

```
#ifdef TESTING
#pragma message("Testing")
#endif
```

#pragma object_attribute

```
#pragma object_attribute=keyword
```

The `#pragma object_attribute` directive can be used for specifying IAR-specific *object attribute*s, which are not part of the ISO/ANSI C language standard. Note however, that a given object attribute may not be applicable to all kind of objects. For a list of all supported object attributes, see *Object attributes*, page 132.

The `#pragma object_attribute` directive affects the definition of the identifier that follows immediately after the directive. The object is modified, not its type.

Example

In the following example, the variable `bar` is placed in the non-initialized segment:

```
#pragma object_attribute=__no_init
char bar;
```

Unlike the directive `#pragma type_attribute` that specifies the storing and accessing of a variable, it is not necessary to specify an object attribute in declarations. The following example declares `bar` without a `#pragma object_attribute`:

```
__no_init char bar;
```

#pragma optimize

```
#pragma optimize=token_1 token_2 token_3
```

where *token_n* is one of the following:

`s`	Optimizes for speed
`z`	Optimizes for size
`2|none|3|low|6|medium|9|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

The `#pragma optimize` directive is used for decreasing the optimization level, or for turning off some specific optimizations. This pragma directive only affects the function that follows immediately after the directive.

Note that it is not possible to optimize for speed and size at the same time. Only one of the `s` and `z` tokens can be used. It is also not possible to use macros embedded in this pragma directive. Any such macro will not get expanded by the preprocessor.

Note: If you use the `#pragma optimize` directive to specify an optimization level that is higher than the optimization level you specify using a compiler option, the pragma directive is ignored.

Example

```
#pragma optimize=s 9
int small_and_used_often()
{
    ...
}

#pragma optimize=z 9
int big_and_seldom_used()
{
    ...
}
```

#pragma required

`#pragma required=`*`symbol`*

Use the `#pragma required` directive to ensure that a symbol which is needed by another symbol is present in the linked output. The *`symbol`* can be any statically linked function or variable, and the pragma directive must be placed immediately before a symbol definition.

Use the directive if the requirement for a symbol is not otherwise visible in the application, for example, if a variable is only referenced indirectly through the segment it resides in.

Example

```
void * const myvar_entry @ "MYSEG" = &myvar;
...
#pragma required=myvar_entry
long myvar;
```

#pragma rtmodel

```
#pragma rtmodel="key","value"
```

Use the `#pragma rtmodel` directive to add a runtime model attribute to a module. Use a text string to specify *key* and *value*.

This pragma directive is useful to enforce consistency between modules. All modules that are linked together and define the same runtime attribute key must have the same value for the corresponding key value, or the special value `*`. Using the special value `*` is equivalent to not defining the attribute at all. It can, however, be useful to state explicitly that the module can handle any runtime model.

A module can have several runtime model definitions.

Note: The predefined compiler runtime model attributes start with a double underscore. In order to avoid confusion, this style must not be used in the user-defined attributes.

Example

```
#pragma rtmodel="I2C","ENABLED"
```

The linker will generate an error if a module that contains this definition is linked with a module that does not have the corresponding runtime model attributes defined.

For more information about runtime model attributes and module consistency, see *Checking module consistency*, page 70.

#pragma segment

```
#pragma segment="segment" [mematttr] [align]
```

The `#pragma segment` directive declares a segment name that can be used by the intrinsic functions `__segment_begin` and `__segment_end`. Each segment may only be declared once. All segment declarations for a specific segment must have the same memory type attribute and alignment.

The optional memory attribute *mematttr* will be used in the return type of the intrinsic function. The optional parameter *align* can be specified to align the segment part. The value must be a constant integer expression to the power of two.

Example

```
#pragma segment="MYSEG" __data8 4
```

See also *__segment_begin*, page 200.

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

#pragma type_attribute

```
#pragma type_attribute=keyword
```

The `#pragma type_attribute` directive can be used for specifying IAR-specific *type 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. For a list of all supported type attributes, see *Type and object attributes*, page 131.

The `#pragma type_attribute` directive affects the declaration of the identifier, the next variable, or the next function that follows immediately after the pragma directive.

Example

In the following example, even though IAR-specific type attributes are used, the application can still be compiled by a different compiler. First, a `typedef` is declared; a `char` object with the memory attribute `__data8` is defined as `MyCharInData8`. Then a pointer is declared; the pointer is located in data16 memory and it points to a `char` object that is located in data8 memory.

```
#pragma type_attribute=__data8
typedef char MyCharInData8;
#pragma type_attribute=__data16
MyCharInData8 * ptr;
```

The following declarations, which use extended keywords, are equivalent. See the chapter *Extended keywords* for more details.

```
char __data8 * __data16 ptr;
```

#pragma vector

```
#pragma vector=vector1[, vector2, vector3, ...]
```

The `#pragma vector` directive specifies the vector(s) of an interrupt function whose declaration follows the pragma directive.

Example

```
#pragma vector=0xFFF4
__interrupt void my_handler(void);
```

The preprocessor

This chapter gives a brief overview of the preprocessor, including reference information about the different preprocessor directives, symbols, and other related information.

Overview of the preprocessor

The preprocessor of the IAR C/C++ Compiler for HCS12 adheres to the ISO/ANSI standard. The compiler also makes the following preprocessor-related features available to you:

- **Predefined preprocessor symbols.** These symbols allow you to inspect the compile-time environment, for example the time and date of compilation. Some of the symbols take arguments and perform more advanced operations than just inspecting the compile-time environment. For details, see *Predefined symbols*, page 189.
- **User-defined preprocessor symbols.** Use the option `-D` to define your own preprocessor symbols, see *-D*, page 151.
- **Preprocessor extensions.** There are several preprocessor extensions, for example many pragma directives; for more information, see the chapter *Pragma directives* in this guide. For information about other extensions, see *Preprocessor extensions*, page 195.
- **Preprocessor output.** Use the option `--preprocess` to direct preprocessor output to a named file, see *--preprocess*, page 166.
- **Implementation-defined behavior.** Some parts listed by the ISO/ANSI standard are implementation-defined, for example the character set used in the preprocessor directives and inclusion of bracketed and quoted filenames. To read more about this, see *Preprocessing directives*, page 214.

Predefined symbols

This section first summarizes all predefined symbols and then provides detailed information about each symbol.

SUMMARY OF PREDEFINED SYMBOLS

The following table summarizes the predefined symbols:

Predefined symbol	Identifies
`__ALIGNOF__()`	Accesses the alignment of an object
`__BASE_FILE__`	Identifies the name of the file being compiled. If the file is a header file, the name of the file that includes the header file is identified.
`__CODE_MODEL__`	Identifies the code model in use
`__CPU__`	Identifies the CPU in use
`__cplusplus`	Determines whether the compiler runs in C++ mode*
`__DATE__`	Determines the date of compilation*
`__embedded_cplusplus`	Determines whether the compiler runs in C++ mode*
`__FILE__`	Identifies the name of the file being compiled*
`__func__`	Expands into a string with the function name as context
`__FUNCTION__`	Expands into a string with the function name as context
`__IAR_SYSTEMS_ICC__`	Identifies the IAR compiler platform
`__ICCHCS12__`	Identifies the IAR C/C++ Compiler for HCS12
`__LINE__`	Determines the current source line number*
`__LITTLE_ENDIAN__`	Identifies the byte order in use
`_Pragma()`	Can be used in preprocessor defines and has the equivalent effect as the pragma directive
`__PRETTY_FUNCTION__`	Expands into a string with the function name, including parameter types and return type, as context
`__STDC__`	Identifies ISO/ANSI Standard C*
`__STDC_VERSION__`	Identifies the version of ISO/ANSI Standard C in use*
`__TID__`	Identifies the target processor of the IAR compiler in use
`__TIME__`	Determines the time of compilation*
`__VER__`	Identifies the version number of the IAR compiler in use

Table 46: Predefined symbols summary

*** This symbol is required by the ISO/ANSI standard.**

DESCRIPTIONS OF PREDEFINED SYMBOLS

The following section gives reference information about each predefined symbol.

`__ALIGNOF__()` The `__ALIGNOF__` operator is used to access the alignment of an object. It takes one of two forms:

- `__ALIGNOF__(`*type*`)`
- `__ALIGNOF__(`*expression*`)`

In the second form, the expression is not evaluated.

`__BASE_FILE__` Use this symbol to identify which file is currently being compiled. This symbol expands to the name of that file, unless the file is a header file. In that case, the name of the file that includes the header file is identified.

See also, *__FILE__*, page 192.

`__CODE_MODEL__` Use this symbol to identify the used code model.

The `__CODE_MODEL__` symbol is set to one of the predefined symbols `__CODE_MODEL_NORMAL__` or `__CODE_MODEL_BANKED__` for the normal and banked code models, respectively.

Example

```
#if __CODE_MODEL__==__CODE_MODEL_NORMAL__
int my_array[10];
#else
int my_array[20];
#endif
```

`__CPU__` Identifies the CPU used. In the current version of the compiler, this symbol has the value 1.

This symbol is provided for compatibility with future versions of the compiler in case they should support more than one core.

__cplusplus
This predefined symbol expands to the number 199711L when the compiler runs in any of the C++ modes. When the compiler runs in ISO/ANSI C mode, the symbol is undefined.

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.

__DATE__
Use this symbol to identify when the file was compiled. This symbol expands to the date of compilation, which is returned in the form "Mmm dd yyyy", for example "Jan 30 2002".

__embedded_cplusplus
This predefined symbol expands to the number 1 when the compiler runs in any of the C++ modes. When the compiler runs in ISO/ANSI C mode, the symbol is undefined.

__FILE__
Use this symbol to identify which file is currently being compiled. This symbol expands to the name of that file.

See also, *__BASE_FILE__*, page 191.

__func__, __FUNCTION__
Use one of these symbols inside a function body to make it expand into a string with the function name as context. This is useful for assertions and other trace utilities. These symbols require that language extensions are enabled, see *-e*, page 156.

See also, *__PRETTY_FUNCTION__*, page 193.

__IAR_SYSTEMS_ICC__
This predefined symbol expands to a number that identifies the IAR compiler platform. The current identifier is 6. Note that the number could be higher in a future version of the product.

This symbol can be tested with #ifdef to detect whether the code was compiled by a compiler from IAR Systems.

`__ICCHCS12__` This predefined symbol expands to the number 1 when the code is compiled with the IAR C/C++ Compiler for HCS12.

`__LINE__` This predefined symbol expands to the current line number of the file currently being compiled.

`__LITTLE_ENDIAN__` This predefined symbol expands to the number 0 when the code is compiled with the little-endian byte order format, and 1 when the code is compiled with the big-endian byte order format. The HCS12 microcontroller uses big endian.

`_Pragma()` The preprocessor operator `_Pragma` can be used in defines and has the equivalent effect of the `#pragma` directive. The syntax is:

```
_Pragma("string")
```

where *`string`* follows the syntax for the corresponding pragma directive. For example:

```
#if NO_OPTIMIZE
  #define NOOPT _Pragma("optimize=2")
#else
  #define NOOPT
#endif
```

See the chapter *Pragma directives*.

Note: The `-e` option—enable language extensions—is not required.

`__PRETTY_FUNCTION__` Use this symbol inside a function body to make it expand into a string, with the function name including parameter types and return type as context. The result might, for example, look like this:

```
"void func(char)"
```

This symbol is useful for assertions and other trace utilities. These symbols require that language extensions are enabled, see *-e*, page 156.

See also, *__func__, __FUNCTION__*, page 192.

__STDC__ This predefined symbol expands to the number 1. This symbol can be tested with `#ifdef` to detect whether the compiler in use adheres to ISO/ANSI C.

__STDC_VERSION__ ISO/ANSI C and version identifier.

This predefined symbol expands to `199409L`.

Note: This predefined symbol does not apply in EC++ mode.

__TID__ Target identifier for the IAR C/C++ Compiler for HCS12.

Expands to the target identifier which contains the following parts:

- A one-bit intrinsic flag (`i`) which is reserved for use by IAR
- A target-identifier (`t`) unique for each IAR compiler. For the HCS12 microcontroller, the target identifier is `0x21`
- A value (`m`) reserved for memory model in IAR 68HC12 IAR C Compiler. For the HCS12 microcontroller, the value is `2` for the Banked code model and `1` for the Normal code model.

The __TID__value is constructed as:

```
((i << 15) | (t << 8) | m)
```

You can extract the values as follows:

```
i = (__TID__ >> 15) & 0x01;    /* intrinsic flag */

t = (__TID__ >> 8) & 0x7F;     /* target identifier */

m = __TID__ & 0x0F;            /* code model */
```

To find the value of the target identifier for the current compiler, execute:

```
printf("%ld",(__TID__ >> 8) & 0x7F)
```

Note: The use of __TID__ is not recommended. We recommend that you use the symbols __ICCHCS12__ and __CODE_MODEL__ instead.

__TIME__ Current time.

Expands to the time of compilation in the form `hh:mm:ss`.

__VER__ Compiler version number.

Expands to an integer representing the version number of the compiler. The value of the number is calculated in the following way:

```
(100 * the major version number + the minor version number)
```

Example

The example below prints a message for version 3.34.

```
#if __VER__ == 334
#pragma message("Compiler version 3.34")
#endif
```

In this example, `3` is the major version number and `34` is the minor version number.

Preprocessor extensions

The following section gives reference information about the extensions that are available in addition to the pragma directives and ISO/ANSI directives.

#warning message Use this preprocessor directive to produce messages. Typically this is useful for assertions and other trace utilities, similar to the way the ISO/ANSI standard `#error` directive is used. The syntax is:

```
#warning message
```

where *`message`* can be any string.

__VA_ARGS__ Variadic macros are the preprocessor macro equivalents of `printf` style functions.

Syntax

```
#define P(...)        __VA_ARGS__
#define P(x,y,...)    x + y + __VA_ARGS__
```

Here, `__VA_ARGS__` will contain all variadic arguments concatenated, including the separating commas.

Example

```
#if DEBUG
  #define DEBUG_TRACE(...) printf(S,__VA_ARGS__)
#else
  #define DEBUG_TRACE(...)
```

```
#endif
...
DEBUG_TRACE("The value is:%d\n",value);
```

will result in:

```
printf("The value is:%d\n",value);
```

Intrinsic functions and extended operators

This chapter gives reference information about the intrinsic functions and the extended operators.

The intrinsic functions provide direct access to low-level processor operations and can be very useful in, for example, time-critical routines. The intrinsic functions compile into inline code, either as a single instruction or as a short sequence of instructions.

Intrinsic functions summary

The following table summarizes the intrinsic functions:

Intrinsic function	Description
`__address_24_of`	Returns the full address of a data object located in banked area
`__disable_interrupt`	Disables interrupts
`__enable_interrupt`	Enables interrupts
`__get_ccr_register`	Returns the value of the `CCR` processor register
`__get_interrupt_state`	Returns the interrupt state
`__max8`	Returns the larger of two arguments
`__max16`	Returns the larger of two arguments
`__min8`	Returns the smaller of two arguments
`__min16`	Returns the smaller of two arguments
`__no_operation`	Generates a `NOP` instruction
`__op_code`	Inserts a constant into the instruction stream
`__set_ccr_register`	Set the value of the `CCR` processor register
`__set_interrupt_state`	Restores the interrupt state
`__software_interrupt`	Inserts a software interrupt

Table 47: Intrinsic functions summary

Intrinsic function	Description
`__stop_CPU`	Inserts the `STOP` instruction
`__wait_for_interrupt`	Inserts a `WAI` instruction

Table 47: Intrinsic functions summary (Continued)

To use intrinsic functions in an application, include the header file `intrinsics.h`.

Extended operators

The following table summarizes the extended operators:

Intrinsic function	Description
`@`	Controls the storage of variables and functions
`asm`, `__asm`	Inserts an assembler instruction
`__segment_begin`	Returns the start address of a segment
`__segment_end`	Returns the end address of a segment

Table 48: Extended operators summary

To use extended operators in your application, enable language extensions, see *-e*, page 156.

Descriptions of intrinsic functions and extended operators

The following section gives reference information about each intrinsic function.

@

The `@` operator can be used for placing global and static variables at absolute addresses. The syntax can also be used for placing variables and functions in named segments.

For more information about the `@` operator, see *Efficient usage of segments and memory*, page 42.

__address_24_of

```
unsigned long __address_24_of(void *memory_location);
```

This function takes a pointer to a memory location in a banked area as an argument and returns the address of the memory location, in other words, the function obtains the 24-bit address of a banked data object.

For information about how to use this function, see *Banked constant data*, page 118.

`asm, __asm`

The `asm` and `__asm` operators both insert an assembler instruction. However, when compiling C source code, the `asm` operator is not available when the option `--strict_ansi` is used. The `__asm` operator is always available.

Note: Not all assembler directives can be inserted using this operator.

Syntax

```
asm ("string");
```

The string can be a valid assembler instruction or an assembler directive, but not a comment. You can write several consecutive inline assembler instructions, for example:

```
asm ("Label:      nop\n"
     "            jmp Label");
```

where `\n` (new line) separates each new assembler instruction. Note that you can define and use local labels in inline assembler instructions.

For more information about inline assembler, see *Mixing C and assembler*, page 79.

`__disable_interrupt`

```
void __disable_interrupt(void);
```

Disables interrupts by inserting the `SEI` instruction.

`__enable_interrupt`

```
void __enable_interrupt(void);
```

Enables interrupts by inserting the `CLI` instruction.

`__get_ccr_register`

```
istate_t __get_ccr_register (void);
```

Returns the value of the `CCR` processor register.

`__get_interrupt_state`

```
istate_t __get_interrupt_state (void);
```

Returns the global interrupt state from the `CCR` register.The return value can be used as a parameter for the `__set_interrupt_state` function, which will restore the interrupt state.

`__max8`

```
unsigned char __max8 (unsigned char, unsigned char);
```

Returns the larger of the two 8-bit arguments.

`__max16` `unsigned short __max16 (unsigned short, unsigned short);`

Returns the larger of the two 16-bit arguments.

`__min8` `unsigned char __min8 (unsigned char, unsigned char);`

Returns the smaller of the two 8-bit arguments.

`__min16` `unsigned short __min16 (unsigned short, unsigned short);`

Returns the smaller of the two 16-bit arguments.

`__no_operation` `void __no_operation(void);`

Generates a `NOP` instruction.

`__op_code` `void __op_code (unsigned char);`

Emits the 8-bit value into the instruction stream for the current function by inserting a `DC8` constant.

`__segment_begin` `void * __segment_begin(`*`segment`*`);`

Returns the address of the first byte of the named *`segment`*. The named *`segment`* must be a string literal that has been declared earlier with the `#pragma segment` directive. See *#pragma segment*, page 187.

If the segment was declared with a memory attribute *`memattr`*, the type of the `__segment_begin` function is pointer to *`memattr`* `void`. Otherwise, the type is a default pointer to `void`.

Example

```
#pragma segment="MYSEG" __data8
...
segment_start_address = __segment_begin("MYSEG");
```

Here, the type of the `__segment_begin` intrinsic function is `void __data8 *`.

Note: You must have enabled language extensions to use this extended operator.

__segment_end

```
void * __segment_end(segment);
```

Returns the address of the first byte *after* the named *segment*. The named *segment* must be a string literal that has been declared earlier with the `#pragma segment` directive. See *#pragma segment*, page 187.

If the segment was declared with a memory attribute *mematttr*, the type of the `__segment_end` function is pointer to *mematttr* `void`. Otherwise, the type is a default pointer to `void`.

Example

```
#pragma segment="MYSEG" __data88
...
segment_end_address = __segment_end("MYSEG");
```

Here, the type of the `__segment_end` intrinsic function is `void __data8 *`.

Note: You must have enabled language extensions to use this intrinsic function.

__set_ccr_register

```
istate_t __set_ccr_register (istate_t);
```

Sets the value of the `CCR` processor register. The return value is the previous state of the `CCR` register.

__set_interrupt_state

```
void __set_interrupt_state (istate_t);
```

Restores the interrupt state by setting the value returned by the `__get_interrupt_state` function.

__software_interrupt

```
void __software_interrupt (void);
```

Inserts a software interrupt.

__stop_CPU

```
void __stop_CPU (void);
```

Inserts the `STOP` instruction.

__wait_for_interrupt

```
void __wait_for_interrupt(void);
```

Inserts a `WAI` instruction.

Library functions

This chapter gives an introduction to the C and C++ library functions. It also lists the header files used for accessing library definitions.

Introduction

The IAR C/C++ Compiler for HCS12 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, multibytes, et cetera.

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

For detailed information about the library functions, see the online documentation supplied with the product. There is also keyword reference information for the DLIB library functions. To obtain reference information for a function, select the function name in the editor window and press F1.

For additional information about library functions, see the chapter *Implementation-defined behaviour* in this guide.

HEADER FILES

Your application program gains access to library definitions through header files, which it incorporates using the `#include` directive. The definitions are divided into a number of different header files, each covering a particular functional area, letting you include just those that are required.

It is essential to include the appropriate header file before making any reference to its definitions. Failure to do so can cause the call to fail during execution, or generate error or warning messages at compile time or link time.

LIBRARY OBJECT FILES

Most of the library definitions can be used without modification, that is, directly from the library object files that are supplied with the product. For information about how to choose a runtime library, see *Basic settings for project configuration*, page 5. The linker will include only those routines that are required—directly or indirectly—by your application.

REENTRANCY

A function that can be simultaneously invoked in the main application and in any number of interrupts is reentrant. A library function that uses statically allocated data is therefore not reentrant. Most parts of the DLIB library are reentrant, but the following functions and parts are not reentrant:

`atexit`	Needs static data
heap functions	Need static data for memory allocation tables
`strerror`	Needs static data
`strtok`	Designed by ISO/ANSI standard to need static data
`I/O`	Every function that uses 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 *The DLIB runtime environment* chapter in this guide.
- Runtime support libraries; for example, low-level floating-point routines.

C HEADER FILES

This section lists the header files specific to the DLIB library C definitions. Header files may additionally contain target-specific definitions; these are documented in the chapter *IAR language extensions*.

The following table lists the C header files:

Header file	Usage
`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
`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
`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 49: 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 streams classes that manipulate external files
`iomanip`	Declaring several I/O streams 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 streams classes before they are necessarily defined
`iostream`	Declaring the I/O streams 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 streams classes that manipulate string containers
`stdexcept`	Defining several classes useful for reporting exceptions
`streambuf`	Defining classes that buffer I/O streams operations
`string`	Defining a class that implements a string container
`strstream`	Defining several I/O streams classes that manipulate in-memory character sequences

Table 50: 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 streams manipulators that take an argument
`iostream.h`	Declaring the I/O streams objects that manipulate the standard streams
`new.h`	Declaring several functions that allocate and free storage

Table 51: Additional Embedded C++ header files—DLIB

Extended Embedded C++ standard template library

The following table lists the Extended Embedded C++ standard template library (STL) header files:

Header file	Description
`algorithm`	Defines several common operations on sequences

Table 52: Standard template library header files

Header file	Description
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 52: Standard template library header files (Continued)

Using standard C libraries in C++

The C++ library works in conjunction with 15 of the header files from the standard C library, sometimes with small alterations. The header files come in two forms—new and traditional—for example, `cassert` and `assert.h`.

The following table shows the new header files:

Header file	Usage
cassert	Enforcing assertions when functions execute
cctype	Classifying characters
cerrno	Testing error codes reported by library functions
cfloat	Testing floating-point type properties
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

Table 53: New standard C header files—DLIB

Header file	Usage
cstdarg	Accessing a varying number of arguments
cstddef	Defining several useful types and macros
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

Table 53: New standard C header files—DLIB (Continued)

Implementation-defined behavior

This chapter describes how the IAR C/C++ Compiler for HCS12 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 IAR C/C++ Compiler for HCS12 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, see *Implementation of system startup code*, page 74.

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 65.

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 65.

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 126, 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 127, 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 130, for information about `size_t`.

Conversion from/to pointers (6.3.4)

See *Casting between non-related types*, page 129, for information about casting of data pointers and function pointers.

ptrdiff_t (6.3.6, 7.1.1)

See *ptrdiff_t*, page 130, 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 126, for information about the alignment requirement for data objects.

Sign of 'plain' bitfields (6.5.2.1)

A 'plain' `int` bitfield is treated as a signed `int` bitfield. All integer types are allowed as bitfields.

Allocation order of bitfields within a unit (6.5.2.1)

Bitfields are allocated within an integer from least-significant to most-significant bit.

Can bitfields straddle a storage-unit boundary (6.5.2.1)

Bitfields cannot straddle a storage-unit boundary for the chosen bitfield integer type.

Integer type chosen to represent enumeration types (6.5.2.2)

The chosen integer type for a specific enumeration type depends on the enumeration constants defined for the enumeration type. The chosen integer type is the smallest possible.

QUALIFIERS

Access to volatile objects (6.5.3)

Any reference to an object with volatile qualified type is an access.

DECLARATORS

Maximum numbers of declarators (6.5.4)

The number of declarators is not limited. The number is limited only by the available memory.

STATEMENTS

Maximum number of case statements (6.6.4.2)

The number of case statements (case values) in a switch statement is not limited. The number is limited only by the available memory.

PREPROCESSING DIRECTIVES

Character constants and conditional inclusion (6.8.1)

The character set used in the preprocessor directives is the same as the execution character set. The preprocessor recognizes negative character values if a 'plain' character is treated as a `signed` character.

Including bracketed filenames (6.8.2)

For file specifications enclosed in angle brackets, the preprocessor does not search directories of the parent files. A parent file is the file that contains the `#include` directive. Instead, it begins by searching for the file in the directories specified on the compiler command line.

Including quoted filenames (6.8.2)

For file specifications enclosed in quotes, the preprocessor directory search begins with the directories of the parent file, then proceeds through the directories of any grandparent files. Thus, searching begins relative to the directory containing the source file currently being processed. If there is no grandparent file and the file has not been found, the search continues as if the filename was enclosed in angle brackets.

Character sequences (6.8.2)

Preprocessor directives use the source character set, with the exception of escape sequences. Thus, to specify a path for an include file, use only one backslash:

```
#include "mydirectory\myfile"
```

Within source code, two backslashes are necessary:

```
file = fopen("mydirectory\\myfile","rt");
```

Recognized pragma directives (6.8.6)

The following pragma directives are recognized:

```
alignment
ARGSUSED
baseaddr
bitfields
can_instantiate
codeseg
constseg
context_handler
cspy_support
data_alignment
dataseg
define_type_info
diag_default
diag_error
diag_remark
diag_suppress
diag_warning
do_not_instantiate
function
```

```
hdrstop
include_alias
inline
instantiate
language
location
memory
message
module_name
none
no_pch
NOTREACHED
object_attribute
once
optimize
__printf_args
public_equ
required
rtmodel
__scanf_args
segment
system_include
type_attribute
VARARGS
vector
warnings
```

For a description of the pragma directives, see the chapter *Pragma directives*.

Default __DATE__ and __TIME__ (6.8.8)

The definitions for `__TIME__` and `__DATE__` are always available.

IAR 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 68.

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 64.

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 64.

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 64.

%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 67.

system() (7.10.4.5)

How the command processor works depends on how you have implemented the `system` function. See *Environment interaction*, page 67.

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 54: Message returned by strerror()—IAR DLIB library

The time zone (7.12.1)

The local time zone and daylight savings time implementation is described in *Time*, page 68.

clock() (7.12.2.1)

From where the system clock starts counting depends on how you have implemented the `clock` function. See *Time*, page 68.

IAR language extensions

This chapter describes IAR language extensions to the ISO/ANSI standard for the C programming language. All extensions can also be used for the C++ programming language.

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

See the compiler options -e on page 156 and --strict_ansi on page 170 for information about how to enable and disable language extensions from the command line.

Why should language extensions be used?

By using language extensions, you gain full control over the resources and features of the target microcontroller, and can thereby fine-tune your application.

If you want to use the source code with different compilers, note that language extensions may require minor modifications before the code can be compiled. A compiler typically supports microcontroller-specific language extensions as well as vendor-specific ones.

Descriptions of language extensions

This section gives an overview of available language extensions.

Memory, type, and object attributes

Entities such as variables and functions may be declared with memory, type, and object attributes. The syntax follows the syntax for qualifiers—such as `const`—but the semantics is different.

- A memory attribute controls the placement of the entity. There can be only one memory attribute.
- A type attribute controls aspects of the object visible to the surrounding context. There can be many different type attributes, and they must be included when the object is declared.
- An object attribute only has to be specified at the definition, but not at the declaration, of an object. The object attribute does not affect the object interface.

See the *Extended keywords* chapter for a complete list of attributes.

Placement at an absolute address or in a named segment

The operator `@` or the directive `#pragma location` can be used for placing a variable at an absolute address, or placing a variable or function in a named segment. The named segment can either be a predefined segment, or a user-defined segment.

Note: Placing variables and functions into named segments can also be done using the pragma directives `#pragma codeseg`, `#pragma constseg`, and `#pragma dataseg`.

Example 1

```
__no_init int x @ 0x1000;
```

An absolute declared variable cannot have an initializer, which means the variable must also be `__no_init` or `const` declared.

Example 2

```
void test(void) @ "MYOWNSEGMENT"
{
  ...
```

Note that all segments, both user-defined and predefined, must be assigned a location, which is done in the linker command file.

_Pragma

For information about the preprocessor operator `_Pragma()`, see *_Pragma()*, page 193.

Variadic macros

Variadic macros are the preprocessor macro equivalents of `printf` style functions. For more information, see *__VA_ARGS__*, page 195.

Inline functions

The `inline` keyword can be used on functions. It works just like the C++ keyword `inline` and the `#pragma inline` directive.

Mixing declarations and statements

It is possible to mix declarations and statements within the same scope.

Declaration in for loops

It is possible to have a declaration in the initialization expression of a `for` loop.

Inline assembler

Inline assembler can be used for inserting assembler instructions in the generated function.

The syntax for inline assembler is:

```
asm("MOVB #1,PORTA");
```

In strict ISO/ANSI mode, the use of inline assembler is disabled.

For more details about inline assembler, see *Mixing C and assembler*, page 79.

C++ style comments

C++ style comments are accepted. A C++ style comment starts with the character sequence `//` and continues to the end of the line. For example:

```
// The length of the bar, in centimeters.
int length;
```

__ALIGNOF__

For information about alignment, see *Alignment*, page 125, and *__ALIGNOF__()*, page 191.

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++.

Anonymous structs and unions

C++ includes a feature named anonymous unions. The IAR C/C++ Compiler for HCS12 allows a similar feature for both structs and unions.

An anonymous structure type (that is, one without a name) defines an unnamed object (and not a type) whose members are promoted to the surrounding scope. External anonymous structure types are allowed.

For example, the structure `str` in the following example contains an anonymous union. The members of the union are accessed using the names `b` and `c`, for example `obj.b`.

Without anonymous structure types, the union would have to be named—for example `u`—and the member elements accessed using the syntax `obj.u.b`.

```
struct str
{
  int a;
  union
  {
    int b;
    int c;
  };
};

struct str obj;
```

Bitfields and non-standard types

In ISO/ANSI C, a bitfield must be of the type `int` or `unsigned int`. Using IAR language extensions, any integer types and enums may be used.

For example, in the following structure an `unsigned char` is used for holding three bits. The advantage is that the struct will be smaller.

```
struct str
{
  unsigned char bitOne   : 1;
  unsigned char bitTwo   : 1;
  unsigned char bitThree : 1;
};
```

This matches G.5.8 in the appendix of the ISO standard, *ISO Portability Issues*.

Incomplete arrays at end of structs

The last element of a `struct` may be an incomplete array. This is useful because one chunk of memory can be allocated for the `struct` itself and for the array, regardless of the size of the array.

Note: The array may not be the only member of the `struct`. If that was the case, then the size of the `struct` would be zero, which is not allowed in ISO/ANSI C.

Example

```
struct str
{
  char a;
  unsigned long b[];
};

struct str * GetAStr(int size)
{
  return malloc(sizeof(struct str) +
                sizeof(unsigned long) * size);
}

void UseStr(struct str * s)
{
  s->b[10] = 0;
}
```

The array itself will not be included in the size of the struct.

Arrays of incomplete types

An array may have an incomplete `struct`, `union`, or `enum` type as its element type. The types must be completed before the array is used (if it is), or by the end of the compilation unit (if it is not).

Empty translation units

A translation unit (source file) is allowed to be empty, that is, it does not have to contain any declarations.

In strict ISO/ANSI mode, a warning is issued if the translation unit is empty.

Example

The following source file is only used in a debug build. (In a debug, build the `NDEBUG` preprocessor flag is undefined.) Since the entire contents of the file is conditionally compiled using the preprocessor, the translation unit will be empty when the application is compiled in release mode. Without this extension, this would be considered an error.

```
#ifndef NDEBUG

void PrintStatusToTerminal()
{
  /* Do something */
}
#endif
```

Comments at the end of preprocessor directives

This extension, which makes it legal to place text after preprocessor directives, is enabled, unless strict ISO/ANSI mode is used. This language extension exists to support compilation of old legacy code; we do *not* recommend that you write new code in this fashion.

Example

```
#ifdef FOO

  ... something ...

#endif FOO /* This is allowed but not recommended. */
```

Forward declaration of enums

The IAR Systems language extensions allow that you first declare the name of an `enum` and later resolve it by specifying the brace-enclosed list.

Extra comma at end of enum list

It is allowed to place an extra comma at the end of an `enum` list. In strict ISO/ANSI mode, a warning is issued.

Note: ISO/ANSI C allows extra commas in similar situations, for example after the last element of the initializers to an array. The reason is, that it is easy to get the commas wrong if parts of the list are moved using a normal cut-and-paste operation.

Example

```
enum
{
   kOne,
   kTwo,   /* This is now allowed. */
};
```

Missing semicolon at end of struct or union specifier

A warning is issued if the semicolon at the end of a `struct` or union specifier is missing.

NULL and void

In operations on pointers, a pointer to `void` is always implicitly converted to another type if necessary, and a null pointer constant is always implicitly converted to a null pointer of the right type if necessary. In ISO/ANSI C, some operators allow such things, while others do not allow them.

A label preceding a "}"

In ISO/ANSI C, a label must be followed by at least one statement. Therefore, it is illegal to place the label at the end of a block. In the IAR C/C++ Compiler for HCS12, a warning is issued.

To create a standard-compliant C program (so that you will not have to see the warning), you can place an empty statement after the label. An empty statement is a single `;` (semi-colon).

Example

```
void test()
{
  if (...) goto end;

   /* Do something */

  end:  /* Illegal at the end of block. */
}
```

Note: This also applies to the labels of switch statements.

The following piece of code will generate a warning:

```
switch (x)
{
case 1:
  ...;
  break;

default:
}
```

A good way to convert this into a standard-compliant C program is to place a `break;` statement after the `default:` label.

Empty declarations

An empty declaration (a semicolon by itself) is allowed, but a remark is issued (provided that remarks are enabled).

This is useful when preprocessor macros are used that could expand to nothing. Consider the following example. In a debug build, the macros `DEBUG_ENTER` and `DEBUG_LEAVE` could be defined to something useful.

However, in a release build they could expand into nothing, leaving the ; character in the code:

```
void test()
{
  DEBUG_ENTER();

  do_something();

  DEBUG_LEAVE();
}
```

Single value initialization

ISO/ANSI C requires that all initializer expressions of static arrays, `structs`, and `unions` are enclosed in braces.

Single-value initializers are allowed to appear without braces, but a warning is issued.

Example

In the IAR C/C++ Compiler for HCS12, the following expression is allowed:

```
struct str
{
  int a;
} x = 10;
```

Casting pointers to integers in static initializers

In an initializer, a pointer constant value may be cast to an integral type if the integral type is large enough to contain it.

In the following example, it is assumed that pointers to `__data8` and `__data16` are 8 and 16 bits, respectively. The first initialization is correct, because it is possible to cast the 8-bit address to an 8-bit `unsigned char` variable. However, it is illegal to use the 16-bit address of `b` as an initializer for an 8-bit value.

```
__data8 int a;
__data16 int b;

unsigned char ap = (unsigned char)&a;  /* Correct */
unsigned char bp = (unsigned char)&b;  /* Warning */
```

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.

Examples

`0x1p0` is `1`

`0xA.8p2` is `10.5*2^2`

Using the bool data type in C

To use the `bool` type in C source code, you must include the `stdbool.h` file. (The `bool` data type is supported by default in C++.)

Taking the address of a register variable

In ISO/ANSI C, it is illegal to take the address of a variable specified as a register variable.

The IAR C/C++ Compiler for HCS12 allows this, but a warning is issued.

Duplicated size and sign specifiers

Should the size or sign specifiers be duplicated (for example, `short short` or `unsigned unsigned`), an error is issued.

"long float" means "double"

`long float` is accepted as a synonym for `double`.

Repeated typedefs

Redeclarations of `typedef` that occur in the same scope are allowed, but a warning is issued.

Mixing pointer types

Assignment and pointer difference is allowed between pointers to types that are interchangeable but not identical; for example, `unsigned char *` and `char *`. This includes pointers to integral types of the same size. A warning is issued.

Assignment of a string constant to a pointer to any kind of character is allowed, and no warning will be issued.

Non-top level const

Assignment of pointers is allowed in cases where the destination type has added type qualifiers that are not at the top level (for example, `int **` to `int const **`). It is also allowed to compare and take the difference of such pointers.

Declarations in other scopes

External and static declarations in other scopes are visible. In the following example, the variable `y` can be used at the end of the function, even though it should only be visible in the body of the `if` statement. A warning is issued.

```
int test(int x)
{
  if (x)
  {
    extern int y;
    y = 1;
  }

  return y;
}
```

Non-lvalue arrays

A non-`lvalue` array expression is converted to a pointer to the first element of the array when it is used.

Diagnostics

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

Message format

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

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

where *filename* is the name of the source file in which the error was encountered, *linenumber* is the line number at which the compiler detected the error, *level* is the level of seriousness of the diagnostic, *tag* is a unique tag that identifies the diagnostic message, and *message* is a self-explanatory message, possibly several lines long.

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

Use the option `--diagnostics_tables` to list all possible compiler diagnostic messages in a named file.

Severity levels

The diagnostics are divided into different levels of severity:

Remark

A diagnostic message that is produced when the compiler finds a source code construct that can possibly lead to erroneous behavior in the generated code. Remarks are by default not issued, but can be enabled, see *--remarks*, page 167.

Warning

A diagnostic that is produced when the compiler finds a programming error or omission which is of concern, but not so severe as to prevent the completion of compilation. Warnings can be disabled by use of the command-line option `--no_warnings`, see page 165.

Error

A diagnostic that is produced when the compiler has found a construct which clearly violates the C or C++ language rules, such that code cannot be produced. An error will produce a non-zero exit code.

Fatal error

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

SETTING THE SEVERITY LEVEL

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

See *Options summary*, page 147, for a description of the compiler options that are available for setting severity levels.

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

INTERNAL ERROR

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

```
Internal error: message
```

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

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

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Symbols

Numerics