

MSP430 IAR C/C++ Compiler

Reference Guide

for Texas Instruments'

MSP430 Microcontroller Family

COPYRIGHT NOTICE

© Copyright 1996–2006 IAR Systems. All rights reserved.

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

DISCLAIMER

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

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

TRADEMARKS

IAR Systems, From Idea to Target, IAR Embedded Workbench, visualSTATE, IAR MakeApp and C-SPY are trademarks owned by IAR Systems AB.

Texas Instruments is a registered trademark of Texas Instruments Incorporated

Microsoft and Windows are registered trademarks of Microsoft Corporation.

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

EDITION NOTICE

Sixth edition: March 2006

Part number: C430-6

This guide applies to version 3.x of the MSP430 IAR Embedded Workbench®.

Brief contents

Tables	xv
Preface	xvii
Part 1. Using the compiler	1
Getting started	3
Data storage	11
Functions	15
Placing code and data	23
The DLIB runtime environment	39
The CLIB runtime environment	69
Assembler language interface	77
Using C++	93
Efficient coding for embedded applications	99
Part 2. Compiler reference	113
Compiler usage	115
Compiler options	121
Data representation	149
Compiler extensions	159
Extended keywords	171
Pragma directives	181
Intrinsic functions	197
The preprocessor	207

Library functions	213
Segment reference	223
Implementation-defined behavior	229
Index	243

Contents

Tables	xv
Preface	xvii
Who should read this guide	xvii
How to use this guide	xvii
What this guide contains	xviii
Other documentation	xix
Further reading	xix
Document conventions	xx
Typographic conventions	xx
Part I. Using the compiler	1
Getting started	3
IAR language overview	3
Supported MSP430 derivatives	4
Building applications—an overview	4
Compiling	4
Linking	4
Basic settings for project configuration	5
Core	6
Hardware multiplier	6
Position-independent code	6
Size of double floating-point type	7
Optimization for speed and size	7
Runtime environment	7
Special support for embedded systems	9
Extended keywords	9
Pragma directives	9
Predefined symbols	10
Special function types	10
Header files for I/O	10

Accessing low-level features	10
Data storage	11
Introduction	11
The stack and auto variables	12
Dynamic memory on the heap	13
Functions	15
Function-related extensions	15
Primitives for interrupts, concurrency, and OS-related programming	15
Interrupt functions	15
Monitor functions	18
C++ and special function types	21
Placing code and data	23
Segments and memory	23
What is a segment?	23
Placing segments in memory	24
Customizing the linker command file	25
Data segments	27
Static memory segments	27
The stack	29
The heap	30
Located data	31
Code segments	32
Startup code	32
Normal code	32
Interrupt functions for MSP430X	32
Interrupt vectors	33
C++ dynamic initialization	33
Controlling data and function placement in memory	33
Data placement at an absolute location	34
Data and function placement in segments	35

Verifying the linked result of code and data placement	37
Segment too long errors and range errors	37
Linker map file	37
The DLIB runtime environment	39
Introduction to the runtime environment	39
Runtime environment functionality	39
Library selection	40
Situations that require library building	40
Library configurations	41
Debug support in the runtime library	41
Using a prebuilt library	42
Customizing a prebuilt library without rebuilding	44
Choosing formatters for printf and scanf	44
Choosing printf formatter	45
Choosing scanf formatter	46
Overriding library modules	47
Building and using a customized library	48
Setting up a library project	48
Modifying the library functionality	49
Using a customized library	49
System startup and termination	50
System startup	51
System termination	51
Customizing system initialization	52
__low_level_init	52
Modifying the file cstartup.s43	53
Standard streams for input and output	53
Implementing low-level character input and output	53
Configuration symbols for printf and scanf	55
Customizing formatting capabilities	56
File input and output	56
Locale	57
Locale support in prebuilt libraries	57

Customizing the locale support	58
Changing locales at runtime	58
Environment interaction	59
Signal and raise	60
Time	60
Strtod	61
Assert	61
Hardware multiplier support	61
C-SPY Debugger runtime interface	62
Low-level debugger runtime interface	62
The debugger terminal I/O window	63
Checking module consistency	64
Runtime model attributes	64
Using runtime model attributes	65
Predefined runtime attributes	65
User-defined runtime model attributes	67
The CLIB runtime environment	69
Runtime environment	69
Input and output	70
Character-based I/O	70
Formatters used by printf and sprintf	71
Formatters used by scanf and sscanf	72
System startup and termination	73
System startup	73
System termination	73
Overriding default library modules	74
Customizing system initialization	74
C-SPY runtime interface	74
The debugger terminal I/O window	74
Termination	75
Checking module consistency	75

Assembler language interface	77
Mixing C and assembler	77
Intrinsic functions	77
Mixing C and assembler modules	78
Inline assembler	79
Calling assembler routines from C	80
Creating skeleton code	80
Compiling the code	81
Calling assembler routines from C++	82
Calling convention	83
Function declarations	83
C and C++ linkage	83
Preserved versus scratch registers	84
Function entrance	85
Function exit	87
Examples	87
Function directives	89
Calling functions	89
Call frame information	90
Using C++	93
Overview	93
Standard Embedded C++	93
Extended Embedded C++	94
Enabling C++ support	94
Feature descriptions	95
Classes	95
Functions	95
Templates	96
Variants of casts	96
Mutable	96
Namespace	96
The STD namespace	97
Using interrupts and EC++ destructors	97

C++ language extensions	97
Efficient coding for embedded applications	99
Taking advantage of the compilation system	99
Controlling compiler optimizations	100
Fine-tuning enabled transformations	101
Selecting data types and placing data in memory	103
Using efficient data types	103
Rearranging elements in a structure	104
Anonymous structs and unions	105
Writing efficient code	106
Saving stack space and RAM memory	107
Function prototypes	107
Integer types and bit negation	108
Protecting simultaneously accessed variables	109
Accessing special function registers	109
Non-initialized variables	110
Efficient switch statements	111
Part 2. Compiler reference	113
Compiler usage	115
Compiler invocation	115
Invocation syntax	115
Passing options to the compiler	116
Environment variables	116
Include file search procedure	116
Compiler output	117
Diagnostics	119
Message format	119
Severity levels	119
Setting the severity level	120
Internal error	120

Compiler options	121
Compiler options syntax	121
Types of options	121
Rules for specifying parameters	121
Options summary	124
Descriptions of options	126
Data representation	149
Alignment	149
Alignment in the MSP430 IAR C/C++ Compiler	150
Basic data types	150
Integer types	150
Floating-point types	152
Pointer types	153
Casting	154
Structure types	155
Alignment	155
General layout	155
Packed structure types	155
Type qualifiers	156
Declaring objects volatile	156
Declaring objects const	157
Data types in C++	157
Compiler extensions	159
Compiler extensions overview	159
Enabling language extensions	160
C language extensions	160
Important language extensions	161
Useful language extensions	162
Minor language extensions	166
Extended keywords	171
General syntax rules for extended keywords	171
Type attributes	171

Object attributes	173
Summary of extended keywords	174
Descriptions of extended keywords	175
Pragma directives	181
Summary of pragma directives	181
Descriptions of pragma directives	182
Intrinsic functions	197
Intrinsic functions summary	197
Descriptions of intrinsic functions	198
The preprocessor	207
Overview of the preprocessor	207
Descriptions of predefined preprocessor symbols	208
Descriptions of miscellaneous preprocessor extensions	210
Library functions	213
Introduction	213
Header files	213
Library object files	214
Reentrancy	214
IAR DLIB Library	214
C header files	215
C++ header files	216
Added C functionality	218
IAR CLIB Library	220
Library definitions summary	220
Segment reference	223
Summary of segments	223
Descriptions of segments	224
Implementation-defined behavior	229
Descriptions of implementation-defined behavior	229
Translation	229

Environment	230
Identifiers	230
Characters	230
Integers	232
Floating point	232
Arrays and pointers	233
Registers	233
Structures, unions, enumerations, and bitfields	233
Qualifiers	234
Declarators	234
Statements	234
Preprocessing directives	234
IAR DLIB Library functions	236
IAR CLIB Library functions	239
Index	243

Tables

1: Typographic conventions used in this guide	xx
2: Command line options for specifying library and dependency files	8
3: XLINK segment memory types	24
4: Memory layout of a target system (example)	25
5: Segment name suffixes	27
6: Library configurations	41
7: Levels of debugging support in runtime libraries	42
8: Prebuilt libraries	43
9: Customizable items	44
10: Formatters for printf	45
11: Formatters for scanf	46
12: Descriptions of printf configuration symbols	55
13: Descriptions of scanf configuration symbols	55
14: Low-level I/O files	56
15: Functions with special meanings when linked with debug info	62
16: Example of runtime model attributes	64
17: Predefined runtime model attributes	65
18: Runtime libraries	70
19: Registers used for passing parameters	85
20: Registers used for returning values	87
21: Call frame information resources defined in a names block	90
22: Compiler optimization levels	100
23: Environment variables	116
24: Error return codes	118
25: Compiler options summary	124
26: Integer types	150
27: Floating-point types	152
28: Extended keywords summary	174
29: Pragma directives summary	181
30: Intrinsic functions summary	197
31: Functions for binary coded decimal operations	199

32: Functions for reading data that has a 20-bit address	201
33: Functions for writing data that has a 20-bit address	201
34: Predefined symbols	208
35: Traditional standard C header files—DLIB	215
36: Embedded C++ header files	216
37: Additional Embedded C++ header files—DLIB	216
38: Standard template library header files	217
39: New standard C header files—DLIB	217
40: IAR CLIB Library header files	220
41: Segment summary	223
42: Message returned by strerror()—IAR DLIB library	238
43: Message returned by strerror()—IAR CLIB library	241

Preface

Welcome to the MSP430 IAR C/C++ Compiler Reference Guide. The purpose of this guide is to provide you with detailed reference information that can help you to use the MSP430 IAR C/C++ Compiler to best suit your application requirements. This guide also gives you suggestions on coding techniques so that you can develop applications with maximum efficiency.

Who should read this guide

You should read this guide if you plan to develop an application using the C or C++ language for the MSP430 microcontroller and need to get detailed reference information on how to use the MSP430 IAR C/C++ Compiler. In addition, you should have a working knowledge of the following:

- The architecture and instruction set of the MSP430 microcontroller. Refer to the documentation from Texas Instruments for information about the MSP430 microcontroller
- The C or C++ programming language
- Application development for embedded systems
- The operating system of your host computer.

How to use this guide

When you start using the MSP430 IAR C/C++ Compiler, you should read *Part 1. Using the compiler* in this guide.

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

If you are new to using the IAR Systems toolkit, we recommend that you first study the *MSP430 IAR Embedded Workbench® IDE User Guide*. This guide contains a product overview, tutorials that can help you get started, conceptual and user information about IAR Embedded Workbench and the IAR C-SPY® Debugger, and corresponding reference information. The *MSP430 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 MSP430 IAR C/C++ Compiler for efficiently developing your application.
- *Data storage* describes how data can be stored in memory.
- *Functions* gives a brief overview of function-related extensions—mechanisms for controlling functions—and describes some of these mechanisms in more detail.
- *Placing code and data* describes the concept of segments, introduces the linker command file, and describes how code and data are placed in memory.
- *The DLIB runtime environment* describes the runtime environment in which an application executes. It covers how you can modify it by setting options, overriding default library modules, or building your own library. The chapter also describes system initialization and introduces the file `cstartup`, as well as how to use modules for locale, and file I/O.
- *The CLIB runtime environment* gives an overview of the runtime libraries and how they can be customized. The chapter also describes system initialization and introduces the file `cstartup`.
- *Assembler language interface* contains information required when parts of an application are written in assembler language. This includes the calling convention.
- *Using C++* gives an overview of the two levels of C++ support: The industry-standard EC++ and IAR Extended EC++.
- *Efficient coding for embedded applications* gives hints about how to write code that compiles to efficient code for an embedded application.

Part 2. Compiler reference

- *Compiler usage* provides reference information about how the compiler interacts with its environment—the invocation syntax, methods for passing options to the compiler, environment variables, the include file search procedure, and the different types of compiler output. The chapter also describes how the compiler’s diagnostic system works.
- *Compiler options* explains how to set the compiler options, gives a summary of the options, and contains detailed reference information for each compiler option.
- *Data representation* describes the available data types, pointers, and structure types. This chapter also gives information about type and object attributes.
- *Compiler extensions* gives a brief overview of the compiler extensions to the ISO/ANSI C standard. More specifically the chapter describes the available C language extensions.
- *Extended keywords* gives reference information about each of the MSP430-specific keywords that are extensions to the standard C/C++ language.
- *Pragma directives* gives reference information about the pragma directives.

- *Intrinsic functions* gives reference information about the functions that can be used for accessing MSP430-specific low-level features.
- *The preprocessor* gives a brief overview of the preprocessor, including reference information about the different preprocessor directives, symbols, and other related information.
- *Library functions* gives an introduction to the C or C++ library functions, and summarizes the header files.
- *Segment reference* gives reference information about the compiler's use of segments.
- *Implementation-defined behavior* describes how the MSP430 IAR C/C++ Compiler handles the implementation-defined areas of the C language standard.

Other documentation

The complete set of IAR Systems development tools for the MSP430 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 *MSP430 IAR Embedded Workbench® IDE User Guide*
- Programming for the MSP430 IAR Assembler, refer to the *MSP430 IAR Assembler Reference Guide*
- Using the IAR XLINK Linker, the IAR XAR Library Builder, and the IAR XLIB Librarian, refer to the *IAR Linker and Library Tools Reference Guide*
- Using the IAR DLIB Library functions, refer to the online help system
- Using the IAR CLIB Library functions, refer to the *IAR C Library Functions Reference Guide*, available from the online help system.
- Porting application code and projects created with a previous MSP430 IAR Embedded Workbench IDE, refer to *MSP430 IAR Embedded Workbench® Migration Guide*.

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

FURTHER READING

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

- *MSP430xxx User's Guide* provided by Texas Instruments
- Barr, Michael, and Andy Oram, ed. *Programming Embedded Systems in C and C++*. O'Reilly & Associates.
- Harbison, Samuel P. and Guy L. Steele (contributor). *C: A Reference Manual*. Prentice Hall.

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

We recommend that you visit the following web sites:

- The Texas Instruments web site, www.ti.com, contains information and news about the MSP430 microcontrollers.
- The IAR Systems web site, www.iar.com, holds application notes and other product information.
- Finally, the Embedded C++ Technical Committee web site, www.caravan.net/ec2plus, contains information about the Embedded C++ standard.

Document conventions

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

TYPOGRAPHIC CONVENTIONS

This guide uses the following typographic conventions:

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

Table 1: Typographic conventions used in this guide

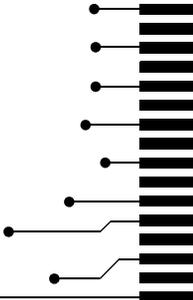
Style	Used for
	Identifies instructions specific to the command line interface.
	Identifies helpful tips and programming hints.

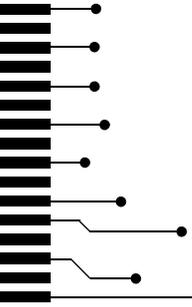
Table 1: Typographic conventions used in this guide (Continued)

Part I. Using the compiler

This part of the MSP430 IAR C/C++ Compiler Reference Guide includes the following chapters:

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





Getting started

This chapter gives the information you need to get started using the MSP430 IAR C/C++ Compiler for efficiently developing your application.

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

Next, the compiler is introduced. You will get an overview of the basic settings needed for a project setup, including an overview of the techniques that enable applications to take full advantage of the MSP430 microcontroller. In the following chapters, these techniques will be studied in more detail.

IAR language overview

There are two high-level programming languages you can use with the MSP430 IAR C/C++ Compiler:

- C, the most widely used high-level programming language used in the embedded systems industry. Using the MSP430 IAR C/C++ Compiler, you can build freestanding applications that follow the standard ISO 9899:1990. This standard is commonly known as ANSI C.
- C++, a modern object-oriented programming language with a full-featured library well suited for modular programming. IAR Systems supports two levels of the C++ language:
 - Embedded C++ (EC++), a subset of the C++ programming standard, which is intended for embedded systems programming. It is defined by an industry consortium, the Embedded C++ Technical committee. See the chapter *Using C++*.
 - IAR Extended EC++, with additional features such as full template support, namespace support, the new cast operators, as well as the Standard Template Library (STL).

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

It is also possible to implement parts of the application, or the whole application, in assembler language. See the *MSP430 IAR Assembler Reference Guide*.

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

Supported MSP430 derivatives

The MSP430 IAR C/C++ Compiler supports all derivatives based on the Texas Instruments MSP430 microcontroller, which includes both the MSP430 architecture and the MSP430X architecture. In addition, the hardware multiplier peripheral unit is also supported.

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 MSP430 IAR C/C++ Compiler or the MSP430 IAR Assembler.

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

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



Below, the process for building an application on the command line is described. For information about how to build an application using the IAR Embedded Workbench IDE, see the *MSP430 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.r43` using the default settings:

```
icc430 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.r43 myfile2.r43 -s __program_start -f lnk430.xcl
c1430f.r43 -o aout.a43 -r
```

In this example, `myfile.r43` and `myfile2.r43` are object files, `lnk430.xcl` is the linker command file, and `c1430f.r43` is the runtime library. The option `-s` specifies the label where the application starts. The option `-o` specifies the name of the output file, and the option `-r` is used for specifying the output format UBROF, which can be used for debugging in C-SPY®.

The IAR XLINK Linker produces output according to your specifications. Choose the output format that suits your purpose. You might want to load the output to a debugger—which means that you need output with debug information. Alternatively, you might want to load the output to a flash loader or a PROM programmer—in which case you need output without debug information, such as Intel-hex or Motorola S-records. The option `-F` can be used for specifying the output format. (The default output format is `mSP430-txt`.)

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 MSP430 device you are using. You can specify the options either from the command line interface or in the IAR Embedded Workbench IDE. For details about how to set options, see *Compiler options syntax*, page 121, and the *MSP430 IAR Embedded Workbench® IDE User Guide*, respectively.

The basic settings available for the MSP430 microcontroller are:

- Core
- Hardware multiplier
- Normal or position-independent code
- 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.

CORE

The MSP430 IAR C/C++ Compiler supports the MSP430 microcontroller, both the MSP430 architecture that has 64 Kbytes of addressable memory and the MSP430X architecture that has the extended instruction set and 1 Mbyte of addressable memory.



In the IAR Embedded Workbench IDE, choose **Project>Options** and select an appropriate device from the **Device** drop-down list on the **Target** page. The core option will then be automatically selected.

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



Use the `--core={430|430X}` option to select the architecture for which the code is to be generated.

HARDWARE MULTIPLIER

Some MSP430 devices contain a hardware multiplier. The MSP430 IAR C/C++ Compiler supports this unit by means of dedicated runtime library modules.



To direct the compiler to take advantage of the unit, choose **Project>Options** and select the **Target** page in the **General Options** category. Select a device, from the **Device** drop-down menu, that contains a hardware multiplier unit.



To use the hardware multiplier, you should, in addition to the runtime library object file, extend the XLINK command line with an additional linker command file:

```
-f multiplier.xcl
```

POSITION-INDEPENDENT CODE

Most applications are designed to be placed at a fixed position in memory. However, by enabling the compiler option `--pic` and choosing a dedicated runtime library object file, you will enable support for a feature known as position-independent code, that allows the application to be placed anywhere in memory. This is useful, for example, when developing modules that should be loaded dynamically at runtime.

The drawback of position-independent code is that the size of the code will be somewhat larger, and that interrupt vectors cannot be specified directly. Also note that global data is not position-independent.

Note: Position-independent code is not supported for the MSP430X architecture.

SIZE OF DOUBLE FLOATING-POINT TYPE

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

OPTIMIZATION FOR SPEED AND SIZE

The MSP430 IAR C/C++ Compiler is a state-of-the-art compiler with an optimizer that performs, among other things, dead-code elimination, constant propagation, inlining, common subexpression 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 100. For more information about efficient coding techniques, see the chapter *Efficient coding for embedded applications*.

RUNTIME ENVIRONMENT

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

There are two different sets of runtime libraries provided:

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

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

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

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



Choosing a runtime library in the IAR Embedded Workbench IDE

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

Note that for the DLIB library there are two different configurations—Normal and Full—which include different levels of support for locale, file descriptors, multibyte characters, et cetera. See *Library configurations*, page 41, for more information.

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



Choosing a runtime library from the command line

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

Command line	Description
<code>-I\msp430\inc</code>	Specifies the include paths
<code>-I\msp430\inc\{clib dlib}</code>	Specifies the library-specific include path. Use <code>clib</code> or <code>dlib</code> depending on which library you are using.
<code>libraryfile.r43</code>	Specifies the library object file
<code>--dlib_config</code> <code>C:\...\configfile.h</code>	Specifies the library configuration file (for the IAR DLIB Library only)

Table 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 8, *Prebuilt libraries*, page 43. The table also shows how the object files correspond to the dependent project options, and the corresponding configuration files. Make sure to use the object file that matches your other project options.

For a list of all prebuilt object files for the IAR CLIB Library, see Table 18, *Runtime libraries*, page 70. The table also shows how the object files correspond to the dependent project options. Make sure to use the object file that matches your other project options.

Setting library and runtime environment options

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

- The formatters used by the functions `printf`, `scanf`, and their variants, see *Choosing formatters for printf and scanf*, page 44 (DLIB) and *Input and output*, page 70 (CLIB).
- The size of the stack and the heap, see *The stack*, page 29, and *The heap*, page 30, respectively.

Special support for embedded systems

This section briefly describes the extensions provided by the MSP430 IAR C/C++ Compiler to support specific features of the MSP430 microcontroller.

EXTENDED KEYWORDS

The MSP430 IAR C/C++ Compiler provides a set of keywords that can be used for configuring how the code is generated. For example, there are keywords for declaring special function types.



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

The command line option `-e` makes the extended keywords available, and reserves them so that they cannot be used as variable names. See, *-e*, page 132 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 MSP430 IAR C/C++ Compiler. They are consistent with ISO/ANSI C, and are very useful when you want to make sure that the source code is portable.

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

PREDEFINED SYMBOLS

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

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

SPECIAL FUNCTION TYPES

The special hardware features of the MSP430 microcontroller are supported by the compiler's special function types: interrupt, monitor, task, and raw. You can write a complete application without having to write any of these functions in assembler language.

For detailed information, see *Primitives for interrupts, concurrency, and OS-related programming*, page 15.

HEADER FILES FOR I/O

Standard peripheral units are defined in device-specific I/O header files called `iodevice.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 `430\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 109.

ACCESSING LOW-LEVEL FEATURES

For hardware-related parts of your application, accessing low-level features is essential. The MSP430 IAR C/C++ Compiler supports several ways of doing this: intrinsic functions, mixing C and assembler modules, and inline assembler. For information about the different methods, see *Mixing C and assembler*, page 77.

Data storage

This chapter gives a brief introduction to the memory layout of the MSP430 microcontroller and the fundamental ways data can be stored in memory: on the stack, in static (global) memory, or in heap memory. Finally, detailed information about data storage on the stack and the heap is provided.

Introduction

The MSP430 IAR C/C++ Compiler supports MSP430 devices with both the MSP430 instruction set and the MSP430X extended instruction set, which means that 64 Kbytes and 1 Mbyte of continuous memory can be used. However, the extended instruction set requires that data—including constant data—interrupt functions, and interrupt vectors must be located in the lower 64 Kbytes of memory.

Different types of physical memory can be placed in the memory range. A typical application will have both read-only memory (ROM or flash) and read/write memory (RAM). In addition, some parts of the memory range contain processor control registers and memory-mapped registers for peripheral units.

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.

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 *recursive function*—and each invocation can store its own data on the stack.

Potential problems

The way the stack works makes it impossible to store data that is supposed to live after the function has returned.

The following function demonstrates a common programming mistake. It returns a pointer to the variable `x`, a variable that ceases to exist when the function returns.

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

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

Dynamic memory on the heap

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

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

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

Potential problems

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

The heap can become exhausted 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. It gives a brief overview of function-related extensions—mechanisms for controlling functions—and describes some of these mechanisms in more detail.

Function-related extensions

In addition to the ISO/ANSI C standard, the MSP430 IAR C/C++ Compiler provides several extensions for writing functions in C. Using these, you can:

- Use primitives for interrupts, concurrency, and OS-related programming
- Facilitate function optimization
- Access hardware features.

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

For more information about optimizations, see *Writing efficient code*, page 106. For information about the available intrinsic functions for accessing hardware operations, see the chapter *Intrinsic functions*, page 197.

Primitives for interrupts, concurrency, and OS-related programming

The MSP430 IAR C/C++ Compiler provides the following primitives related to writing interrupt functions, concurrent functions, and OS-related functions:

- The extended keywords `__interrupt`, `__task`, `__monitor`, `__raw`, and `__save_reg20`
- The pragma directives `#pragma vector` and `#pragma no_epilogue`
- The intrinsic functions `__enable_interrupt`, `__disable_interrupt`, as well as many others
- The compiler option `--save_reg20`.

INTERRUPT FUNCTIONS

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

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

The MSP430 microcontroller supports many interrupt sources. For each interrupt source, an interrupt routine can be written. Each interrupt routine is associated with a vector number, which is specified in the MSP430 microcontroller documentation from the chip manufacturer. The interrupt vector is the offset into the interrupt vector table. If you want to handle several different interrupts using the same interrupt function, you can specify several interrupt vectors. For the MSP430 microcontroller, the interrupt vector table contains 16 or 32 vectors and the table starts at either the address `0xFFE0` or `0xFFC0`, respectively. The last entry in the table contains the reset vector, which is placed in a separate linker segment—`RESET`.

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

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

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

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

If a vector is specified in the definition of an interrupt function, the processor interrupt vector table is populated. It is also possible to define an interrupt function without a vector. This is useful if an application is capable of populating or changing the interrupt vector table at runtime. See the chip manufacturer's MSP430 microcontroller documentation for more information about the interrupt vector table.

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

Preventing registers from being saved at function entrance

As noted, the interrupt function preserves the content of all used processor register at the entrance and restores them at exit. However, for some very special applications, it can be desirable to prevent the registers from being saved at function entrance.

This can be accomplished by the use of the extended keyword `__raw`, for example:

```
__raw __interrupt void my_interrupt_function()
```

This creates an interrupt service routine where you must make sure that the code within the routine does not affect any of the registers used by the interrupted environment. Typically, this is useful for applications that have an empty foreground loop and use interrupt routines to perform all work.

Note: The same effect can be achieved for normal functions by using the `__task` keyword.

Interrupt Vector Generator interrupt functions

The MSP430 IAR C/C++ Compiler provides a way to write very efficient interrupt service routines for the Interrupt Vector Generators for Timer A (TAIV), Timer B (TBIV), the I²C module (I2CIV), and the ADC12 module.

The interrupt vector register contains information about the interrupt source, and the interrupt service routine normally uses a switch statement to find out which interrupt source issued the interrupt. To help the compiler generate optimal code for the switch statement, the intrinsic function `__even_in_range` can be used. The following example defines a Timer A interrupt routine:

```
#pragma vector=TIMER_A1_VECTOR
__interrupt void Timer_A1_ISR(void)
{
    switch (__even_in_range(TAIV, 10))
    {
        case 2: P1POUT ^= 0x04;
                break;
        case 4: P1POUT ^= 0x02;
                break;
        case 10: P1POUT ^= 0x01;
                break;
    }
}
```

The intrinsic function `__even_in_range` requires two parameters, the interrupt vector register and the last value in the allowed range, which in this example is 10. The effect of the intrinsic function is that the generated code can only handle even values within the given range, which is exactly what is required in this case as the interrupt vector register for Timer A can only be 0, 2, 4, 6, 8, or 10. If the `__even_in_range` intrinsic function is used in a case where an odd value or a value outside the given range could occur, the program will fail.

For more information about the intrinsic keyword, see `__even_in_range`, page 202.

Interrupt functions for the MSP430X architecture

When compiling for the MSP430X architecture, all interrupt functions are automatically placed in the segment `ISR_CODE`, which must be located in the lower 64 Kbytes of memory. If you are using a ready-made linker command file for an MSP430X device, the segment will be correctly located.

By default, all functions save only 16 bits of the 20-bit registers on entry and exit. If you have assembler routines that use the upper 4 bits of the registers, you must use either the `__save_reg20` keyword on all your interrupt functions, alternatively the `--save_reg20` compiler option.

This will ensure that the interrupt functions will save and restore all 20 bits of any 20-bit registers that are used. The drawback is that the entry and leave sequences will become slower and consume more stack space.

Note: If a `__save_reg20` function, compiled using either the `--lock_R4` or the `--lock_R5` option, calls another function that is not `__save_reg20` declared and does not lock R4/R5, the upper four bits of R4/R5 might be destroyed. For this reason, it is not recommended to use different settings of the `--lock_R4/R5` option for different modules.

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.

Avoid using the `__monitor` keyword on large functions, since the interrupt will otherwise be turned off for a significant period of time.

Example of implementing a semaphore in C

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

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

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

```

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

/* release_lock -- Unlock the lock. */
__monitor void release_lock(void)
{
    the_lock = 0;
}

```

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

```

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

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

```

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 MSP430 IAR C/C++ Compiler does not support inlining of functions and methods that are declared using the `__monitor` keyword.

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

```

#include <intrinsics.h>

volatile long tick_count = 0;

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

    ~Mutex ()
    {
        __set_interrupt_state(_state);
    }

private:
    __istate_t _state;
};

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

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

        tick = tick_count;
    }

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

```

C++ AND SPECIAL FUNCTION TYPES

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

Placing code and data

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

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

Segments and memory

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

WHAT IS A SEGMENT?

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

The MSP430 IAR C/C++ Compiler has a number of predefined segments for different purposes. Each segment has a name that describes the contents of the segment, and a *segment memory type* that denotes the type of content. In addition to the predefined segments, you can define your own segments.

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

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

Segment memory type

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

By default, the MSP430 IAR C/C++ Compiler uses only the following XLINK segment memory types:

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

Table 3: XLINK segment memory types

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

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

Placing segments in memory

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

In particular, the linker command file specifies:

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

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

CUSTOMIZING THE LINKER COMMAND FILE

The `config` directory contains one ready-made linker command file for each MSP430 device. The files contain the information required by the linker, and is ready to be used. The only change you will normally have to make to the supplied linker command file is to customize it so it fits the target system memory map. If, for example, your application uses additional external RAM, you need to add details about the external RAM memory area.

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

Range	Type
0x0200–0x09FF	RAM
0x1000–0x10FF	ROM
0x1100–0xFFFF	RAM

Table 4: Memory layout of a target system (example)

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

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

The contents of the linker command file

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

- The CPU used:
`-cmsp430`
This specifies your target microcontroller.
- Definitions of constants used later in the file. These are defined using the XLINK option `-D`.
- The placement directives (the largest part of the linker command file). Segments can be placed using the `-Z` and `-P` options. The former will place the segment parts in the order they are found, while the latter will try to rearrange them to make better use of the memory. The `-P` option is useful when the memory where the segment should be placed is not continuous.

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

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

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

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

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

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

```
-Z (CONST) MYSEGMENTA, MYSEGMENTB=2000-CFFF
```

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

```
-Z (CONST) MYSEGMENTA=2000-CFFF
-Z (CODE) MYCODE
```

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

```
-Z (CONST) MYSMALLSEGMENT=2000-20FF
-Z (CONST) MYLARGESEGMENT=2000-CFFF
```



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

Using the **-P** command for packed placement

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

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

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

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

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

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

Data segments

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

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
- Restrictions for segments holding initialized data
- The placement and size limitation of the static memory segments.

Segment naming

The actual segment names consist of two parts—a *segment base name* and a *suffix* that specifies what the segment is used for. In the MSP430 IAR C/C++ Compiler, the segment base name is `DATA16`.

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

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

Table 5: Segment name suffixes

Categories of declared data	Segment memory type	Suffix
Constants	CONST	C
Non-initialized absolute addressed data		AN
Constant absolute addressed data		AC

Table 5: Segment name suffixes (Continued)

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 24.

For a summary of all supported segments, see *Summary of segments*, page 223.

Examples

Assume the following examples:

<code>int j;</code>	The variables that are to be initialized to zero when the system starts will be placed in the segment <code>DATA16_Z</code> .
<code>int i = 0;</code>	
<code>__no_init int j;</code>	The non-initialized variables will be placed in the segment <code>DATA16_N</code> .
<code>int j = 4;</code>	The non-zero initialized variables will be placed in the segment <code>DATA16_I</code> , and initializer data in segment <code>DATA16_ID</code> .

Initialized data

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

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

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

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

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

```
DATA16_I           0x200-0x2FF and 0x400-0x4FF
DATA16_ID          0x600-0x7FF
```

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           0x200-0x2FF and 0x400-0x4FF
DATA16_ID          0x600-0x6FF and 0x800-0x8FF
```

Note that the gap between the ranges will also be copied.

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

Data segments for static memory in the default linker command file

In this example, the directives for placing the segments in the linker command file would be:

```
// The RAM segments
-Z (DATA) DATA16_I, DATA16_Z, DATA16_N=200-9FF

// The ROM segments
-Z (CONST) DATA16_C=1100-FFDF, DATA16_ID
```

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 IDE

Select **Project>Options**. In the **General Options** category, click the **Stack/Heap** tab. Add the required stack size in the **Stack size** text box.



Stack size allocation from the command line

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

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

```
-D_CSTACK_SIZE=size
```

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#0200-09FF
```

Note:

- This range does not specify the size of the stack; it specifies the range of the available memory
- The # allocates the `CSTACK` segment at the end of the memory area. In practice, this means that the stack will get all remaining memory at the same time as it is guaranteed that it will be at least `_CSTACK_SIZE` bytes. See the *IAR Linker and Library Tools Reference Guide* for more information.



Stack size considerations

The compiler uses the internal data stack, `CSTACK`, for a variety of user program operations, and the required stack size depends heavily on the details of these operations. If the given stack size is too large, RAM will be wasted. If the given stack size is too small, memory might be overwritten leading to undefined behavior. Because of this, you should consider placing the stack at the end of the RAM memory.

THE HEAP

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

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

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

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



Heap size allocation in the IAR Embedded Workbench IDE

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

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



Heap size allocation from the command line

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

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

```
-D_HEAP_SIZE=50
```

Specify the appropriate size for your application.



Placement of heap segment

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

```
-Z (DATA) HEAP+_HEAP_SIZE=08000-08FFF
```

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



Heap size and standard I/O

When you are using the full DLIB 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 buffered I/O. 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 MSP430 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, for example 1 Kbyte.

If you have excluded `FILE` descriptors from the DLIB runtime environment, as in the normal DLIB configuration, there are no input and output buffers at all.

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 constants, 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 223.

STARTUP CODE

The segment `CSTART` contains code used during system setup (`cstartup`) and system termination (`cexit`). The system setup code should be placed at the location where the chip starts executing code after a reset. For the MSP430 microcontroller, this is at the reset vector address. In addition, the segments must be placed into one continuous memory space, which means the `-P` segment directive cannot be used.

In the default linker command file, the following line will place the `CSTART` segment at the address `0x1100`:

```
-Z (CODE) CSTART=1100-FFBF
```

NORMAL CODE

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

```
/* For MSP430 devices */  
-Z (CODE) CODE=1100-FFDF  
  
/* For MSP430X devices */  
-Z (CODE) CODE=1100-FFBF,10000-FFFFF
```

INTERRUPT FUNCTIONS FOR MSP430X

When you compile for the MSP430X architecture, the interrupt functions are placed in the `ISR_CODE` segment. Again, this is a simple operation in the linker command file:

```
-Z (CODE) ISR_CODE=1100-FFDF
```

INTERRUPT VECTORS

The interrupt vector table contains pointers to interrupt routines, including the reset routine. The table is placed in the segment `INTVEC`. For the MSP430 microcontroller, you must place this segment on the address `0xFFE0`. The linker directive would then look like this for a device with 16 interrupt vectors:

```
-Z (CONST) INTVEC=FFE0-FFFE
```

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=1100-FFFE
```

For additional information, see *DIFUNCT*, page 227.

Controlling data and function placement in memory

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

- For absolute placement, use the `@` operator and the `#pragma location` directive

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

Use the `@` operator or the `#pragma location` directive to place groups of functions or global and static variables in named segments, without having explicit control of each object. The variables must be declared either `__no_init` or `const`. The segments can, for example, be placed in specific areas of memory, or initialized or

copied in controlled ways using the segment begin and end operators. This is also useful if you want an interface between separately linked units, for example an application project and a boot loader project. Use named segments when absolute control over the placement of individual variables is not needed, or not useful.

At compile time, data and functions are placed in different segments as described in *Data segments*, page 27, and *Code segments*, page 32, respectively. At link time, one of the most important functions of the linker is to assign load addresses to the various segments used by the application. All segments, except for the segments holding absolute located data, are automatically allocated to memory according to the specifications of memory ranges in the linker command file, as described in *Placing segments in memory*, page 24.

DATA PLACEMENT AT AN ABSOLUTE LOCATION

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

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

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

Declaring located variables extern and volatile

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

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

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

For information about `volatile` declared objects, see *Protecting simultaneously accessed variables*, page 109.

Examples

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

```
__no_init char alpha @ 0x0200;    /* OK */
```

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

```
#pragma location=0x0202
volatile const int beta;          /* OK */

volatile const int gamma @ 0x0204 = 3; /* OK */
```

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

```
volatile __no_init const char c @ 0x0204;
void foo(void)
{
    ...
    a = b + c + d;
    ...
}
```

The following examples show incorrect usage:

```
int delta @ 0x0206;                /* Error, neither */
                                   /* "__no_init" nor "const".*/

const int epsilon @ 0x0207;        /* Error, misaligned. */
```

DATA AND FUNCTION PLACEMENT IN SEGMENTS

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

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

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

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

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

Examples of placing variables in named segments

In the following three examples, a data object is placed in a user-defined segment.

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

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

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

The following example shows incorrect usage:

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

Examples of placing functions in named segments

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

void g(void) @ "MYSEGMENT"
{
}

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

Verifying the linked result of code and data placement

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

SEGMENT TOO LONG ERRORS AND RANGE ERRORS

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

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

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

LINKER MAP FILE

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

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



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

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

For further information about the listing options and the linker listing, see the *IAR Linker and Library Tools Reference Guide*, and the *MSP430 IAR Embedded Workbench® IDE User Guide*.

Verifying the linked result of code and data placement

The DLIB runtime environment

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

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

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

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

Introduction to the runtime environment

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

This section gives an overview of:

- The runtime environment and its components
- Library selection.

RUNTIME ENVIRONMENT FUNCTIONALITY

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

The runtime library is delivered both as prebuilt libraries and as source files, and you can find them in the product subdirectories `430\lib` and `430\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
 - The MSP430 hardware multiplier peripheral unit.
- 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 heap must be tailored for the specific hardware and application requirements.

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

LIBRARY SELECTION

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

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

- Setting library options, for example, for choosing `scanf` input and `printf` output formatters, and for specifying the size of the stack and the heap
- Overriding certain library functions, for example `cstartup.s43`, with your own customized versions
- Choosing the level of support for certain standard library functionality, for example, locale, file descriptors, and multibyte characters, by choosing a *library configuration*: normal or full.

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

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

SITUATIONS THAT REQUIRE LIBRARY BUILDING

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

You must build your own library when:

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

For information about how to build a customized library, see *Building and using a customized library*, page 48.

LIBRARY CONFIGURATIONS

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

The following DLIB library configurations are available:

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

Table 6: 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 48.

The prebuilt libraries are based on the default configurations, see Table 8, *Prebuilt libraries*, page 43. There is also a ready-made library project template that you can use if you want to rebuild the runtime library.

DEBUG SUPPORT IN THE RUNTIME LIBRARY

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

The following table describes the different levels of debugging support:

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

Table 7: 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 62.



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

Using a prebuilt library

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

- Core
- Size of the double floating-point type
- Library configuration—Normal or Full
- Position-independent code.

For the MSP430 IAR C/C++ Compiler, the following prebuilt runtime libraries are available:

Library	Core	Size of double	Library configuration	PIC
d1430fn.r43	430	32	Normal	No
d1430fnp.r43	430	32	Normal	Yes
d1430ff.r43	430	32	Full	No
d1430ffp.r43	430	32	Full	Yes
d1430dn.r43	430	64	Normal	No
d1430dnp.r43	430	64	Normal	Yes
d1430df.r43	430	64	Full	No
d1430dfp.r43	430	64	Full	Yes
d1430xfn.r43	430X	32	Normal	No
d1430xff.r43	430X	32	Full	No
d1430xdn.r43	430X	64	Normal	No
d1430xdf.r43	430X	64	Full	No

Table 8: Prebuilt libraries

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

```
<type><core><size_of_double><library_config><PIC>.r43
```

where

- `<type>` is `d1` for the IAR DLIB runtime environment
- `<core>` is either `430` or `430x`
- `<size_of_double>` is either `f` for 32 bits or `d` for 64 bits
- `<library_config>` is one of `n` or `f` for normal and full, respectively
- `<PIC>` is either empty for no support for position-independent code or `p` for position-independent code.

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



The IAR Embedded Workbench IDE will include the correct library object file and library configuration file based on the options you select. See the *MSP430 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:


```
d1430fn.r43
```
- Specify the include paths for the compiler and assembler:


```
-I msp430\inc
```

- Specify the library configuration file for the compiler:
`--dlib_config C:\...\dl430fn.h`

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

You can find the library object files and the library configuration files in the subdirectory `430\lib`.

CUSTOMIZING A PREBUILT LIBRARY WITHOUT REBUILDING

The prebuilt libraries delivered with the MSP430 IAR C/C++ Compiler can be used as is. However, it is possible to customize parts of a library without rebuilding it. There are two different methods:

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

The following items can be customized:

Items that can be customized	Described on page
Formatters for <code>printf</code> and <code>scanf</code>	<i>Choosing formatters for printf and scanf</i> , page 44
Startup and termination code	<i>System startup and termination</i> , page 50
Low-level input and output	<i>Standard streams for input and output</i> , page 53
File input and output	<i>File input and output</i> , page 56
Low-level environment functions	<i>Environment interaction</i> , page 59
Low-level signal functions	<i>Signal and raise</i> , page 60
Low-level time functions	<i>Time</i> , page 60
Size of heaps, stacks, and segments	<i>Placing code and data</i> , page 23

Table 9: Customizable items

For a description about how to override library modules, see *Overriding library modules*, page 47.

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 55.

CHOOSING PRINTF FORMATTER

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

The following table summarizes the capabilities of the different formatters:

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

Table 10: 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 55.



Specifying the print formatter in the IAR Embedded Workbench IDE

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



Specifying printf formatter from the command line

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

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

```
-e_PrintfTiny=_Printf
```

CHOOSING SCANF FORMATTER

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

The following table summarizes the capabilities of the different formatters:

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

Table 11: 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 55.



Specifying `scanf` formatter in the IAR Embedded Workbench IDE

To use any other formatter than the default (Large), 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 formatter 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 `430\src\lib` directory.

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



Overriding library modules using the IAR Embedded Workbench IDE

This procedure is applicable to any source file in the library, which means `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:

```
icc430 library_module
```

This creates a replacement object module file named `library_module.r43`.

Note: The 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.r43* to the XLINK command line, either directly or by using an extended linker command file, for example:

```
xlink library_module dl430fn.r43
```

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.r43*, 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 40, 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 *MSP430 IAR Embedded Workbench® IDE User Guide*.

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

SETTING UP A LIBRARY PROJECT

The IAR Embedded Workbench IDE provides a library project template which can be used for customizing the runtime environment configuration. This library template has full library configuration, see Table 6, *Library configurations*, page 41.



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

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

MODIFYING THE LIBRARY FUNCTIONALITY

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

The library functionality is determined by a set of *configuration symbols*. The default values of these symbols are defined in the file `Dlib_defaults.h`. This read-only file describes the configuration possibilities. In addition, your library has its own library configuration file `dl430Custom.h`, which sets up that specific library with full library configuration. For more information, see Table 9, *Customizable items*, page 44.

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

Modifying the library configuration file

In your library project, open the file `dl430Custom.h` and customize it by setting the values of the configuration symbols according to the application requirements.

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

USING A CUSTOMIZED LIBRARY

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



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

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

System startup and termination

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

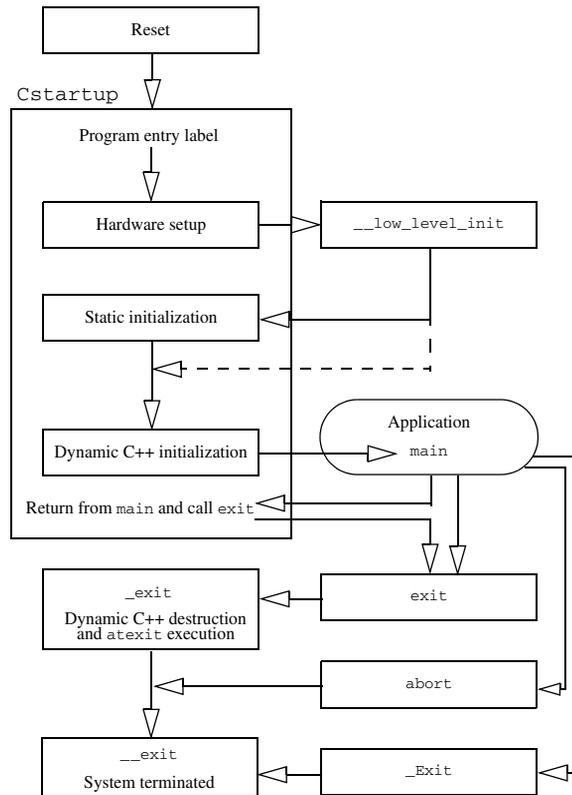


Figure 1: Startup and exit sequences

The code for handling startup and termination is located in the source files `cstartup.s43`, `cexit.s43`, and `low_level_init.c` located in the `430\src\lib` directory.

SYSTEM STARTUP

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

- When the CPU is reset it will jump to the program entry label `__program_start` in the system startup code
- The stack pointer (`SP`) is initialized
- 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.

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

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

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

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

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

C-SPY interface to system termination

If your project is linked with the XLINK options **With runtime control modules** or **With I/O emulation modules**, the normal `__exit` and `abort` functions are replaced with special ones. C-SPY will then recognize when those functions are called and can take appropriate actions to simulate program termination. For more information, see *C-SPY Debugger runtime interface*, page 62.

Customizing system initialization

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

You can do this by providing a customized version of the routine `__low_level_init`, which is called from `cstartup.s43` before the data segments are initialized. Modifying the file `cstartup` directly should be avoided.

The code for handling system startup is located in the source files `cstartup.s43` and `low_level_init.c`, located in the `430\src\lib` directory.

Note: Normally, there is no need for customizing either of the files `cstartup.s43` or `cexit.s43`.



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 48.

Note: Regardless of whether you modify the routine `__low_level_init` or the file `cstartup.s43`, 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.s43`. The latter is part of the prebuilt runtime environment. The only limitation using the C source version is that static initialized variables cannot be used within the file.

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

Note: The file `intrinsics.h` must be included by `low_level_init.c` to assure correct behavior of the `__low_level_init` routine.

MODIFYING THE FILE CSTARTUP.S43

As noted earlier, you should not modify the file `cstartup.s43` if a customized version of `__low_level_init` is enough for your needs. However, if you do need to modify the file `cstartup.s43`, 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 47.

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 `430\src\lib` directory.

If you intend to rebuild the library, the source files are available in the template library project, see *Building and using a customized library*, page 48. 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 62.

Example of using `__write` and `__read`

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

```
__no_init volatile unsigned char LCD_IO @ address;

size_t __write(int Handle, const unsigned char * Buf,
              size_t Bufsize)
{
    int nChars = 0;
```

```

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

```

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

```

__no_init volatile unsigned char KB_IO @ 0xD2;

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

```

For information about the @ operator, see *Data and function placement in segments*, page 35.

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 44.

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

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

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

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

Table 12: Descriptions of printf configuration symbols

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

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

Table 13: 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 48. 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 41. In other words, file I/O is supported when the configuration symbol `__DLIB_FILE_DESCRIPTOR` is enabled. If not enabled, functions taking a `FILE *` argument cannot be used.

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

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

Table 14: 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 41.



Heap size and standard I/O

When you are using the full DLIB 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 buffered I/O. 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 MSP430 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, for example 1 Kbyte.

If you have excluded `FILE` descriptors from the DLIB runtime environment, as in the normal DLIB configuration, there are no input and output buffers at all.

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 main modes:

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

LOCALE SUPPORT IN PREBUILT LIBRARIES

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

- All prebuilt libraries 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 48.

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

Building a library without support for locale interface

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

Building a library with support for locale interface

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

CHANGING LOCALES AT RUNTIME

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

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

```
lang_REGION
```

or

```
lang_REGION.encoding
```

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

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

Example

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

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

Environment interaction

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

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

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

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

```
key=value\0
```

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

For example:

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

If you need a more sophisticated environment variable handling, you should implement your own `getenv`, and possibly `putenv` function. This does not require that you rebuild the library. You can find source templates in the files `getenv.c` and `environ.c` in the `430\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 47.

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 48.

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 41.

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 `430\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 47.

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 48.

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 `430\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 47.

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 48.

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 62.

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 48. 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 `430\src\lib` directory. For further information, see *Building and using a customized library*, page 48. To turn off assertions, you must define the symbol `NDEBUG`.



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

Hardware multiplier support

Some MSP430 devices contain a hardware multiplier. The MSP430 IAR C/C++ Compiler supports this unit by means of dedicated runtime library modules.



To make the compiler take advantage of the hardware multiplier unit, choose **Project>Options>General Options>Target** and select a device that contains a hardware multiplier unit from the **Device** drop-down menu. Make sure that the option **Hardware multiplier** is selected.



Specify which runtime library object file to use on the XLINK command line.

In addition to the runtime library object file, you must extend the XLINK command line with an additional linker command file if you want support for the hardware multiplier.

To use the hardware multiplier, extend the XLINK command line with the directive:

```
-f multiplier.xcl
```

Note: Interrupts are disabled during a hardware-multiply operation.

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

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

Table 15: 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 41. 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 *MSP430 IAR Embedded Workbench® IDE User Guide* for more information about the Terminal I/O window.

Speeding up terminal output

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

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

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

```
-e__write_buffered=__write
```

Checking module consistency

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

When developing an application, it is important to ensure that incompatible modules are not used together. For example, in the MSP430 IAR C/C++ Compiler, it is possible to specify the size of the `double` floating-point type. If you write a routine that only works for 64-bit doubles, it is possible to check that the routine is not used in an application built using 32-bit doubles.

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

RUNTIME MODEL ATTRIBUTES

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

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

Example

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

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

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

Table 16: Example of runtime model attributes

USING RUNTIME MODEL ATTRIBUTES

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

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

For detailed syntax information, see *rtmodel*, page 193.

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

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

For detailed syntax information, see the *MSP430 IAR Assembler Reference Guide*.

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

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

PREDEFINED RUNTIME ATTRIBUTES

The table below shows the predefined runtime model attributes that are available for the MSP430 IAR C/C++ Compiler. These can be included in assembler code or in mixed C or C++ and assembler code.

Runtime model attribute	Value	Description
<code>__core</code>	64kb or 1MB	This runtime key is set either to 64kb or 1MB depending on the <code>--core</code> option.
<code>__double_size</code>	32 or 64	The size, in bits, of the double floating-point type.
<code>__reg_r4</code> <code>__reg_r5</code>	free or undefined	Corresponds to the use of the register, or undefined when the register is not used. A routine that assumes that the register is locked should set the attribute to a value other than <code>free</code> .

Table 17: Predefined runtime model attributes

Runtime model attribute	Value	Description
<code>__rt_version</code>	<code>n</code>	This runtime key is always present in all modules generated by the MSP430 IAR C/C++ Compiler. If a major change in the runtime characteristics occurs, the value of this key changes.

Table 17: Predefined runtime model attributes (Continued)

Note: The value `free` should be seen as the opposite of `locked`, that is, the register is free to be used by the compiler.

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 *MSP430 IAR Assembler Reference Guide*.

Example

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

```

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

```

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

```

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

```

USER-DEFINED RUNTIME MODEL ATTRIBUTES

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

For example, if you have a UART that can run in two modes, you can specify a runtime model attribute, for example `uart`. For each mode, specify a value, for example `mode1` and `mode2`. 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"
```


The CLIB runtime environment

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

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

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

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

For information about migrating from CLIB to DLIB, see the *MSP430 IAR Embedded Workbench® Migration Guide*.

Runtime environment

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

The MSP430 IAR Embedded Workbench comes with a set of prebuilt runtime libraries, which are configured for different combinations of the following features:

- Core
- Size of the double floating-point type
- Position-independent code.

The following prebuilt runtime libraries are provided:

Library object file	Processor core	Size of double	PIC
c1430f.r43	430	32	No
c1430fp.r43	430	32	Yes
c1430d.r43	430	64	No
c1430dp.r43	430	64	Yes
c1430xf.r43	430X	32	No
c1430xd.r43	430X	64	No

Table 18: Runtime libraries

The runtime library names are constructed in the following way:

```
<type><core><size_of_double><PIC>.r43
```

where

- `<type>` c1 for the IAR CLIB Library
- `<core>` is 430 or 430x
- `<size_of_double>` is either `f` for 32 bits or `d` for 64 bits
- `<PIC>` is either empty for no support for position-independent code or `p` for position-independent code.



The IAR Embedded Workbench IDE includes the correct runtime library based on the options you select. See the *MSP430 IAR Embedded Workbench® IDE User Guide* for additional information.



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

```
c1430d.r43
```

Input and output

You can customize:

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

CHARACTER-BASED I/O

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

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

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

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

```
__no_init volatile unsigned char DEV_IO @ address;

int putchar(int outchar)
{
    DEV_IO = outchar;
    return outchar;
}
```

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

For information about how to include your own modified version of `putchar` and `getchar` in your project build process, see *Overriding library modules*, page 47.

FORMATTERS USED BY PRINTF AND SPRINTF

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

`_medium_write`

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

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

`_medium_write` is considerably smaller than `_formatted_write`.

`_small_write`

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



Specifying the printf formatter in the IAR Embedded Workbench IDE

- 1 Choose **Project>Options** and select the **General Options** category. Click the **Library options** tab.
- 2 Select the appropriate **Printf formatter** option, which can be either **Small**, **Medium**, or **Large**.



Specifying the printf formatter from the command line

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

```
-e_small_write=_formatted_write
```

or

```
-e_medium_write=_formatted_write
```

To use the full version, remove the line.

Customizing printf

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

For such applications, a much reduced version of the `printf` function (without `sprintf`) is supplied in source form in the file `intwri.c`. This file can be modified to meet your requirements, and the compiled module inserted into the library in place of the original file; see *Overriding library modules*, page 47.

FORMATTERS USED BY SCANF AND SSCANF

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

`_medium_read`

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



Specifying the scanf formatter in the IAR Embedded Workbench IDE

- 1 Choose **Project>Options** and select the **General Options** category. Click the **Library options** tab.
- 2 Select the appropriate **Scanf formatter** option, which can be either **Medium** or **Large**.



Specifying the read formatter from the command line

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

```
-e_medium_read=_formatted_read
```

To use the full version, remove the line.

System startup and termination

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

The code for handling startup and termination is located in the source files `cstartup.s43`, `cexit.s43`, and `low_level_init.c` located in the `430\src\lib` directory.

Note: Normally, there is no need for customizing either of the files `cstartup.s43` or `cexit.s43`.

SYSTEM STARTUP

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

- The stack pointer (`SP`) is initialized
- The custom 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 remaining initialized variables
- The `main` function is called, which starts the application.

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

SYSTEM TERMINATION

An application can terminate normally in two different ways:

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

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

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

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

Overriding default library modules

The IAR CLIB Library contains modules which you probably need to override with your own customized modules, for example for character-based I/O, without rebuilding the entire library. For information about how to override default library modules, see *Overriding library modules*, page 47 in the chapter *The DLIB runtime environment*.

Customizing system initialization

For information about how to customize system initialization, see *Customizing system initialization*, page 52.

C-SPY runtime interface

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

THE DEBUGGER TERMINAL I/O WINDOW

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

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

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

To make the Terminal I/O window available, the application must be linked with the XLINK option **With I/O emulation modules** selected. See the *MSP430 IAR Embedded Workbench® IDE User Guide*.

TERMINATION

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

Checking module consistency

For information about how to check module consistency, see *Checking module consistency*, page 64 in the chapter *The DLIB runtime environment*.

Assembler language interface

When you develop an application for an embedded system, there may be situations where you will find it necessary to write parts of the code in assembler, for example, when using mechanisms in the MSP430 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 functions are called in the different code models and how you can implement support for call frame information in your assembler routines for use in the C-SPY® Call Stack window.

Mixing C and assembler

The MSP430 IAR C/C++ Compiler provides several ways to mix C or C++ and assembler:

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

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

INTRINSIC FUNCTIONS

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

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

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

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

MIXING C AND ASSEMBLER MODULES

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

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

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?
- 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 83.

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 90.

There will be some overhead in the form of a function call and return instruction sequences, and the compiler will regard some registers as scratch registers.

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

The recommended method for mixing C or C++ and assembler modules is described in *Calling assembler routines from C*, page 80, and *Calling assembler routines from C++*, page 82, 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.B &PIN,&flag");
    }
}
```

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

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

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

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

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

Calling assembler routines from C

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

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

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

or

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

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

CREATING SKELETON CODE

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

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

```
extern int gInt;
extern double gDouble;

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

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

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

COMPILING THE CODE



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



Use the following options to compile the skeleton code:

```
icc430 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 `s43`. Also remember to specify a low level of optimization and `-e` for enabling language extensions.

The result is the assembler source output file `skeleton.s43`.

Note: The `-lA` option creates a list file containing call frame information (CFI) directives, which can be useful if you intend to study these directives and how they are used. If you only want to study the calling convention, you can exclude the CFI directives from the list file. In the IAR Embedded Workbench IDE, select



Project>Options>C/C++ Compiler>List and deselect the suboption **Include compiler runtime information**. On the command line, use the option `-lB` instead of `-lA`. Note that CFI information must be included in the source code to make the C-SPY Call Stack window work.



The output file

The output file contains the following important information:

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

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

Calling assembler routines from C++

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

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

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

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

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

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

```
class X;

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

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

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

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

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

Calling convention

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

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

This section describes the calling conventions used by the MSP430 IAR C/C++ Compiler. The following items are examined:

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

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

FUNCTION DECLARATIONS

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

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

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

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

C AND C++ LINKAGE

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

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

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

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

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

    int f(int);

#ifdef __cplusplus
}
#endif
```

PRESERVED VERSUS SCRATCH REGISTERS

The general MSP430 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.

Any of the registers R12 to R15, as well as the return address registers, are considered scratch registers and can be used by the function.

When the registers R11 : R10 : R9 : R8 are used for passing a 64-bit scalar parameter, they are also considered to be scratch registers.

Preserved registers

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

The registers R4 to R11, but not including the return address registers, are preserved registers.

If the registers R11 : R10 : R9 : R8 are used for passing a 64-bit scalar parameter, they do not have to be preserved.

Note:

- When compiling for the MSP430X architecture, only the lower 16 bits of the registers are preserved, unless the `__save_reg20` attribute is specified. It is only necessary to save and restore the upper 4 bits if you have an assembler routine that uses these bits.

- When compiling using the options `--lock_r4` or `--lock_r5`, the R4 and R5 registers are not preserved.

Special registers

You must consider that the stack pointer register must at all times point to the last element on the stack or below. In the eventuality of an interrupt, which can occur at any time, everything below the point the stack pointer points to, will be destroyed.

FUNCTION ENTRANCE

Parameters can be passed to a function using one of two basic methods: in registers or on the stack. It is much more efficient to use registers than to take a detour via memory, so the calling convention is designed to utilize registers as much as possible. There is only a limited number of registers that can be used for passing parameters; when no more registers are available, the remaining parameters are passed on the stack. In addition, the parameters are passed on the stack in the following cases:

- Structure types: `struct`, `union`, and classes
- Unnamed parameters to variable length functions; in other words, functions declared as `foo(param1, ...)`, for example `printf`.

Note: Interrupt functions cannot take any parameters.

Hidden parameters

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

- If the function returns a structure, the memory location where to store the structure is passed in the register R12 as a hidden parameter.
- If the function is a non-static C++ member function, then the `this` pointer is passed as the first parameter (but placed after the return structure pointer, if there is one). The reason for the requirement that the member function must be non-static is that static member methods do not have a `this` pointer.

Register parameters

The following registers are available for passing parameters:

Parameters	Passed in registers
8-bit values	R12, R14
16-bit values	R12, R14
32-bit values	R13 : R12, R15 : R14

Table 19: Registers used for passing parameters

Parameters	Passed in registers
64-bit values	R15 : R14 : R13 : R12, R11 : R10 : R9 : R8

Table 19: Registers used for passing parameters (Continued)

Note: When compiling for the MSP430X architecture, only the lower 16 bits of the registers are used.

The assignment of registers to parameters is a straightforward process. The first parameter is assigned to R12 or R13 : R12, depending on the size of the parameter. The second parameter is passed in R14 or R15 : R14. Should there be no more available registers, the parameter is passed on the stack.

Stack parameters and layout

Stack parameters are stored in the main memory starting at the location pointed to by the stack pointer. Below the stack pointer (towards low memory) there is free space that the called function can use. The first stack parameter is stored at the location pointed to by the stack pointer. The next one is stored at the next even location on the stack. It is the responsibility of the caller to remove the parameters from the stack by restoring the stack pointer.

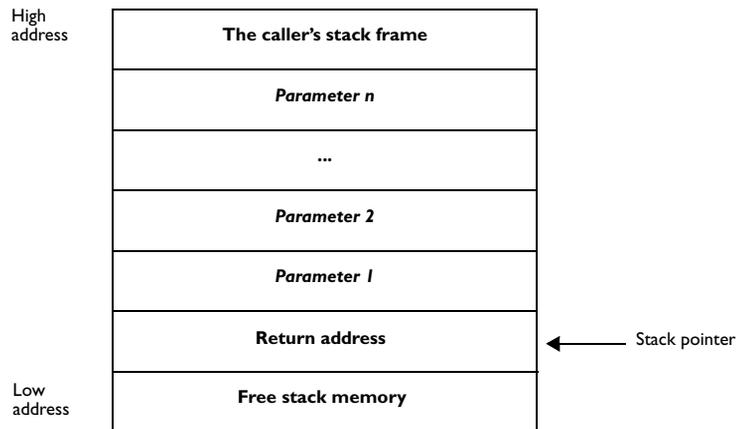


Figure 2: Stack image after the function call

Note: The number of bytes reserved for the return address depends on the `--core` option, see *Calling functions*, page 89.

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.

Note: An interrupt function must have the return type `void`.

A function returns by performing any of the following instructions:

- `RET` instruction, when compiling for the MSP430 architecture
- `RETA`, when compiling for the MSP430X architecture
- `RETI`, for interrupt functions regardless of which architecture is being used.

Registers used for returning values

The registers available for returning values are:

Return values	Passed in registers
8-bit values	R12
16-bit values	R12
32-bit values	R13 : R12
64-bit values	R15 : R14 : R13 : R12

Table 20: Registers used for returning values

Stack layout

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

Return value pointer

If a structure is returned, the caller passes a pointer to a location where the called function should write the result. The pointer should be passed in the register `R12`.

The called function must return the pointer in the register `R12`.

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 R12, and the return value is passed back to its caller in the register R12.

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

```
ADD.W    #1,R12
RETA                                ; For the MSP430X architecture
```

Example 2

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

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

The calling function must reserve 4 bytes on the top of the stack and copy the contents of the `struct` to that location. The integer parameter `y` is passed in the register R12. The return value is passed back to its caller in the register R12.

Example 3

The function below will return a `struct`.

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

It is the responsibility of the calling function to allocate a memory location—typically on the stack—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 R12. The caller assumes that this register remain untouched. The parameter `x` is passed in R14.

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

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

In this case, the return value is a pointer, so there is no hidden parameter. The parameter `x` is passed in R12 and the return value is also returned in R12.

FUNCTION DIRECTIVES

Note: This type of directives are primarily intended to support static overlay, a feature which is useful in some smaller microcontrollers. The MSP430 IAR C/C++ Compiler does not use static overlay, as it has no use for it.

The function directives `FUNCTION`, `ARGFRAME`, `LOCFRAME`, and `FUNCALL` are generated by the MSP430 IAR C/C++ Compiler to pass information about functions and function calls to the IAR XLINK Linker. These directives can be seen if you use the compiler option **Assembler file** (`-lA`) to create an assembler list file.

For reference information about the function directives, see the *MSP430 IAR Assembler Reference Guide*.

Calling functions

When calling an assembler module from C modules, it is important to match the calling convention which is different depending on the `--core` option.

When C modules are compiled with the `--core=430` option, they use the `CALL` instruction to call an external function, and with `--core=430X` they use the `CALLA` instruction. These two call instructions have different stack layouts. The `CALL` instruction pushes a 2-byte return address on the stack whereas `CALLA` pushes 4 bytes. This must be matched in your assembler routine by using the corresponding `RET` and `RETA` return instructions, or the function will not return properly which leads to a corrupt stack.

Note: Interrupt functions written in assembler are not affected, because all interrupt routines must return using the `RETI` instruction, regardless of which architecture that you are using.



Because the calling convention differs slightly between the two architectures, you can define the runtime attribute `__core` in all your assembler routines, to avoid inconsistency. Use one of the following lines:

```
RTMODEL "__core"="64kb"
RTMODEL "__core"="1MB"
```

Using this module consistency check, 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 64.

Call frame information

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

If you want the call stack to be available when debugging a routine written in assembler language, you must supply equivalent debug information in your assembler source using the assembler directive `CFI`. This directive is described in detail in the *MSP430 IAR Assembler Reference Guide*.

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

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

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

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

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

Resource	Description
CFA	The call frames of the stack
R4–R15	Normal registers
R4L–R15L	Lower 16 bits, when compiling for the MSP430X architecture
R4H–R15H	Higher 4 bits, when compiling for the MSP430X architecture
SP	The stack pointer
SR	The processor state register
PC	The program counter

Table 21: Call frame information resources defined in a names block

Example

The following is an example of an assembler routine that stores a permanent register as well as the return register to the stack when compiling for the MSP430 architecture:

```
#include "cfi.m43"

XCFI_NAMES myNames
XCFI_COMMON myCommon, myNames
MODULE cfiexample
PUBLIC cfiexample

RSEG CODE:CODE:NOROOT(1)
CFI Block myBlock Using myCommon
CFI Function 'cfiexample'

// The common block does not declare the scratch
// registers as undefined.
CFI R12 Undefined
cfiexample:
PUSH R11
CFI R11 Frame(CFA, -4)
CFI CFA SP+4

// Do something useless just to demonstrate that the call
// stack window works properly. You can check this by
// single-stepping in the Disassembly window and
// double-clicking on the parent function in the
// Call Stack window.
MOV #0, R11
POP R11
CFI R11 SameValue
CFI CFA SP+2

// Do something else.
MOV #0, R12
RET
CFI ENDBLOCK myBlock

ENDMOD
END
```


Using C++

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

Overview

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

STANDARD EMBEDDED C++

The following C++ features are supported:

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

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

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

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

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

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

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

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

EXTENDED EMBEDDED C++

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

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

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

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

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

ENABLING C++ SUPPORT



In the MSP430 IAR C/C++ Compiler, the default language is C. To be able to compile files written in Embedded C++, you must use the `--ec++` compiler option. See `--ec++`, page 132. 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 132.



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

Feature descriptions

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

CLASSES

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

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

For further information about attributes, see *Type qualifiers*, page 156.

Example

```
class A {
    public:
        static __no_init int i @ 60; //Located at address 60
};
```

FUNCTIONS

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

Example

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

fpC f1;
```

```

fpCpp f2;
void f(fpC);

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

```

TEMPLATES

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

The standard template library

The STL (standard template library) delivered with the product is tailored for *Extended EC++*, as described in *Extended Embedded C++*, page 94.

STL and the IAR C-SPY® Debugger

C-SPY has built-in display support for STL containers and some STL iterators. The logical structure of containers is presented in the watch views in a comprehensive way that is easy to understand and follow.

Note: To be able to watch STL containers with many elements in a comprehensive way, the **STL container expansion** option—available by choosing **Tools>Options>Debugger**—is set to display only a small number of items at first.

VARIANTS OF CASTS

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

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

MUTABLE

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

NAMESPACE

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

THE STD NAMESPACE

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

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

USING INTERRUPTS AND EC++ DESTRUCTORS

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

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

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

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

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

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

C++ language extensions

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

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

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

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

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

    int a[size];
};
```

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

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

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

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

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

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

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

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

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

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

Efficient coding for embedded applications

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

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 MSP430 IAR C/C++ Compiler allows you to specify whether generated code should be optimized for size or for speed, at a selectable optimization level. The purpose of optimization is to reduce the code size and to improve the execution speed. When only one of these two goals can be reached, the compiler prioritizes according to the settings you specify. Note that one optimization sometimes enables other optimizations to be performed, and an application may become smaller even when optimizing for speed rather than size.

The following table describes the optimization levels:

Optimization level	Description
None (Best debug support) *	Dead code elimination Redundant label elimination Redundant branch elimination
Low	All the above except that variables only live for as long as they are needed
Medium	All the above Live-dead analysis and optimization Code hoisting Register content analysis and optimization Common subexpression elimination
High (Maximum optimization)	All the above Peephole optimization Cross jumping Cross call Loop unrolling Function inlining Code motion Type-based alias analysis

Table 22: Compiler optimization levels

* **Variables live through their entire scope.**

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 *MSP430 IAR Embedded Workbench® IDE User Guide* for information about the compiler options available in the IAR Embedded Workbench IDE. Refer to *optimize*, page 189, for information about the pragma directives that can be used for specifying optimization type and level.

FINE-TUNING ENABLED TRANSFORMATIONS

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

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

Common subexpression elimination

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

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

To read more about the command line option, see `--no_cse`, page 138.

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 140.

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 139.

Code motion

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

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

Type-based alias analysis

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

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

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

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

To read more about the command line option, see `--no_tbaa`, page 139.

Example

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

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

Selecting data types and placing data in memory

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

USING EFFICIENT DATA TYPES

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

- Use small data types.
- Try to avoid 64-bit data types, such as `double` and `long long`.
- Bitfields with sizes other than 1 bit should be avoided because they will result in inefficient code compared to bit operations.
- When using arrays, it is more efficient if the type of the index expression matches the index type of the memory of the array. For the MSP430 IAR C/C++ Compiler, this is `int`.
- Using floating-point types is very inefficient, both in terms of code size and execution speed. If possible, consider using integer operations instead.
- Declaring a pointer to `const` data tells the calling function that the data pointed to will not change, which opens for better optimizations.

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

Floating-point types

Using floating-point types on a microprocessor without a math coprocessor is very inefficient, both in terms of code size and execution speed. The MSP430 IAR C/C++ Compiler supports two floating-point formats—32 and 64 bits. The 32-bit floating-point type `float` is more efficient in terms of code size and execution speed. However, the 64-bit format `double` supports higher precision and larger numbers.

In the MSP430 IAR C/C++ Compiler, the floating-point type `float` always uses the 32-bit format. The format used by the `double` floating-point type depends on the setting of the `--double` compiler option.

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

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

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

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

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

REARRANGING ELEMENTS IN A STRUCTURE

The MSP430 microcontroller requires that when accessing data in memory, the data must be aligned. Each element in a structure needs to be aligned according to its specified type requirements. This means that the compiler must insert *pad bytes* if the alignment is not correct.

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

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

For information about alignment requirements, see *Alignment*, page 149.

There are two ways to solve the problem:

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

For further details about the `#pragma pack` directive, see *pack*, page 190.

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 MSP430 IAR C/C++ Compiler they can be used in C if language extensions are enabled.



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



Use the `-e` compiler option to enable language extensions. See *-e*, page 132, for additional information.

Example

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

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

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

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

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

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

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

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

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

Writing efficient code

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

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

- Local variables—auto variables and parameters—are preferred over static or global variables. The reason is that the optimizer must assume, for example, that called functions may modify non-local variables. When the life spans for local variables end, the previously occupied memory can then be reused. Globally declared variables will occupy data memory during the whole program execution.

- Avoid taking the address of local variables using the `&` operator. There are two main reasons why this is inefficient. First, the variable must be placed in memory, and thus cannot be placed in a processor register. This results in larger and slower code. Second, the optimizer can no longer assume that the local variable is unaffected over function calls.
- Module-local variables—variables that are declared `static`—are preferred over global variables. Also avoid taking the address of frequently accessed static variables.
- The compiler is capable of inlining functions. This means that instead of calling a function, the compiler inserts the content of the function at the location where the function was called. The result is a faster, but often larger, application. Also, inlining may enable further optimizations. The compiler often inlines small functions declared `static`. The use of the `#pragma inline` directive and the C++ keyword `inline` gives you fine-grained control, and it is the preferred method compared to the traditional way of using preprocessor macros. This feature can be disabled using the `--no_inline` command line option; see *--no_inline*, page 139.
- 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 77.

SAVING STACK SPACE AND RAM MEMORY

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

- If stack space is limited, avoid long call chains and recursive functions.
- Avoid using large non-scalar types, such as structures, as parameters or return type; in order to save stack space, you should instead pass them as pointers or, in C++, as references
- Use the `--reduced_stack_space` option. This will eliminate holes in the stack resulting from normal optimizations.

FUNCTION PROTOTYPES

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

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

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

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

Prototyped style

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

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

Kernighan & Ritchie style

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

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

INTEGER TYPES AND BIT NEGATION

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

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

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

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

PROTECTING SIMULTANEOUSLY ACCESSED VARIABLES

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

To mark a variable properly, use the `volatile` keyword. This informs the compiler, among other things, that the variable can be changed from other threads. The compiler will then avoid optimizing on the variable (for example, keeping track of the variable in registers), will not delay writes to it, and be careful accessing the variable only the number of times given in the source code. To read more about the `volatile` type qualifier, see *Declaring objects volatile*, page 156.

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



Protecting the eeprom write mechanism

A typical example of when it can be necessary to use the `__monitor` keyword is when protecting the eeprom write mechanism, which can be used from two threads (for example, main code and interrupts).

ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for a number of MSP430 derivatives are included in the MSP430 IAR C/C++ Compiler delivery. The header files are named `iodevice.h` and define the processor-specific memory-mapped peripheral units.

Note: Assembler files must use the header files called `mspdevice.h`.

The header file contains definitions that include bitfields, so individual bits can be accessed. The following example is from `io430x14x.h`:

```
/* Watchdog Timer Control */
__no_init volatile union
{
    unsigned short WDTCTL;
```

```

struct
{
    unsigned short WDTIS0      : 1;
    unsigned short WDTIS1      : 1;
    unsigned short WDTSSSEL     : 1;
    unsigned short WDCNTCL      : 1;
    unsigned short WDTTMSEL     : 1;
    unsigned short WDTNMI       : 1;
    unsigned short WDTNMIES     : 1;
    unsigned short WDTTHOLD     : 1;
    unsigned short              : 8;
} WDTCTL_bit;
} @ 0x0120;

enum {
    WDTIS0      = 0x0001,
    WDTIS1      = 0x0002,
    WDTSSSEL     = 0x0004,
    WDCNTCL      = 0x0008,
    WDTTMSEL     = 0x0010,
    WDTNMI       = 0x0020,
    WDTNMIES     = 0x0040,
    WDTTHOLD     = 0x0080
};

#define WDTPW          (0x5A00)

```

By including the appropriate include file in your source code, you make it possible to access either the object or any individual bit (or bitfields) from C code as follows:

```

/* Object access */
WDTCTL = 0x1234;

/* Bitfield accesses */
WDTCTL_bit.WDTSSSEL = 1;

```

If more than one bit must be written to a memory-mapped peripheral unit at the same time, for instance to stop the watchdog timer, the defined bit constants can be used instead, for example:

```

WDTCTL = WDTPW + WDTTHOLD;          /* Stop watchdog timer */

```

You can also use the header files as templates when you create new header files for other MSP430 derivatives. For details about the @ operator, see *Located data*, page 31.

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 176. Note that to use this keyword, language extensions must be enabled; see `-e`, page 132. For information about the `#pragma object_attribute`, see page 189.

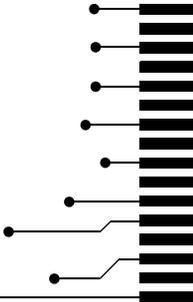
EFFICIENT SWITCH STATEMENTS

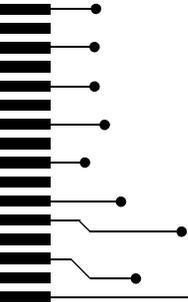
The compiler provides a way to generate very efficient code for switch statements when it is known that the value in the expression is even and within a specific limit. This can for example be used for writing efficient interrupt service routines that use the Interrupt Vector Generators Timer A, Timer B, the I²C module, and the ADC12 module. For more information, see *Interrupt Vector Generator interrupt functions*, page 17.

Part 2. Compiler reference

This part of the MSP430 IAR C/C++ Compiler Reference Guide contains the following chapters:

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





Compiler usage

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

Compiler invocation

You can use the compiler either from the IAR Embedded Workbench IDE or from the command line. Refer to the *MSP430 IAR Embedded Workbench® IDE User Guide* for information about using the compiler from the IAR Embedded Workbench IDE.

INVOCATION SYNTAX

The invocation syntax for the compiler is:

```
icc430 [options][sourcefile][options]
```

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

```
icc430 prog --debug
```

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

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

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

PASSING OPTIONS TO THE COMPILER

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

- Directly from the command line

Specify the options on the command line after the `icc430` command, either before or after the source filename; see *Invocation syntax*, page 115.
- Via environment variables

The compiler automatically appends the value of the environment variables to every command line; see *Environment variables*, page 116.
- Via a text file by using the `-f` option; see *-f*, page 133.

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

ENVIRONMENT VARIABLES

The following environment variables can be used with the MSP430 IAR C/C++ Compiler:

Environment variable	Description
<code>C_INCLUDE</code>	Specifies directories to search for include files; for example: <code>C_INCLUDE=c:\program files\iar systems\embedded workbench 4.n\430\inc;c:\headers</code>
<code>QCC430</code>	Specifies command line options; for example: <code>QCC430=-lA asm.lst -z9</code>

Table 23: Environment variables

Include file search procedure

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

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

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

it searches the following directories for the file to include:

- 1 The directories specified with the `-I` option, in the order that they were specified, see *-l*, page 134.
- 2 The directories specified using the `C_INCLUDE` environment variable, if any, see *Environment variables*, page 116.

- 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:

```
icc430 ..\src\src.c -I..\include -I..\debugconfig
```

Then the following directories are searched in the order listed below for the file `config.h`, which in this example is located in the `dir\debugconfig` directory:

<code>dir\include</code>	Current file is <code>src.h</code> .
<code>dir\src</code>	File including current file (<code>src.c</code>).
<code>dir\include</code>	As specified with the first <code>-I</code> option.
<code>dir\debugconfig</code>	As specified with the second <code>-I</code> option.

Use angle brackets for standard header files, like `stdio.h`, and double quotes for files that are part of your application.

Compiler output

The compiler can produce the following output:

- A linkable object file

The object files produced by the compiler uses a proprietary format called UBROF, which stands for Universal Binary Relocatable Object Format. By default, the object file has the filename extension `r43`.

- Optional list files

Different types of list files can be specified using the compiler option `-l`, see `-l`, page 134. By default, these files will have the filename extension `lst`.

- Optional preprocessor output files

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

- Diagnostic messages

Diagnostic messages are directed to `stderr` and displayed on the screen, as well as printed in an optional list file. To read more about diagnostic messages, see *Diagnostics*, page 119.

- Error return codes

These codes provide status information to the operating system which can be tested in a batch file, see *Error return codes*, page 118.

- Size information

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

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

Error return codes

The MSP430 IAR C/C++ Compiler returns status information to the operating system which can be tested in a batch file.

The following command line error codes are supported:

Code	Description
0	Compilation successful, but there may have been warnings.
1	There were warnings, provided that the option <code>--warnings_affect_exit_code</code> was used.
2	There were errors.
3	There were fatal errors making the compiler abort.

Table 24: Error return codes

Diagnostics

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

MESSAGE FORMAT

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

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

where *filename* is the name of the source file in which the error was encountered, *linenumber* is the line number at which the compiler detected the error, *level* is the level of seriousness of the diagnostic, *tag* is a unique tag that identifies the diagnostic message, and *message* is a self-explanatory message, possibly several lines long.

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

Use the option `--diagnostics_tables` to list all possible compiler diagnostic messages in a named file. In addition, you can find all messages specific to the MSP430 IAR C/C++ Compiler in the readme file `ICC430_msg.htm`.

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 145.

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 141.

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 124, for a description of the compiler options that are available for setting severity levels.

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

INTERNAL ERROR

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

Internal error: *message*

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

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

Compiler options

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

Compiler options syntax

Compiler options are parameters you can specify to change the default behavior of the compiler. You can specify compiler options from the command line—which is described in more detail in this section—and from within the IAR Embedded Workbench IDE. Refer to the *MSP430 IAR Embedded Workbench® IDE User Guide* for information about the compiler options available in the IAR Embedded Workbench IDE and how to set them.

TYPES OF OPTIONS

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

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

For information about the different methods for passing options, see *Passing options to the compiler*, page 116.

RULES FOR SPECIFYING PARAMETERS

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

Rules for optional parameters

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

```
-z or -z3
```

For options with a long name and an optional parameter, any parameter should be specified with a preceding equal sign (=), for example:

```
--misrac=n
```

Rules for mandatory parameters

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

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

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

```
--diagnostics_tables=filename
```

or

```
-diagnostics_tables filename
```

Rules for options with both optional and mandatory parameters

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

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

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

```
-lA filename
```

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

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

Rules for specifying a filename or directory as parameters

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

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

```
icc430 prog -l ..\listings\list.lst
```

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

```
icc430 prog -l ..\listings\
```

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

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

```
icc430 prog -l .
```

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

```
icc430 prog -l -
```

Additional rules

In addition, the following rules apply:

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

```
icc430 prog -l ---r
```

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

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

Alternatively, the option may be repeated for each argument, for example:

```
--diag_warning=Be0001
```

```
--diag_warning=Be0002
```

Options summary

The following table summarizes the compiler command line options:

Command line option	Description
<code>--char_is_signed</code>	Treats <code>char</code> as signed
<code>--core={430 430X}</code>	Specifies the architecture
<code>-D symbol[=value]</code>	Defines preprocessor symbols
<code>--debug</code>	Generates debug information
<code>--dependencies[={i m}]</code> <code>{filename directory}</code>	Lists file dependencies
<code>--diag_error=tag[, tag, ...]</code>	Treats these as errors
<code>--diag_remark=tag[, tag, ...]</code>	Treats these as remarks
<code>--diag_suppress=tag[, tag, ...]</code>	Suppresses these diagnostics
<code>--diag_warning=tag[, tag, ...]</code>	Treats these as warnings
<code>--diagnostics_tables</code> <code>{filename directory}</code>	Lists all diagnostic messages
<code>--dlib_config filename</code>	Determines the library configuration file
<code>--double={32 64}</code>	Forces the compiler to use 32-bit or 64-bit doubles
<code>-e</code>	Enables language extensions
<code>--ec++</code>	Enables Embedded C++ syntax
<code>--eec++</code>	Enables Extended Embedded C++ syntax
<code>--enable_multibytes</code>	Enables support for multibyte characters in source files
<code>--error_limit=n</code>	Specifies the allowed number of errors before compilation stops
<code>-f filename</code>	Extends the command line
<code>--header_context</code>	Lists all referred source files and header files
<code>-I path</code>	Specifies the include file path
<code>-l[a A b B c C D][N][H]</code> <code>{filename directory}</code>	Creates a list file
<code>--library_module</code>	Creates a library module
<code>--lock_r4</code>	Excludes R4 from use
<code>--lock_r5</code>	Excludes R5 from use

Table 25: Compiler options summary

Command line option	Description
<code>--migration_preprocessor_extensions</code>	Extends the preprocessor
<code>--misrac[={n,o-p,... all required}]</code>	Enables MISRA C-specific error messages
<code>--misrac_verbose</code>	Enables verbose logging of MISRA C checking
<code>--module_name=name</code>	Sets the object module name
<code>--no_code_motion</code>	Disables code motion optimization
<code>--no_cse</code>	Disables common subexpression elimination
<code>--no_inline</code>	Disables function inlining
<code>--no_tbaa</code>	Disables type-based alias analysis
<code>--no_typedefs_in_diagnostics</code>	Disables the use of typedef names in diagnostics
<code>--no_unroll</code>	Disables loop unrolling
<code>--no_warnings</code>	Disables all warnings
<code>--no_wrap_diagnostics</code>	Disables wrapping of diagnostic messages
<code>-o {filename directory}</code>	Sets the object filename
<code>--omit_types</code>	Excludes type information
<code>--only_stdout</code>	Uses standard output only
<code>--pic</code>	Produces position-independent code
<code>--preinclude includefile</code>	Includes an include file before reading the source file
<code>--preprocess[={c}[n][l]] {filename directory}</code>	Generates preprocessor output
<code>--public_equ symbol[=value]</code>	Defines a global named assembler label
<code>-r</code>	Generates debug information
<code>--reduce_stack_usage</code>	Reduces stack usage
<code>--regvar_r4</code>	Reserves R4 for use by global register variables
<code>--regvar_r5</code>	Reserves R5 for use by global register variables
<code>--remarks</code>	Enables remarks
<code>--require_prototypes</code>	Verifies that functions are declared before they are defined
<code>-s[2 3 6 9]</code>	Optimizes for speed

Table 25: Compiler options summary (Continued)

Command line option	Description
--save_reg20	Declares all interrupt functions __save_reg20 by default
--silent	Sets silent operation
--strict_ansi	Checks for strict compliance with ISO/ANSI C
--warnings_affect_exit_code	Warnings affects exit code
--warnings_are_errors	Warnings are treated as errors
-z[2 3 6 9]	Optimizes for size

Table 25: Compiler options summary (Continued)

Descriptions of options

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



Note that if you use the options page **Extra Options** to specify specific command line options, the IAR Embedded Workbench IDE does not perform an instant check for consistency problems like conflicting options, reuse of options, or use of irrelevant options.

--char_is_signed

Syntax

--char_is_signed

Description

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

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



Project>Options>C/C++ Compiler>Language>Plain 'char' is

--core

Syntax	<code>--core={430 430X}</code>	
Parameters	430 (default)	For devices based on the MSP430 architecture
	430X	For devices based on the MSP430X architecture
Description	<p>Use this option to select the architecture for which the code is to be generated. If you do not use the option, the compiler generates code for the MSP430 architecture by default.</p> <p>The object code that is generated when compiling for the MSP430 architecture can be executed also by the MSP430X architecture. The other way around is not possible.</p>	



Project>Options>General Options>Target>Device

-D

Syntax	<code>-D <i>symbol</i>[=<i>value</i>]</code>	
Parameters	<i>symbol</i>	The name of the preprocessor symbol
	<i>value</i>	The value of the preprocessor symbol
Description	<p>Use this option to define a preprocessor symbol. If no value is specified, 1 is used. This option can be used one or more times on the command line.</p> <p>The option <code>-D</code> has the same effect as a <code>#define</code> statement at the top of the source file:</p> <pre>-D<i>symbol</i></pre> <p>is equivalent to:</p> <pre>#define <i>symbol</i> 1</pre> <p>In order to get the equivalence of:</p> <pre>#define FOO</pre> <p>specify the <code>=</code> sign but nothing after, for example:</p> <pre>-DFOO=</pre>	



Project>Options>C/C++ Compiler>Preprocessor>Defined symbols

--debug, -r

Syntax	--debug -r
Description	Use the --debug or -r option to make the compiler include information in the object modules that is useful to the IAR C-SPY® Debugger and other symbolic debuggers. Note: Including debug information will make the object files larger than otherwise.



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

--dependencies

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

Parameters

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

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 122.

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

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

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

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

```
foo.r43: c:\iar\product\include\stdio.h
foo.r43: d:\myproject\include\foo.h
```

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

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

```
%.r43 : %.c
$(ICC) $(ICCFLAGS) $< --dependencies=m $*.d
```

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

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

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

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



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

--diag_error

Syntax

```
--diag_error=tag[, tag, ...]
```

Parameters

tag The number of a diagnostic message, for example the message number `Pe117`

Description

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



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

--diag_remark

Syntax

```
--diag_remark=tag[, tag, ...]
```

Parameters

tag The number of a diagnostic message, for example the message number `Pe177`

Description

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

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



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

--diag_suppress

Syntax	<code>--diag_suppress=tag[, tag, ...]</code>	
Parameters	<i>tag</i>	The number of a diagnostic message, for example the message number <code>Pe117</code>
Description	Use this option to suppress certain diagnostic messages. These messages will not be displayed. This option may be used more than once on the command line.	



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

--diag_warning

Syntax	<code>--diag_warning=tag[, tag, ...]</code>	
Parameters	<i>tag</i>	The number of a diagnostic message, for example the message number <code>Pe826</code>
Description	Use this option to reclassify certain diagnostic messages as warnings. A warning indicates an error or omission that is of concern, but which will not cause the compiler to stop before compilation is completed. This option may be used more than once on the command line.	



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

--diagnostics_tables

Syntax	<code>--diagnostics_tables {filename directory}</code>	
Parameters	For information about specifying a filename or a directory, see <i>Rules for specifying a filename or directory as parameters</i> , page 122.	
Description	Use this option to list all possible diagnostic messages in a named file. This can be very convenient, for example if you have used a pragma directive to suppress or change the severity level of any diagnostic messages, but forgot to document why.	



This option cannot be given together with other options.

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

--dlib_config

Syntax	<code>--dlib_config filename</code>
Parameters	For information about specifying a filename, see <i>Rules for specifying a filename or directory as parameters</i> , page 122.
Description	<p>Each runtime library has a corresponding library configuration file. Use this option to specify the library configuration file for the compiler. Make sure that you specify a configuration file that corresponds to the library you are using.</p> <p>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 <code>430\lib</code>. For examples and a list of prebuilt runtime libraries, see <i>Using a prebuilt library</i>, page 42.</p> <p>If you build your own customized runtime library, you should also create a corresponding customized library configuration file, which must be specified to the compiler. For more information, see <i>Building and using a customized library</i>, page 48.</p> <p>Note: This option only applies to the IAR DLIB runtime environment.</p>



To set related options, choose:

Project>Options>General Options>Library Configuration

--double

Syntax	<code>--double={32 64}</code>				
Parameters	<table> <tr> <td>32 (default)</td> <td>32-bit doubles are used</td> </tr> <tr> <td>64</td> <td>64-bit doubles are used</td> </tr> </table>	32 (default)	32-bit doubles are used	64	64-bit doubles are used
32 (default)	32-bit doubles are used				
64	64-bit doubles are used				
Description	Use this option to select the precision used by the compiler for representing the floating-point types <code>double</code> and <code>long double</code> . The compiler can use either 32-bit or 64-bit precision. By default, the compiler uses 32-bit precision.				
See also	<i>Floating-point types</i> , page 152.				



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

-e

Syntax `-e`

Description In the command line version of the MSP430 IAR C/C++ Compiler, language extensions are disabled by default. If you use language extensions such as extended keywords and anonymous structs and unions in your source code, you must enable them by using this option.

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

See also The chapter *Compiler extensions*.



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

Note: By default, this option is enabled in the IAR Embedded Workbench IDE.

--ec++

Syntax `--ec++`

Description In the MSP430 IAR C/C++ Compiler, the default language is C. If you use Embedded C++, you must use this option to set the language the compiler uses to Embedded C++.



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

--eec++

Syntax `--eec++`

Description In the MSP430 IAR C/C++ Compiler, the default language is C. If you take advantage of Extended Embedded C++ features like namespaces or the standard template library in your source code, you must use this option to set the language the compiler uses to Extended Embedded C++.

See also *Extended Embedded C++*, page 94.



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

--enable_multibytes

Syntax	<code>--enable_multibytes</code>
Description	By default, multibyte characters cannot be used in C or C++ source code. Use this option to make multibyte characters in the source code be interpreted according to the host computer's default setting for multibyte support. Multibyte characters are allowed in C and C++ style comments, in string literals, and in character constants. They are transferred untouched to the generated code.



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

--error_limit

Syntax	<code>--error_limit=n</code>
Parameters	<i>n</i> The number of errors before the compiler stops the compilation. <i>n</i> must be a positive integer; 0 indicates no limit.
Description	Use the <code>--error_limit</code> option to specify the number of errors allowed before the compiler stops the compilation. By default, 100 errors are allowed.



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

-f

Syntax	<code>-f filename</code>
Parameters	For information about specifying a filename, see <i>Rules for specifying a filename or directory as parameters</i> , page 122.
Descriptions	Use this option to make the compiler read command line options from the named file, with the default filename extension <code>.xcl</code> . 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.

a	Assembler list file
b	Basic assembler list file. This file has the same contents as a list file produced with <code>-la</code> , except that no extra compiler-generated information (runtime model attributes, call frame information, frame size information) is included *
B	Basic assembler list file. This file has the same contents as a list file produced with <code>-lA</code> , except that no extra compiler generated information (runtime model attributes, call frame information, frame size information) is included *
c	C or C++ list file
C (default)	C or C++ list file with assembler source as comments
D	C or C++ list file with assembler source as comments, but without instruction offsets and hexadecimal byte values
N	No diagnostics in file
H	Include source lines from header files in output. Without this option, only source lines from the primary source file are included

* This makes the list file less useful as input to the assembler, but more useful for reading by a human.

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 122.

Description Use this option to generate an assembler or C/C++ listing to a file. Note that this option can be used one or more times on the command line.



To set related options, choose:

Project>Options>C/C++ Compiler>List

--library_module

Syntax `--library_module`

Description Use this option to make the compiler generate a library module rather than a program module. A program module is always included during linking. A library module will only be included if it is referenced in your program.



Project>Options>C/C++ Compiler>Output>Output file>Library

--lock_r4

Syntax `--lock_R4`

Description Use this option to exclude the register R4 from use by the compiler. This makes the module linkable with both modules that use R4 as `__regvar` and modules that does not define its R4 usage. Use this option if the R4 registers is used by another tool, for example a ROM-monitor debugger.



Project>Options>C/C++ Compiler>Code>R4 utilization>Not used

--lock_r5

Syntax `--lock_R5`

Description Use this option to exclude the register R5 from use by the compiler. This makes the module linkable with both modules that use R5 as `__regvar` and modules that does not define its R5 usage. Use this option if the R5 registers is used by another tool, for example a ROM-monitor debugger.



Project>Options>C/C++ Compiler>Code>R5 utilization>Not used

--migration_preprocessor_extensions

Syntax `--migration_preprocessor_extensions`

Description If you need to migrate code from an earlier IAR Systems C or C/C++ compiler, you may want to use this option. Use this option to use the following in preprocessor expressions:

- Floating-point expressions
- Basic type names and `sizeof`
- All symbol names (including typedefs and variables).

Note: If you use this option, not only will the compiler accept code that is not standards conformant, but it will also reject some code that *does* conform to the standard.

Important! Do not depend on these extensions in newly written code, because support for them may be removed in future compiler versions.



Project>Options>C/C++ Compiler>Language>Enable IAR migration preprocessor extensions

--misrac

Syntax	<code>--misrac[={<i>n</i>, <i>o-p</i>, ... all required}]</code>												
Parameters	<table> <tr> <td><code>--misrac=<i>n</i></code></td> <td>Enables checking for the MISRA C rule with number <i>n</i></td> </tr> <tr> <td><code>--misrac=<i>o</i>, <i>n</i></code></td> <td>Enables checking for the MISRA C rules with numbers <i>o</i> and <i>n</i></td> </tr> <tr> <td><code>--misrac=<i>o-p</i></code></td> <td>Enables checking for all MISRA C rules with numbers from <i>o</i> to <i>p</i></td> </tr> <tr> <td><code>--misrac=<i>m</i>, <i>n</i>, <i>o-p</i></code></td> <td>Enables checking for MISRA C rules with numbers <i>m</i>, <i>n</i>, and from <i>o</i> to <i>p</i></td> </tr> <tr> <td><code>--misrac=all</code></td> <td>Enables checking for all MISRA C rules</td> </tr> <tr> <td><code>--misrac=required</code></td> <td>Enables checking for all MISRA C rules categorized as required</td> </tr> </table>	<code>--misrac=<i>n</i></code>	Enables checking for the MISRA C rule with number <i>n</i>	<code>--misrac=<i>o</i>, <i>n</i></code>	Enables checking for the MISRA C rules with numbers <i>o</i> and <i>n</i>	<code>--misrac=<i>o-p</i></code>	Enables checking for all MISRA C rules with numbers from <i>o</i> to <i>p</i>	<code>--misrac=<i>m</i>, <i>n</i>, <i>o-p</i></code>	Enables checking for MISRA C rules with numbers <i>m</i> , <i>n</i> , and from <i>o</i> to <i>p</i>	<code>--misrac=all</code>	Enables checking for all MISRA C rules	<code>--misrac=required</code>	Enables checking for all MISRA C rules categorized as required
<code>--misrac=<i>n</i></code>	Enables checking for the MISRA C rule with number <i>n</i>												
<code>--misrac=<i>o</i>, <i>n</i></code>	Enables checking for the MISRA C rules with numbers <i>o</i> and <i>n</i>												
<code>--misrac=<i>o-p</i></code>	Enables checking for all MISRA C rules with numbers from <i>o</i> to <i>p</i>												
<code>--misrac=<i>m</i>, <i>n</i>, <i>o-p</i></code>	Enables checking for MISRA C rules with numbers <i>m</i> , <i>n</i> , and from <i>o</i> to <i>p</i>												
<code>--misrac=all</code>	Enables checking for all MISRA C rules												
<code>--misrac=required</code>	Enables checking for all MISRA C rules categorized as required												
Description	<p>Use this option to enable the compiler to check for deviations from the rules described in the MISRA <i>Guidelines for the Use of the C Language in Vehicle Based Software</i> (1998). By using one or more arguments with the option, you can restrict the checking to a specific subset of the MISRA C rules. If the compiler is unable to check for a rule, specifying the option for that rule has no effect. For instance, MISRA C rule 15 is a documentation issue, and the rule is not checked by the compiler. As a consequence, specifying <code>--misrac=15</code> has no effect.</p>												



To set related options, choose:

Project>Options>General Options>MISRA C or **Project>Options>C/C++ Compiler>MISRA C**

--misrac_verbose

Syntax	<code>--misrac_verbose</code>
Description	<p>Use this option to generate a MISRA C log during compilation and linking. This is a list of the rules that are enabled—but not necessarily checked—and a list of rules that are actually checked.</p> <p>If this option is enabled, the compiler displays a text at sign-on that shows both enabled and checked MISRA C rules.</p>



Project>Options>General Options>MISRA C>Log MISRA C Settings

--module_name

Syntax	<code>--module_name=<i>name</i></code>
Parameters	<i>name</i> An explicit object module name
Description	<p>Normally, the internal name of the object module is the name of the source file, without a directory name or extension. Use this option to specify an object module name explicitly.</p> <p>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.</p>



Project>Options>C/C++ Compiler>Output>Object module name

--no_code_motion

Syntax	<code>--no_code_motion</code>
Description	<p>Use this option to disable code motion optimizations. These optimizations, which are performed at optimization levels 6 and 9, normally reduce code size and execution time. However, the resulting code may be difficult to debug.</p> <p>Note: This option has no effect at optimization levels below 6.</p>



Project>Options>C/C++ Compiler>Code>Enable transformations>Code motion

--no_cse

Syntax	<code>--no_cse</code>
Description	<p>Use this option to disable common subexpression elimination. At optimization levels 6 and 9, the compiler avoids calculating the same expression more than once. This optimization normally reduces both code size and execution time. However, the resulting code may be difficult to debug.</p> <p>Note: This option has no effect at optimization levels below 6.</p>



Project>Options>C/C++ Compiler>Code>Enable transformations>Common subexpression elimination

--no_inline

Syntax `--no_inline`

Description Use this option to disable function inlining. Function inlining means that a simple function, whose definition is known at compile time, is integrated into the body of its caller to eliminate the overhead of the call.

This optimization, which is performed at optimization level 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 than for size.

Note: This option has no effect at optimization levels below 9.



Project>Options>C/C++ Compiler>Code>Enable transformations>Function inlining

--no_tbaa

Syntax `--no_tbaa`

Description Use this option to disable type-based alias analysis. When this options is not used, the compiler is free to assume that objects are only accessed through the declared type or through `unsigned char`.

See also *Type-based alias analysis*, page 102.



Project>Options>C/C++ Compiler>Code>Enable transformations>Type-based alias analysis

--no_typedefs_in_diagnostics

Syntax `--no_typedefs_in_diagnostics`

Description Use this option to disable the use of typedef names in diagnostics. Normally, when a type is mentioned in a message from the compiler, most commonly in a diagnostic message of some kind, the typedef names that were used in the original declaration are used whenever they make the resulting text shorter.

Example

```
typedef int (*MyPtr)(char const *);
MyPtr p = "foo";
```

will give an error message like the following:

```
Error[Pe144]: a value of type "char *" cannot be used to
initialize an entity of type "MyPtr"
```

If the `--no_typedefs_in_diagnostics` option is used, the error message will be like this:

```
Error[Pe144]: a value of type "char *" cannot be used to
initialize an entity of type "int (*)(char const *)"
```



To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--no_unroll

Syntax

```
--no_unroll
```

Description

Use this option to disable loop unrolling. The code body of a small loop, whose number of iterations can be determined at compile time, is duplicated to reduce the loop overhead.

For small loops, the overhead required to perform the looping can be large compared with the work performed in the loop body.

The loop unrolling optimization duplicates the body several times, reducing the loop overhead. The unrolled body also opens up for other optimization opportunities.

This optimization, which is performed at optimization level 9, normally reduces execution time, but increases code size. The resulting code may also be difficult to debug.

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

Note: This option has no effect at optimization levels below 9.



Project>Options>C/C++ Compiler>Code>Enable transformations>Loop unrolling

--no_warnings

Syntax `--no_warnings`

Description By default, the compiler issues warning messages. Use this option to disable all warning messages.



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

--no_wrap_diagnostics

Syntax `--no_wrap_diagnostics`

Description By default, long lines in compiler diagnostic messages are broken into several lines to make the message easier to read. Use this option to disable line wrapping of diagnostic messages.



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

-o

Syntax `-o {filename|directory}`

Parameters For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 122.

Description By default, the object code output produced by the compiler is located in a file with the same name as the source file, but with the extension `.r43`. Use this option to explicitly specify a different output filename for the object code output.



Project>Options>General Options>Output>Output directories>Object files

--omit_types

Syntax `--omit_types`

Description By default, the compiler includes type information about variables and functions in the object output. Use this option if you do not want the compiler to include this type information in the output, which is useful when you build a library that should not contain type information. The object file will then only contain type information that is a part of a symbol's name. This means that the linker cannot check symbol references for type correctness.



To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--only_stdout

Syntax `--only_stdout`

Description Use this option to make the compiler use the standard output stream (`stdout`) also for messages that are normally directed to the error output stream (`stderr`).



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

--preinclude

Syntax `--preinclude includefile`

Parameters For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 122.

Description Use this option to make the compiler include the specified include file before it starts to read the source file. This is useful if you want to change something in the source code for the entire application, for instance if you want to define a new symbol.



Project>Options>C/C++ Compiler>Preprocessor>Preinclude file

--preprocess

Syntax `--preprocess[=[c][n][l]] {filename|directory}`

Parameters

<code>c</code>	Preserve comments
<code>n</code>	Preprocess only
<code>l</code>	Generate #line directives

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 122.

Description Use this option to generate preprocessed output to a named file.



Project>Options>C/C++ Compiler>Preprocessor>Preprocessor output to file

--public_equ

Syntax `--public_equ symbol[=value]`

Parameters

<i>symbol</i>	The name of the assembler symbol to be defined
<i>value</i>	An optional value of the defined assembler symbol

Description This option is equivalent to defining a label in assembler language using the EQU directive and exporting it using the PUBLIC directive. This option may be used more than once on the command line.



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

--pic

Syntax `--pic`

Description

Use this option to make the compiler produce position-independent code, which means that functions can be relocated in runtime. This option is only available when compiling for the standard MSP430 architecture.



Project>Options>General Options>Target>Position-independent code

-r, --debug

Syntax

`-r`
`--debug`

Description

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

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



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

--reduce_stack_usage

Syntax

`--reduce_stack_usage`

Description

Use this option to make the compiler minimize the use of stack space at the cost of somewhat larger and slower code.



Project>Options>C/C++ Compiler>Code>Reduce stack usage

--regvar_r4

Syntax

`--regvar_R4`

Description

Use this option to reserve the register `R4` for use by global register variables, declared with the `__regvar` attribute. This can give more efficient code if used on frequently used variables.

See also

`--lock_r4`, page 136 and `__regvar`, page 177.



Project>Options>C/C++ Compiler>Code>R4 utilization>__regvar variables

--regvar_r5

Syntax

`--regvar_R5`

Description

Use this option to reserve the register `R5` for use by global register variables, declared with the `__regvar` attribute. This can give more efficient code if used on frequently used variables.

See also `--lock_r5`, page 136 and `__regvar`, page 177.



Project>Options>C/C++ Compiler>Code>R5 utilization>__regvar variables

--remarks

Syntax `--remarks`

Description The least severe diagnostic messages are called remarks. A remark indicates a source code construct that may cause strange behavior in the generated code. By default, the compiler does not generate remarks. Use this option to make the compiler generate remarks.

See also *Severity levels*, page 119.



Project>Options>C/C++ Compiler>Diagnostics>Enable remarks

--require_prototypes

Syntax `--require_prototypes`

Description Use this option to force the compiler to verify that all functions have proper prototypes. Using this option means that code containing any of the following will generate an error:

- A function call of a function with no declaration, or with a Kernighan & Ritchie C declaration
- A function definition of a public function with no previous prototype declaration
- An indirect function call through a function pointer with a type that does not include a prototype.



Project>Options>C/C++ Compiler>Language>Require prototypes

-s

Syntax `-s [2 | 3 | 6 | 9]`

Parameters

2	None* (Best debug support)
3 (default)	Low*
6	Medium

9

High (Maximum optimization)

***The most important difference between `-s2` and `-s3` is that at level 2, all non-static variables will live during their entire scope.**

Description

Use this option to make the compiler optimize the code for maximum execution speed. If no optimization option is specified, the optimization level 3 is used by default.

A low level of optimization makes it relatively easy to follow the program flow in the debugger, and, conversely, a high level of optimization makes it relatively hard.

Note: The `-s` and `-z` options cannot be used at the same time.



Project>Options>C/C++ Compiler>Code>Speed

--save_reg20

Syntax

`--save_reg20`

Description

Use this option to make all interrupt functions be treated as a `__save_reg20` declared function without explicitly using the `__save_reg20` keyword.

This is necessary if your application requires that all 20 bits of registers are preserved. The drawback is that the code will be somewhat slower.

Note: This option has no effect when compiling for the MSP430 architecture.

See also

`__save_reg20`, page 178



Project>Options>C/C++ Compiler>Code>20-bit context save on interrupt

--silent

Syntax

`--silent`

Description

By default, the compiler issues introductory messages and a final statistics report. Use this option to make the compiler operate without sending these messages to the standard output stream (normally the screen).

This option does not affect the display of error and warning messages.



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

--strict_ansi

Syntax `--strict_ansi`

Description By default, the compiler accepts a relaxed superset of ISO/ANSI C, see *Minor language extensions*, page 166. Use this option to ensure that the program conforms to the ISO/ANSI C standard.

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



Project>Options>C/C++ Compiler>Language>Language conformances>Strict ISO/ANSI

--warnings_affect_exit_code

Syntax `--warnings_affect_exit_code`

Description By default, the exit code is not affected by warnings, because only errors produce a non-zero exit code. With this option, warnings will generate a non-zero exit code.



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

--warnings_are_errors

Syntax `--warnings_are_errors`

Description Use this option to make the compiler treat all warnings as errors. If the compiler encounters an error, no object code is generated. Warnings that have been changed into remarks are not treated as errors.

Note: Any diagnostic messages that have been reclassified as warnings by the compiler option `--diag_warning` or the `#pragma diag_warning` directive will also be treated as errors when `--warnings_are_errors` is used.

See also `--diag_warning`, page 130 and `diag_warning`, page 186.



Project>Options>C/C++ Compiler>Diagnostics>Treat all warnings as errors

-Z

Syntax `-z [2 | 3 | 6 | 9]`

Parameters

2	None* (Best debug support)
3 (default)	Low*
6	Medium
9	High (Maximum optimization)

***The most important difference between `-z2` and `-z3` is that at level 2, all non-static variables will live during their entire scope.**

Description

Use this option to make the compiler optimize the code for minimum size. If no optimization option is specified, the optimization level 3 is used by default.

A low level of optimization makes it relatively easy to follow the program flow in the debugger, and, conversely, a high level of optimization makes it relatively hard.

Note: The `-s` and `-z` options cannot be used at the same time.



Project>Options>C/C++ Compiler>Code>Size

Data representation

This chapter describes the data types, pointers, and structure types supported by the MSP430 IAR C/C++ Compiler.

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

Alignment

Every C data object has an alignment that controls how the object can be stored in memory. Should an object have an alignment of, for example, 4, it must be stored on an address that is divisible by 4.

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

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

Another processor might only be able to read 2 bytes at a time; in that environment, the alignment for a 4-byte `long` integer might be 2.

A structure type will inherit the alignment from its components.

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

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

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

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

ALIGNMENT IN THE MSP430 IAR C/C++ COMPILER

The MSP430 microcontroller can access memory using 8- or 16-bit operations. However, when a 16-bit access is used, the data must be located at an even address. The MSP430 IAR C/C++ Compiler ensures this by assigning an alignment to every data type, ensuring that the MSP430 microcontroller will be able to read the data.

Basic data types

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

INTEGER TYPES

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

Data type	Size	Range	Alignment
bool	8 bits	0 to 1	1
char	8 bits	0 to 255	1
signed char	8 bits	-128 to 127	1
unsigned char	8 bits	0 to 255	1
signed short	16 bits	-32768 to 32767	2
unsigned short	16 bits	0 to 65535	2
signed int	16 bits	-32768 to 32767	2
unsigned int	16 bits	0 to 65535	2
signed long	32 bits	-2^{31} to $2^{31}-1$	2
unsigned long	32 bits	0 to $2^{32}-1$	2
signed long long	64 bits	-2^{63} to $2^{63}-1$	4
unsigned long long	64 bits	0 to $2^{64}-1$	4

Table 26: Integer types

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

Bool

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

The long long type

The `long long` data type is supported with the following restrictions:

- The CLIB runtime library does not support the `long long` type
- A `long long` variable cannot be used in a switch statement.

The enum type

The compiler will use the smallest type required to hold `enum` constants, preferring `signed` in front of `unsigned`.

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

To make the compiler use a larger type than it would automatically use, define an `enum` constant with a large enough value. For example,

```
/* Disables usage of the char type for enum */
enum Cards{Spade1, Spade2,
           DontUseChar=257};
```

The char type

The `char` type is by default `unsigned` in the compiler, but the `--char_is_signed` compiler option allows you to make it `signed`. Note, however, that the library is compiled with the `char` type as `unsigned`.

The wchar_t type

The `wchar_t` data type is an integer type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locals.

The `wchar_t` data type is supported by default in the C++ language. To use the `wchar_t` type also in C source code, you must include the file `stddef.h` from the runtime library.

Note: The IAR CLIB Library has only rudimentary support for `wchar_t`.

Bitfields

In ISO/ANSI C, `int` and `unsigned int` can be used as the base type for integer bitfields. In the MSP430 IAR C/C++ Compiler, any integer type can be used as the base type when language extensions are enabled.

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

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

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

FLOATING-POINT TYPES

In the MSP430 IAR C/C++ Compiler, floating-point values are represented in standard IEEE 754 format. The sizes for the different floating-point types are:

Type	Size if <code>--double=32</code>	Size if <code>--double=64</code>
float	32 bits	32 bits
double	32 bits (default)	64 bits
long double	32 bits	64 bits

Table 27: Floating-point types

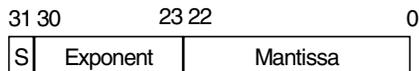
Note: The size of `double` and `long double` depends on the `--double={32|64}` option, see `--double`, page 131. The type `long double` uses the same precision as `double`.

The compiler does not support subnormal numbers. All operations that should produce subnormal numbers will instead generate zero.

Exception flags according to the IEEE 754 standard are not supported.

32-bit floating-point format

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



The exponent is 8 bits, and the mantissa is 23 bits.

The value of the number is:

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

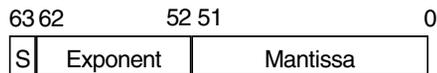
The range of the number is:

$$\pm 1.18\text{E}-38 \text{ to } \pm 3.39\text{E}+38$$

The precision of the float operators (+, -, *, and /) is approximately 7 decimal digits.

64-bit floating-point format

The representation of a 64-bit floating-point number as an integer is:



The exponent is 11 bits, and the mantissa is 52 bits.

The value of the number is:

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

The range of the number is:

$$\pm 2.23\text{E}-308 \text{ to } \pm 1.79\text{E}+308$$

The precision of the float operators (+, -, *, and /) is approximately 15 decimal digits.

Representation of special floating-point numbers

The following list describes the representation of special floating-point numbers:

- Zero is represented by zero mantissa and exponent. The sign bit signifies positive or negative zero.
- Infinity is represented by setting the exponent to the highest value and the mantissa to zero. The sign bit signifies positive or negative infinity.
- Not a number (NaN) is represented by setting the exponent to the highest positive value and the most significant bit of the mantissa to a non-zero value. The value of the sign bit is ignored.

Note: The IAR CLIB Library does not fully support the special cases of floating-point numbers, such as infinity, and NaN. A library function which gets one of these special cases of floating-point numbers as an argument may behave unexpectedly.

Pointer types

The MSP430 IAR C/C++ Compiler has two basic types of pointers: function pointers and data pointers.

For the MSP430 architecture, the size of code and data pointers is always 16 bits and they can address the entire memory.

For the MSP430X architecture, the size of:

- Data pointers is always 16 bits and they can address 64 Kbytes of memory
- Code pointers is always 32 bits and they can address the entire 1 Mbyte of memory.

Data pointers use `int` as index type for both the MSP430 and the MSP430X architectures.

CASTING

Casts between pointers have the following characteristics:

- Casting a *value* of an integer type to a pointer of a smaller type is performed by truncation
- Casting a *value* of an integer type to a pointer of a larger type is performed by zero extension
- Casting a *pointer type* to a smaller integer type is performed by truncation
- Casting a *pointer type* to a larger integer type is performed 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 MSP430 IAR C/C++ Compiler, 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 MSP430 IAR C/C++ Compiler, 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];           /* 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 MSP430 IAR C/C++ Compiler, 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

The `struct` and `union` types inherit the alignment requirements of their members. In addition, the size of a `struct` is adjusted to allow arrays of aligned structure objects.

GENERAL LAYOUT

Members of a `struct` (fields) are always allocated in the order given in the declaration. The members are placed in memory according to the given alignment (offsets).

Example

```
struct {
    short s; /* stored in byte 0 and 1 */
    char c; /* stored in byte 2 */
    long l; /* stored in byte 4, 5, 6, and 7 */
    char c2; /* stored in byte 8 */
} s;
```

The following diagram shows the layout in memory:

s.s 2 bytes	s.c 1 byte	pad 1 byte	s.l 4 bytes	s.c2 1 byte	pad 1 byte
----------------	---------------	---------------	----------------	----------------	---------------

The alignment of the structure is 2 bytes, and its size is 10 bytes.

PACKED STRUCTURE TYPES

The `#pragma pack` directive is used for changing the alignment requirements of the members of a structure. This will change the way the layout of the structure is performed. The members will be placed in the same order as when declared, but there might be less pad space between members.

Example

```
#pragma pack(1)
struct {
    short s;
    char c;
    long l;
    char c2;
} s;
```

will be placed:



For more information, see *Rearranging elements in a structure*, page 104.

Type qualifiers

According to the ISO/ANSI C standard, `volatile` and `const` are type qualifiers.

DECLARING OBJECTS VOLATILE

There are three main reasons for declaring an object `volatile`:

- Shared access; the object is shared between several tasks in a multitasking environment
- Trigger access; as for a memory-mapped SFR where the fact that an access occurs has an effect
- Modified access; where the contents of the object can change in ways not known to the compiler.

Definition of access to volatile objects

The ISO/ANSI standard defines an abstract machine, which governs the behavior of accesses to `volatile` declared objects. In general and in accordance to the abstract machine, the compiler:

- Considers each read and write access to an object that has been declared `volatile` as an access
- The unit for the access is either the entire object or, for accesses to an element in a composite object—such as an array, struct, class, or union—the element. For example:

```
char volatile a;
a = 5; /* A write access */
a += 6; /* First a read then a write access */
```

- An access to a bitfield is treated as an access to the underlying type.

However, these rules are not detailed enough to handle the hardware-related requirements. The rules specific to the MSP430 IAR C/C++ Compiler are described below.

Rules for accesses

In the MSP430 IAR C/C++ Compiler, accesses to `volatile` declared objects are subject to the following rules:

- All accesses are preserved
- All accesses are complete, that is, the whole object is accessed
- All accesses are performed in the same order as given in the abstract machine
- All accesses are atomic, that is, non-interruptible.

The MSP430 IAR C/C++ Compiler adheres to these rules for all 8-bit and 16-bit memory accesses.

DECLARING OBJECTS CONST

The `const` type qualifier is used for indicating that a data object, accessed directly or via a pointer, is non-writable. A pointer to `const` declared data can point to both constant and non-constant objects. It is good programming practice to use `const` declared pointers whenever possible because this improves the compiler's possibilities to optimize the generated code and reduces the risk of application failure due to erroneously modified data.

Static and global objects declared `const` are allocated in ROM.

In C++, objects that require runtime initialization cannot be placed in ROM.

Data types in C++

In C++, all plain C data types are represented in the same way as described earlier in this chapter. However, if any Embedded C++ features are used for a type, no assumptions can be made concerning the data representation. This means, for example, that it is not legal to write assembler code that accesses class members.

Compiler extensions

This chapter gives a brief overview of the MSP430 IAR C/C++ Compiler extensions to the ISO/ANSI C standard. All extensions can also be used for the C++ programming language. More specifically the chapter describes the available C language extensions.

Compiler extensions overview

The compiler offers the standard features of ISO/ANSI C as well as a wide set of extensions, ranging from features specifically tailored for efficient programming in the embedded industry to the relaxation of some minor standards issues.

You can find the extensions available as:

- C/C++ language extensions

For a summary of available language extensions, see *C language extensions*, page 160. For reference information about the extended keywords, see the chapter *Extended keywords*. For information about C++, the two levels of support for the language, and C++ language extensions; see the chapter *Using C++*.

- Pragma directives

The `#pragma` directive is defined by the ISO/ANSI C standard and is a mechanism for using vendor-specific extensions in a controlled way to make sure that the source code is still portable.

The compiler provides a set of predefined pragma directives, which can be used for controlling the behavior of the compiler, for example how it allocates memory, whether it allows extended keywords, and whether it outputs warning messages. Most pragma directives are preprocessed, which means that macros are substituted in a pragma directive. The pragma directives are always enabled in the compiler and many of them have an equivalent C/C++ language extension, such as an extended keyword. For a list of available pragma directives, see the chapter *Pragma directives*.

- Preprocessor extensions

The preprocessor of the compiler adheres to the ISO/ANSI standard. In addition, the compiler also makes a number of preprocessor-related extensions available to you. For more information, see the chapter *The preprocessor*.

- Intrinsic functions

The intrinsic functions provide direct access to low-level processor operations and can be very useful in, for example, time-critical routines. The intrinsic functions compile into inline code, either as a single instruction or as a short sequence of instructions. To read more about using intrinsic functions, see *Mixing C and assembler*, page 77. For a list of available functions, see the chapter *Intrinsic functions*.

- Library functions

The IAR DLIB Library provides most of the important C and C++ library definitions that apply to embedded systems. In addition, the library also provides some extensions, partly taken from the C99 standard. For more information, see *IAR DLIB Library*, page 214.

Note: Any use of these extensions, except for the pragma directives, makes your application inconsistent with the ISO/ANSI C standard.

ENABLING LANGUAGE EXTENSIONS



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

For information about how to enable and disable language extensions from the command line, see the compiler options *-e*, page 132 and *--strict_ansi*, page 147.

C language extensions

This section gives a brief overview of the C language extensions available in the MSP430 IAR C/C++ Compiler. The compiler provides a wide set of extensions, so to help you to find the extensions required by your application, the extensions have been grouped according to their expected usefulness. In short, this means:

- Important language extensions—extensions specifically tailored for efficient embedded programming, typically to meet memory restrictions
- Useful language extensions—features considered useful and typically taken from related standards, such as C99 and C++
- Minor language extensions, that is, the relaxation of some minor standards issues and also some useful but minor syntax extensions.

IMPORTANT LANGUAGE EXTENSIONS

The following language extensions available both in the C and the C++ programming languages are well suited for embedded systems programming:

- Memory attributes, type attributes, and object attributes

For information about the related concepts, the general syntax rules, and for reference information, see the chapter *Extended keywords*.

- Placement at an absolute address or in a named segment

The @ operator or the directive #pragma location can be used for placing global and static variables at absolute addresses, or placing a variable or function in a named segment. For more information about using these primitives, see *Controlling data and function placement in memory*, page 33 and *location*, page 188.

- Alignment

Each data type has its own alignment, for more details, see *Alignment*, page 149. If you want to change alignment, the #pragma pack and #pragma data_alignment are available. If you want to check the alignment of an object, use the __ALIGNOF__ () operator.

The __ALIGNOF__ operator is used for accessing the alignment of an object. It takes one of two forms:

- __ALIGNOF__ (type)
- __ALIGNOF__ (expression)

In the second form, the expression is not evaluated.

- Anonymous structs and unions

C++ includes a feature named anonymous unions. The compiler allows a similar feature for both structs and unions in the C programming language. For more information, see *Anonymous structs and unions*, page 105.

- Bitfields and non-standard types

In ISO/ANSI C, a bitfield must be of the type int or unsigned int. Using IAR Systems language extensions, any integer types and enums may be used. The advantage is that the struct will be smaller. This matches G.5.8 in the appendix of the ISO standard, *ISO Portability Issues*. For more information, see *Bitfields*, page 151.

- Dedicated segment operators __segment_begin and __segment_end

The syntax for these operators are:

```
void * __segment_begin(segment)
void * __segment_end(segment)
```

These operators return the address of the first byte of the named *segment* and the first byte *after* the named *segment*, respectively. This can be useful if you have used the @ operator or the #pragma location directive to place a data object or a function in a user-defined segment.

The named *segment* must be a string literal that has been declared earlier with the #pragma segment directive. The type of the __segment_begin function is a pointer to void. Note that you must have enabled language extensions to use these operators.

In the following example, the type of the __segment_begin intrinsic function is void *.

```
#pragma segment="MYSEG"
...
segment_start_address = __segment_begin("MYSEG");
```

See also *segment*, page 194 and *location*, page 188.

USEFUL LANGUAGE EXTENSIONS

This section lists and briefly describes useful extensions, that is, useful features typically taken from related standards, such as C99 and C++:

- Inline functions

The #pragma inline directive, alternatively the inline keyword, advises the compiler that the function whose declaration follows immediately after the directive should be inlined. This is similar to the C++ keyword inline. For more information, see *inline*, page 187.

- Mixing declarations and statements

It is possible to mix declarations and statements within the same scope. This feature is part of the C99 standard and C++.

- Declaration in for loops

It is possible to have a declaration in the initialization expression of a for loop, for example:

```
for (int i = 0; i < 10: ++i)
{ ... }
```

This feature is part of the C99 standard and C++.

- The bool data type

To use the bool type in C source code, you must include the file stdbool.h. This feature is part of the C99 standard and C++. (The bool data type is supported by default in C++.)

- C++ style comments

C++ style comments are accepted. A C++ style comment starts with the character sequence `//` and continues to the end of the line. For example:

```
// The length of the bar, in centimeters.
int length;
```

This feature is copied from the C99 standard and C++.

Inline assembler

Inline assembler can be used for inserting assembler instructions in the generated function. This feature is part of the C99 standard and C++.

The `asm` and `__asm` extended keywords both insert an assembler instruction. However, when compiling C source code, the `asm` keyword is not available when the option `--strict_ansi` is used. The `__asm` keyword is always available.

Note: Not all assembler directives or operators can be inserted using this keyword.

The syntax is:

```
asm ("string");
```

The string can be a valid assembler instruction or a data definition assembler directive, but not a comment. You can write several consecutive inline assembler instructions, for example:

```
asm ("Label:      nop\n"
     "             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 77.

Compound literals

To create compound literals you can use the following syntax:

```
/* Create a pointer to an anonymous array */
int *p = (int []) {1,2,3};

/* Create a pointer to an anonymous structX */
structX *px = &(amp;structX) {5,6,7};
```

Note:

- A compound literal can be modified unless it is declared `const`
- Compound literals are not supported in Embedded C++ and Extended EC++.
- This feature is part of the C99 standard.

Incomplete arrays at end of structs

The last element of a `struct` may be an incomplete array. This is useful because one chunk of memory can be allocated for the `struct` itself and for the array, regardless of the size of the array.

Note: The array cannot be the only member of the `struct`. If that was the case, then the size of the `struct` would be zero, which is not allowed in ISO/ANSI C.

Example

```
struct str
{
    char a;
    unsigned long b[];
};

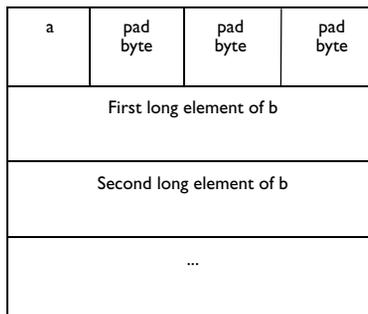
struct str * GetAStr(int size)
{
    return malloc(sizeof(struct str) +
                 sizeof(unsigned long) * size);
}

void UseStr(struct str * s)
{
    s->b[10] = 0;
}
```

The `struct` will inherit the alignment requirements from all elements, including the alignment of the incomplete array. The array itself will not be included in the size of the `struct`. However, the alignment requirements will ensure that the `struct` will end exactly at the beginning of the array; this is known as padding.

In the example, the alignment of `struct str` will be 4 and the size is also 4. (Assuming a processor where the alignment of `unsigned long` is 4.)

The memory layout of `struct str` is described in the following figure.



This feature is copied from the C99 standard.

Hexadecimal floating-point constants

Floating-point constants can be given in hexadecimal style. The syntax is $0xMANTp\{+|- \}EXP$, where *MANT* is the mantissa in hexadecimal digits, including an optional `.` (decimal point), and *EXP* is the exponent with decimal digits, representing an exponent of 2. This feature is copied from the C99 standard.

Examples

`0x1p0` is 1

`0xA.8p2` is $10.5 * 2^2$

Designated initializers in structures and arrays

Any initialization of either a structure (`struct` or `union`) or array can have a designation. A designation consists of one or more designators followed by an initializer. A designator for a structure is specified as `.elementname` and for an array [`constant index expression`]. Using designated initializers is not supported in C++.

Examples

The following definition shows a `struct` and its initialization using designators:

```
struct{
    int i;
    int j;
    int k;
    int l;
```

```

short array[10]
} x = {
    .l.j = 6,      /* initialize l and j to 6 */
    8,            /* initialize k to 8 */
    {[7][3] = 2,  /* initialize element 7 and 3 to 2 */
    5}           /* initialize element 4 to 5 */
    .k = 4       /* reinitialize k to 4 */
};

```

Note that a designator specifies the destination element of the initialization. Note also that if one element is initialized more than once, it is the last initialization that will be used.

To initialize an element in a union other than the first, do like this:

```

union{
    int i;
    float f;
}y = {.f = 5.0};

```

To set the size of an array by initializing the last element, do like this:

```
char array[] = {[10] = 'a'};
```

MINOR LANGUAGE EXTENSIONS

This section lists and briefly describes minor extensions, that is, the relaxation of some standards issues and also some useful but minor syntax extensions:

- Arrays of incomplete types

An array may have an incomplete `struct`, `union`, or `enum` type as its element type. The types must be completed before the array is used (if it is), or by the end of the compilation unit (if it is not).

- Forward declaration of `enum` types

The IAR Systems language extensions allow that you first declare the name of an `enum` and later resolve it by specifying the brace-enclosed list.

- Missing semicolon at end of `struct` or `union` specifier

A warning is issued if the semicolon at the end of a `struct` or `union` specifier is missing.

- Null and `void`

In operations on pointers, a pointer to `void` is always implicitly converted to another type if necessary, and a null pointer constant is always implicitly converted to a null pointer of the right type if necessary. In ISO/ANSI C, some operators allow such things, while others do not allow them.

- Casting pointers to integers in static initializers

In an initializer, a pointer constant value may be cast to an integral type if the integral type is large enough to contain it. For more information about casting pointers, see *Casting*, page 154.

- Taking the address of a register variable

In ISO/ANSI C, it is illegal to take the address of a variable specified as a register variable. The compiler allows this, but a warning is issued.

- Duplicated size and sign specifiers

Should the size or sign specifiers be duplicated (for example, `short short` or `unsigned unsigned`), an error is issued.

- `long float` means `double`

The type `long float` is accepted as a synonym for `double`.

- Repeated `typedef` declarations

Redeclarations of `typedef` that occur in the same scope are allowed, but a warning is issued.

- Mixing pointer types

Assignment and pointer difference is allowed between pointers to types that are interchangeable but not identical; for example, `unsigned char *` and `char *`. This includes pointers to integral types of the same size. A warning is issued.

Assignment of a string constant to a pointer to any kind of character is allowed, and no warning will be issued.

- Non-top level `const`

Assignment of pointers is allowed in cases where the destination type has added type qualifiers that are not at the top level (for example, `int **` to `int const **`). Comparing and taking the difference of such pointers is also allowed.

- Non-lvalue arrays

A non-lvalue array expression is converted to a pointer to the first element of the array when it is used.

- Empty translation units

A translation unit (source file) is allowed to be empty, that is, it does not have to contain any declarations.

In strict ISO/ANSI mode, a warning is issued if the translation unit is empty.

- Comments at the end of preprocessor directives

This extension, which makes it legal to place text after preprocessor directives, is enabled, unless strict ISO/ANSI mode is used. The purpose of this language extension is to support compilation of legacy code; we do *not* recommend that you write new code in this fashion.

- An extra comma at the end of `enum` lists

Placing an extra comma is allowed at the end of an `enum` list. In strict ISO/ANSI mode, a warning is issued.

- A label preceding a `}`

In ISO/ANSI C, a label must be followed by at least one statement. Therefore, it is illegal to place the label at the end of a block. In the MSP430 IAR C/C++ Compiler, a warning is issued.

Note: This also applies to the labels of `switch` statements.

- Empty declarations

An empty declaration (a semicolon by itself) is allowed, but a remark is issued (provided that remarks are enabled).

- Single-value initialization

ISO/ANSI C requires that all initializer expressions of static arrays, `structs`, and `unions` are enclosed in braces.

Single-value initializers are allowed to appear without braces, but a warning is issued. In the MSP430 IAR C/C++ Compiler, the following expression is allowed:

```
struct str
{
    int a;
} x = 10;
```

- Declarations in other scopes

External and static declarations in other scopes are visible. In the following example, the variable `y` can be used at the end of the function, even though it should only be visible in the body of the `if` statement. A warning is issued.

```
int test(int x)
{
    if (x)
    {
        extern int y;
        y = 1;
    }
}
```

```
    return y;  
}
```

- Expanding function names into strings with the function as context

Use any of the symbols `__func__` or `__FUNCTION__` inside a function body to make it expand into a string, with the function name as context. Use the symbol `__PRETTY_FUNCTION__` to also include the parameter types and return type. The result might, for example, look like this if you use the `__PRETTY_FUNCTION__` symbol:

```
"void func(char)"
```

These symbols are useful for assertions and other trace utilities and they require that language extensions are enabled, see `-e`, page 132.

Extended keywords

This chapter describes the extended keywords that support specific features of the MSP430 microcontroller and the general syntax rules for the keywords. Finally the chapter gives a detailed description of each keyword.

General syntax rules for extended keywords

To understand the syntax rules for the extended keywords, it is important to be familiar with some related concepts.

The MSP430 IAR C/C++ Compiler provides a set of attributes that can be used on functions or data objects to support specific features of the MSP430 microcontroller. There are two types of attributes—*type attributes* and *object attributes*:

- Type attributes affect the *external functionality* of the data object or function
- Object attributes affect the *internal functionality* of the data object or function.

The syntax for the keywords differs slightly depending on whether it is a type attribute or an object attribute, and whether it is applied to a data object or a function.

For detailed information about each attribute, see *Descriptions of extended keywords*, page 175.

Note: The extended keywords are only available when language extensions are enabled in the MSP430 IAR C/C++ Compiler.



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

Use the `-e` compiler option to enable language extensions. See `-e`, page 132 for additional information.

TYPE ATTRIBUTES

Type attributes define how a function is called, or how a data object is accessed. This means that type attributes must be specified both when they are defined and in the declaration.

You can either place the type attributes directly in your source code, or use the pragma directive `#pragma type_attribute`.

Type attributes can be further divided into *memory attributes* and *general type attributes*.

Memory attributes

A memory attribute corresponds to a certain logical or physical memory in the microcontroller.

Available *data memory attributes*: `__data16` and `__regvar`

Data objects, functions, and destinations of pointers or C++ references always have a memory attribute. If no attribute is explicitly specified in the declaration or by the pragma directive `#pragma type_attribute`, an appropriate default attribute is used. You can only specify one memory attribute for each level of pointer indirection.

General type attributes

The following general type attributes are available:

- *Function type attributes* change the calling convention of a function:
`__interrupt`, `__monitor`, and `__task`
- *Data type attributes*: `const` and `volatile`

You can specify as many type attributes as required for each level of pointer indirection.

To read more about `volatile`, see *Declaring objects volatile*, page 156.

Syntax for type attributes used on data objects

In general, type attributes for data objects follow the same syntax as the type qualifiers `const` and `volatile`.

The following declaration assigns the `__data16` type attribute to the variables `i` and `j`; in other words, the variable `i` and `j` is placed in `data16` memory. The variables `k` and `l` behave in the same way:

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

Note that the attribute affects both identifiers.

The following declaration of `i` and `j` is equivalent with the previous one:

```
#pragma type_attribute=__data16
int i, j;
```

The advantage of using pragma directives for specifying keywords is that it offers you a method to make sure that the source code is portable. Note that the pragma directive has no effect if a memory attribute is already explicitly declared.

An easier way of specifying storage is to use type definitions. The following two declarations are equivalent:

```
typedef char __data16 Byte;
```

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

and

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

Note that `#pragma type_attribute` can be used together with a `typedef` declaration.

Syntax for type attributes on functions

The syntax for using type attributes on functions, differs slightly from the syntax of type attributes on data objects. For functions, the attribute must be placed either in front of the return type, alternatively in parentheses, for example:

```
__interrupt void my_handler(void);
```

or

```
void (__interrupt my_handler)(void);
```

The following declaration of `my_handler` is equivalent with the previous one:

```
#pragma type_attribute=__interrupt
void my_handler(void);
```

OBJECT ATTRIBUTES

Object attributes affect the internal functionality of functions and data objects, but not how the function is called or how the data is accessed. This means that an object attribute does not need to be present in the declaration of an object.

The following object attributes are available:

- Object attributes that can be used for variables: `__no_init`
- Object attributes that can be used for functions and variables: `location`, `@`, and `__root`
- Object attributes that can be used for functions: `__intrinsic`, `__noreturn`, `__raw`, `__save_reg20`, and `vector`.

You can specify as many object attributes as required for a specific function or data object.

For more information about `location` and `@`, see *Controlling data and function placement in memory*, page 33. For more information about `vector`, see *vector*, page 195.

Syntax for object attributes

The object attribute must be placed in front of the type. For example, to place `myarray` in memory that is not initialized at startup:

```
__no_init int myarray[10];
```

The `#pragma object_attribute` directive can also be used. The following declaration is equivalent to the previous one:

```
#pragma object_attribute=__no_init
int myarray[10];
```

Note: Object attributes cannot be used in combination with the `typedef` keyword.

Summary of extended keywords

The following table summarizes the extended keywords:

Extended keyword	Description
<code>__data16</code>	Controls the storage of data objects
<code>__interrupt</code>	Supports interrupt functions
<code>__intrinsic</code>	Reserved for compiler internal use only
<code>__monitor</code>	Supports atomic execution of a function
<code>__no_init</code>	Supports non-volatile memory
<code>__noreturn</code>	Informs the compiler that the declared function will not return
<code>__raw</code>	Prevents saving used registers in interrupt functions
<code>__regvar</code>	Permanently places a variable in a specified register
<code>__root</code>	Ensures that a function or variable is included in the object code even if unused
<code>__save_reg20</code>	Saves and restores all 20 bits in 20-bit registers
<code>__task</code>	Allows functions to exit without restoring registers

Table 28: Extended keywords summary

Descriptions of extended keywords

The following sections give detailed information about each extended keyword.

`__data16`

Syntax	Follows the generic syntax rules for memory type attributes that can be used on data objects, see <i>Type attributes</i> , page 171.
Description	<p>The <code>__data16</code> memory attribute explicitly places individual variables and constants in data16 memory, which is the entire 64 Kbytes of memory in the MSP430 architecture and the lower 64 Kbytes in the MSP430X architecture. You can also use the <code>__data16</code> attribute to create a pointer explicitly pointing to an object located in the data16 memory.</p> <p>Note: The <code>__data16</code> attribute is only available for possible future needs as this attribute can be used for placing objects anywhere in the entire memory and because there is no need for another attribute for placing an object within a separate range of the memory.</p>
Storage information	<ul style="list-style-type: none"> ● Address range: 0–0xFFFF (64 Kbytes) ● Maximum object size: 65535 bytes ● Pointer size: 2 bytes
Example	<code>__data16 int x;</code>

`__interrupt`

Syntax	Follows the generic syntax rules for type attributes that can be used on functions, see <i>Type attributes</i> , page 171.
Description	<p>The <code>__interrupt</code> keyword specifies interrupt functions. To specify one or several interrupt vectors, use the <code>#pragma vector</code> directive. The range of the interrupt vectors depends on the device used. It is possible to define an interrupt function without a vector, but then the compiler will not generate an entry in the interrupt vector table.</p> <p>An interrupt function must have a <code>void</code> return type and cannot have any parameters.</p> <p>The header file <code>iodevice.h</code>, where <code>device</code> corresponds to the selected device, contains predefined names for the existing interrupt vectors.</p>
Example	<pre>#pragma vector=0x14 __interrupt void my_interrupt_handler(void);</pre>

See also *Interrupt functions*, page 15, *vector*, page 195, *INTVEC*, page 228.

__intrinsic

Description The `__intrinsic` keyword is reserved for compiler internal use only.

__monitor

Syntax Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 171.

Description The `__monitor` keyword causes interrupts to be disabled during execution of the function. This allows atomic operations to be performed, such as operations on semaphores that control access to resources by multiple processes. A function declared with the `__monitor` keyword is equivalent to any other function in all other respects.

Example

```
__monitor int get_lock(void);
```

See also *Monitor functions*, page 18. Read also about the intrinsic functions `__disable_interrupt`, page 202, `__enable_interrupt`, page 202, `__get_interrupt_state`, page 203, and `__set_interrupt_state`, page 205.

__no_init

Syntax Follows the generic syntax rules for object attributes, see *Object attributes*, page 173.

Description Use the `__no_init` keyword to place a data object in non-volatile memory. This means that the initialization of the variable, for example at system startup, is suppressed.

Example

```
__no_init int myarray[10];
```

__noreturn

Syntax Follows the generic syntax rules for object attributes, see *Object attributes*, page 173.

Description The `__noreturn` keyword can be used on a function to inform the compiler that the function will not return. If you use this keyword on such functions, the compiler can optimize more efficiently. Examples of functions that do not return are `abort` and `exit`.

Example

```
__noreturn void terminate(void);
```

__raw

Syntax	Follows the generic syntax rules for object attributes, see <i>Object attributes</i> , page 173.
Description	This keyword prevents saving used registers in interrupt functions. Interrupt functions preserve the contents of all used processor registers at function entrance and restore them at exit. However, for some very special applications, it can be desirable to prevent the registers from being saved at function entrance. This can be accomplished by the use of the keyword <code>__raw</code> .
Example	<pre>__raw __interrupt void my_interrupt_function()</pre>

__regvar

Syntax	Follows the generic syntax rules for type attributes, see <i>Type attributes</i> , page 171.
Description	This keyword is used for declaring that a <i>global</i> or <i>static</i> variable should be placed permanently in the specified register. The registers R4–R5 can be used for this purpose, provided that they have been reserved with one of the <code>--regvar_r4</code> or <code>--regvar_r5</code> compiler options. The <code>__regvar</code> attribute can be used on integer types, pointers, 32-bit floating-point numbers, structures with one element and unions of all these. However, it is <i>not</i> possible to point to an object that has been declared <code>__regvar</code> . An object declared <code>__regvar</code> cannot have an initial value. Note: If a module in your application has been compiled using <code>--regvar_r4</code> , it can only be linked with modules that have been compiled with either <code>--regvar_r4</code> or <code>--lock_r4</code> . The same is true for <code>--regvar_r5/--lock_r5</code> .
Example	To declare a global register variable, use the following syntax: <pre>__regvar __no_init type variable_name @ location</pre> where <i>location</i> is either <code>__R4</code> or <code>__R5</code> , declared in <code>intrinsics.h</code> . This will create a variable called <i>variable_name</i> of type <i>type</i> , located in register R4 or R5, for example: <pre>__regvar __no_init int counter @ __R4;</pre>
See also	<code>--regvar_r4</code> , page 144 and <code>--regvar_r5</code> , page 144

__root

Syntax	Follows the generic syntax rules for object attributes, see <i>Object attributes</i> , page 173.
Description	A function or variable with the <code>__root</code> attribute is kept whether or not it is referenced from the rest of the application, provided its module is included. Program modules are always included and library modules are only included if needed.
Example	<pre>__root int myarray[10];</pre>
See also	To read more about modules, segments, and the link process, see the <i>IAR Linker and Library Tools Reference Guide</i> .

__save_reg20

Syntax	Follows the generic syntax rules for object attributes, see <i>Object attributes</i> , page 173.
Description	<p>When compiling for the MSP430X architecture, use this keyword to save and restore all 20 bits of the registers that are used, instead of only 16 bits, which are saved and restored by normal functions. This keyword will make the function save all registers and not only the ones used by the function to guarantee that 20-bit registers are not destroyed by subsequent calls.</p> <p>This may be necessary if the function is called from assembler routines that use the upper 4 bits of the 20-bit registers.</p> <p>Note: The <code>__save_reg20</code> keyword has no effect when compiling for the MSP430 architecture.</p>
Example	<pre>__save_reg20 void myFunction(void);</pre>
See also	<i>Interrupt functions for the MSP430X architecture</i> , page 18

__task

Syntax	Follows the generic syntax rules for type attributes that can be used on functions, see <i>Type attributes</i> , page 171.
Description	This keyword allows functions to exit without restoring registers and it is typically used for the <code>main</code> 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 `__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`.

Example

```
__task void my_handler(void);
```


Pragma directives

This chapter describes the pragma directives of the MSP430 IAR C/C++ Compiler.

The `#pragma` directive is defined by the ISO/ANSI C standard and is a mechanism for using vendor-specific extensions in a controlled way to make sure that the source code is still portable.

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

The pragma directives are always enabled in the compiler.

Summary of pragma directives

The following table lists the pragma directives of the compiler that can be used either with the `#pragma` preprocessor directive or the `_Pragma()` preprocessor operator:

Pragma directive	Description
<code>bis_nmi_ie1</code>	Generates a BIS instruction just before the RETI instruction
<code>bitfields</code>	Controls the order of bitfield members
<code>constseg</code>	Places constant variables in a named segment
<code>data_alignment</code>	Gives a variable a higher (more strict) alignment
<code>dataseg</code>	Places variables in a named segment
<code>diag_default</code>	Changes the severity level of diagnostic messages
<code>diag_error</code>	Changes the severity level of diagnostic messages
<code>diag_remark</code>	Changes the severity level of diagnostic messages
<code>diag_suppress</code>	Suppresses diagnostic messages
<code>diag_warning</code>	Changes the severity level of diagnostic messages
<code>include_alias</code>	Specifies an alias for an include file
<code>inline</code>	Inlines a function
<code>language</code>	Controls the IAR Systems language extensions

Table 29: Pragma directives summary

Pragma directive	Description
location	Specifies the absolute address of a variable, or places groups of functions or variables in named segments
message	Prints a message
no_epilogue	Performs a local return sequence
object_attribute	Changes the definition of a variable or a function
optimize	Specifies the type and level of an optimization
pack	Specifies the alignment of structures and union members
required	Ensures that a symbol that is needed by another symbol is included in the linked output
rtmodel	Adds a runtime model attribute to the module
segment	Declares a segment name to be used by intrinsic functions
type_attribute	Changes the declaration and definitions of a variable or function
vector	Specifies the vector of an interrupt or trap function

Table 29: Pragma directives summary (Continued)

Note: For portability reasons, see also *Recognized pragma directives (6.8.6)*, page 235 and the *MSP430 IAR Embedded Workbench® Migration Guide*.

Descriptions of pragma directives

This section gives detailed information about each pragma directive.

bis_nmi_ie1

Syntax	<code>#pragma bis_nmi_ie1=mask</code>
Parameters	<p><i>mask</i> A constant expression</p>
Description	<p>Use this pragma directive for changing the interrupt control bits in the register IE, within an NMI service routine. A <code>BIS.W #mask, IE1</code> instruction is generated right before the <code>RETI</code> instruction at the end of the function, after any <code>POP</code> instructions.</p> <p>The effect is that NMI interrupts cannot occur until after the <code>BIS</code> instruction. The advantage of placing it at the end of the <code>POP</code> instructions is that less stack will be used in the case of nested interrupts.</p>

Example

In the following example, the `OFIE` bit will be set as the last instruction before the `RETI` instruction:

```
#pragma bis_nmi_ie1=OFIE
#pragma vector=NMI_VECTOR
__interrupt void myInterruptFunction(void)
{
  ...
}
```

bitfields

Syntax

```
#pragma bitfields={reversed|default}
```

Parameters

<code>reversed</code>	Bitfield members are placed from the most significant bit to the least significant bit.
<code>default</code>	Bitfield members are placed from the least significant bit to the most significant bit.

Description

Use this pragma directive to control the order of bitfield members.

By default, the MSP430 IAR C/C++ Compiler places bitfield members from the least significant bit to the most significant bit in the container type. Use the `#pragma bitfields=reversed` directive to place the bitfield members from the most significant to the least significant bit. This setting remains active until you turn it off again with the `#pragma bitfields=default` directive.

constseg

Syntax

```
#pragma constseg={SEGMENT_NAME|default}
```

Parameters

<i>SEGMENT_NAME</i>	A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.
<code>default</code>	Uses the default segment for constants.

Description

Use this pragma directive to place constant variables in a named segment. The segment name cannot be a segment name predefined for use by the compiler and linker. The setting remains active until you turn it off again with the `#pragma constseg=default` directive.

Example

```
#pragma constseg=MY_CONSTANTS
const int factorySettings[] = {42, 15, -128, 0};
#pragma constseg=default
```

data_alignment

Syntax

```
#pragma data_alignment=expression
```

Parameters

expression A constant which must be a power of two (1, 2, 4, etc.).

Description

Use this pragma directive to give a variable a higher (more strict) alignment than it would otherwise have. It can be used on variables with static and automatic storage duration.

When you use this directive on variables with automatic storage duration, there is an upper limit on the allowed alignment for each function, determined by the calling convention used.

dataseg

Syntax

```
#pragma dataseg={SEGMENT_NAME|default}
```

Parameters

SEGMENT_NAME A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.

default Uses the default segment.

Description

Use this pragma directive to place variables in a named segment. The segment name cannot be a segment name predefined for use by the compiler and linker. The variable will not be initialized at startup, and can for this reason not have an initializer, which means it must be declared `__no_init`. The setting remains active until you turn it off again with the `#pragma constseg=default` directive.

Example

```
#pragma dataseg=MY_SEGMENT
__no_init char myBuffer[1000];
#pragma dataseg=default
```

diag_default

Syntax	<code>#pragma diag_default=tag[, tag, ...]</code>
Parameters	<i>tag</i> The number of a diagnostic message, for example the message number <code>Pe117</code>
Description	Use this pragma directive to change the severity level back to default, or to the severity level defined on the command line by using any of the options <code>--diag_error</code> , <code>--diag_remark</code> , <code>--diag_suppress</code> , or <code>--diag_warnings</code> , for the diagnostic messages specified with the tags.
See also	<i>Diagnostics</i> , page 119.

diag_error

Syntax	<code>#pragma diag_error=tag[, tag, ...]</code>
Parameters	<i>tag</i> The number of a diagnostic message, for example the message number <code>Pe117</code>
Description	Use this pragma directive to change the severity level to <code>error</code> for the specified diagnostics.
See also	<i>Diagnostics</i> , page 119.

diag_remark

Syntax	<code>#pragma diag_remark=tag[, tag, ...]</code>
Parameters	<i>tag</i> The number of a diagnostic message, for example the message number <code>Pe177</code>
Description	Use this pragma directive to change the severity level to <code>remark</code> for the specified diagnostic messages.
See also	<i>Diagnostics</i> , page 119.

diag_suppress

Syntax	<code>#pragma diag_suppress=tag[, tag, ...]</code>
Parameters	<p><i>tag</i> The number of a diagnostic message, for example the message number Pe117</p>
Description	Use this pragma directive to suppress the specified diagnostic messages.
See also	<i>Diagnostics</i> , page 119.

diag_warning

Syntax	<code>#pragma diag_warning=tag[, tag, ...]</code>
Parameters	<p><i>tag</i> The number of a diagnostic message, for example the message number Pe826</p>
Description	Use this pragma directive to change the severity level to <code>warning</code> for the specified diagnostic messages.
See also	<i>Diagnostics</i> , page 119.

include_alias

Syntax	<pre>#pragma include_alias "orig_header" "subst_header" #pragma include_alias <orig_header> <subst_header></pre>
Parameters	<p><i>orig_header</i> The name of a header file for which you want to create an alias.</p> <p><i>subst_header</i> The alias for the original header file.</p>
Description	<p>Use this pragma directive to provide an alias for a header file. This is useful for substituting one header file with another, and for specifying an absolute path to a relative file.</p> <p>This pragma directive must appear before the corresponding <code>#include</code> directives and <code>subst_header</code> must match its corresponding <code>#include</code> directive exactly.</p>

Example

```
#pragma include_alias <stdio.h> <C:\MyHeaders\stdio.h>
#include <stdio.h>
```

This example will substitute the relative file `stdio.h` with a counterpart located according to the specified path.

See also *Include file search procedure*, page 116.

inline

Syntax `#pragma inline[=forced]`

Parameters

`forced` Disables the compiler's heuristics and forces inlining.

Description

Use this pragma directive to advise the compiler that the function whose declaration follows immediately after the directive should be inlined—that is, expanded into the body of the calling function. Whether the inlining actually takes place is subject to the compiler's heuristics.

This is similar to the C++ keyword `inline`, but has the advantage of being available in C code.

Specifying `#pragma inline=forced` disables the compiler's heuristics and forces inlining. If the inlining fails for some reason, for example if it cannot be used with the function type in question (like `printf`), an error message is emitted.

language

Syntax `#pragma language={extended|default}`

Parameters

`extended` Turns on the IAR Systems language extensions and turns off the `--strict_ansi` command line option.

`default` Uses the language settings specified by compiler options.

Description

Use this pragma directive to enable the compiler language extensions or for using the language settings specified on the command line.

location

Syntax	<code>#pragma location={<i>address</i> <i>SEGMENT_NAME</i>}</code>
Parameters	<p><i>address</i> The absolute address of the global or static variable for which you want an absolute location.</p> <p><i>SEGMENT_NAME</i> A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.</p>
Description	Use this pragma directive to specify the location—the absolute address—of the global or static variable whose declaration follows the pragma directive. The variable must be declared either <code>__no_init</code> or <code>const</code> . Alternatively, the directive can take a string specifying a segment for placing either a variable or a function whose declaration follows the pragma directive.
Example	<pre>#pragma location=0x022E __no_init volatile char PORT1; /* PORT1 is located at address 0x022E */ #pragma location="foo" char PORT1; /* PORT1 is located in segment foo */ /* A better way is to use a corresponding mechanism */ #define FLASH _Pragma("location=\"FLASH\"") ... FLASH int i; /* i is placed in the FLASH segment */</pre>
See also	<i>Controlling data and function placement in memory</i> , page 33.

message

Syntax	<code>#pragma message(<i>message</i>)</code>
Parameters	<i>message</i> The message that you want to direct to <code>stdout</code> .
Description	Use this pragma directive to make the compiler print a message to <code>stdout</code> when the file is compiled.
Example:	<pre>#ifdef TESTING #pragma message("Testing") #endif</pre>

no_epilogue

Syntax	<code>#pragma no_epilogue</code>
Description	Use this pragma directive to use a local return sequence instead of a call to the library routine <code>?EpilogueN</code> . This pragma directive can be used when a function needs to exist on its own as in for example a bootloader that needs to be independent of the libraries it is replacing.
Example	<pre>#pragma no_epilogue void bootloader(void) @"BOOTSECTOR" {...</pre>

object_attribute

Syntax	<code>#pragma object_attribute=<i>object_attribute</i>[, <i>object_attribute</i>, ...]</code>
Parameters	For a list of object attributes that can be used with this pragma directive, see <i>Object attributes</i> , page 173.
Description	Use this pragma directive to declare a variable or a function with an object attribute. This directive affects the definition of the identifier that follows immediately after the directive. The object is modified, not its type. Unlike the directive <code>#pragma type_attribute</code> that specifies the storing and accessing of a variable or function, it is not necessary to specify an object attribute in declarations.
Example	<pre>#pragma object_attribute=__no_init char bar;</pre>
See also	<i>General syntax rules for extended keywords</i> , page 171.

optimize

Syntax	<code>#pragma optimize=<i>param</i>[, <i>param</i>, ...]</code>								
Parameters	<table> <tr> <td><code>s</code></td> <td>Optimizes for speed</td> </tr> <tr> <td><code>z</code></td> <td>Optimizes for size</td> </tr> <tr> <td><code>2 none 3 low 6 medium 9 high</code></td> <td>Specifies the level of optimization</td> </tr> <tr> <td><code>no_code_motion</code></td> <td>Turns off code motion</td> </tr> </table>	<code>s</code>	Optimizes for speed	<code>z</code>	Optimizes for size	<code>2 none 3 low 6 medium 9 high</code>	Specifies the level of optimization	<code>no_code_motion</code>	Turns off code motion
<code>s</code>	Optimizes for speed								
<code>z</code>	Optimizes for size								
<code>2 none 3 low 6 medium 9 high</code>	Specifies the level of optimization								
<code>no_code_motion</code>	Turns off code motion								

<code>no_cse</code>	Turns off common subexpression elimination
<code>no_inline</code>	Turns off function inlining
<code>no_tbaa</code>	Turns off type-based alias analysis
<code>no_unroll</code>	Turns off loop unrolling

Description

Use this pragma directive to decrease the optimization level, or to turn off some specific optimizations. This pragma directive only affects the function that follows immediately after the directive.

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 preprocessor macros embedded in this pragma directive. Any such macro will not be expanded by the preprocessor.

Note: If you use the `#pragma optimize` directive to specify an optimization level that is higher than the optimization level you specify using a compiler option, the pragma directive is ignored.

Example

```
#pragma optimize=s 9
int small_and_used_often()
{
    ...
}

#pragma optimize=z 9
int big_and_seldom_used()
{
    ...
}
```

pack

Syntax

```
#pragma pack(n)
#pragma pack()
#pragma pack({push|pop}[, name] [, n])
```

Parameters

<code>n</code>	Sets an optional structure alignment; one of: 1, 2, 4, 8, or 16
Empty list	Restores the structure alignment to default
<code>push</code>	Sets a temporary structure alignment

<code>pop</code>	Restores the structure alignment from a temporarily pushed alignment
<code>name</code>	An optional pushed or popped alignment label

Description

Use this pragma directive to specify the alignment of structs and union members.

The `#pragma pack` directive affects declarations of structures following the pragma directive to the next `#pragma pack` or end of file.

Note that accessing an object that is not aligned at its correct alignment requires code that is both larger and slower than the code needed to access the same kind of object when aligned correctly. If there are many accesses to such fields in the program, it is usually better to construct the correct values in a struct that is not packed, and access this instead.

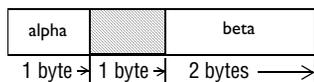
Also, special care is needed when creating and using pointers to misaligned fields. For direct access to misaligned fields in a packed struct, the compiler will emit the correct (but slower and larger) code when needed. However, when a misaligned field is accessed through a pointer to the field, the normal (smaller and faster) code for accessing the type of the field is used. In the general case, this will not work.

Example 1

This example declares a structure without using the `#pragma pack` directive:

```
struct First
{
    char alpha;
    short beta;
};
```

In this example, the structure `First` is not packed and has the following memory layout:



Note that one pad byte has been added.

Example 2

This example declares a similar structure using the `#pragma pack` directive:

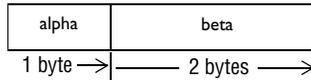
```
#pragma pack(1)

struct FirstPacked
{
    char alpha;
    short beta;
```

```
};

#pragma pack()
```

In this example, the structure `FirstPacked` is packed and has the following memory layout:

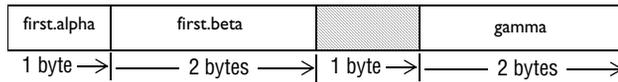


Example 3

This example declares a new structure, `Second`, that contains the structure `FirstPacked` declared in the previous example. The declaration of `Second` is not placed inside a `#pragma pack` block:

```
struct Second
{
    struct FirstPacked first;
    short gamma;
};
```

The following memory layout is used:



Note that the structure `FirstPacked` will use the memory layout, size, and alignment described in Example 2. The alignment of the member `gamma` is 2, which means that alignment of the structure `Second` will become 2 and one pad byte will be added.

required

Syntax

```
#pragma required=symbol
```

Parameters

symbol Any statically linked function or variable.

Description

Use this pragma directive to ensure that a symbol which is needed by a second symbol is included in the linked output. The directive must be placed immediately before the second symbol.

Use the directive if the requirement for a symbol is not otherwise visible in the application, for example if a variable is only referenced indirectly through the segment it resides in.

Example

```
const char copyright[] = "Copyright by me";
...
#pragma required=copyright
int main[]
{...}
```

Even if the copyright string is not used by the application, it will still be included by the linker and available in the output.

rtmodel**Syntax**

```
#pragma rtmodel="key", "value"
```

Parameters

"key"	A text string that specifies the runtime model attribute.
"value"	A text string that specifies the value of the runtime model attribute. Using the special value * is equivalent to not defining the attribute at all.

Description

Use this pragma directive to add a runtime model attribute to a module, which can be used by the linker to check consistency between modules.

This pragma directive is useful for enforcing consistency between modules. All modules that are linked together and define the same runtime attribute key must have the same value for the corresponding key, or the special value *. It can, however, be useful to state explicitly that the module can handle any runtime model.

A module can have several runtime model definitions.

Note: The predefined compiler runtime model attributes start with a double underscore. In order to avoid confusion, this style must not be used in the user-defined attributes.

Example

```
#pragma rtmodel="I2C", "ENABLED"
```

The linker will generate an error if a module that contains this definition is linked with a module that does not have the corresponding runtime model attributes defined.

See also

Checking module consistency, page 64.

segment

Syntax	<code>#pragma segment="SEGMENT_NAME" [align]</code>	
Parameters	<code>"SEGMENT_NAME"</code>	The name of the segment
	<code>align</code>	Aligns the segment part. The value must be a constant integer expression to the power of two.
Description	Use this pragma directive to declare a segment name that can be used by the segment operators <code>__segment_begin</code> and <code>__segment_end</code> . All segment declarations for a specific segment must have the same memory type attribute and alignment.	
Example	<code>#pragma segment="MYSEG" 4</code>	
See also	<i>Important language extensions</i> , page 161. For more information about segments and segment parts, see the chapter <i>Placing code and data</i> .	

type_attribute

Syntax	<code>#pragma type_attribute=type_attribute[, type_attribute, ...]</code>	
Parameters	For a list of type attributes that can be used with this pragma directive, see <i>Type attributes</i> , page 171.	
Description	Use this pragma directive to specify IAR-specific <i>type attributes</i> , which are not part of the ISO/ANSI C language standard. Note however, that a given type attribute may not be applicable to all kind of objects.	
	This directive affects the declaration of the identifier, the next variable, or the next function that follows immediately after the pragma directive.	
Example	<p>In the following example, an <code>int</code> object with the memory attribute <code>__regvar</code> is defined:</p> <pre>#include "intrinsics.h" #pragma type_attribute=__regvar #pragma object_attribute=__no_init #pragma location=__R4 short x;</pre> <p>The following declaration, which uses extended keywords, is equivalent:</p> <pre>__no_init __regvar short x @__R4;</pre>	

See also See the chapter *Extended keywords* for more details.

vector

Syntax `#pragma vector=vector1[, vector2, vector3, ...]`

Parameters *vector* The vector number(s) of an interrupt or trap function.

Description Use this pragma directive to specify the vector(s) of an interrupt or trap function whose declaration follows the pragma directive. Note that several vectors can be defined for each function.

Example

```
#pragma vector=0x14
__interrupt void my_handler(void);
```


Intrinsic functions

This chapter gives reference information about the intrinsic functions, a predefined set of functions available in the compiler.

The intrinsic functions provide direct access to low-level processor operations and can be very useful in, for example, time-critical routines. The intrinsic functions compile into inline code, either as a single instruction or as a short sequence of instructions.

Intrinsic functions summary

To use intrinsic functions in an application, include the header file `intrinsics.h`.

Note that the intrinsic function names start with double underscores, for example:

```
__disable_interrupt
```

The following table summarizes the intrinsic functions:

Intrinsic function	Description
<code>__bcd_add_type</code>	Performs a binary coded decimal operation
<code>__bic_SR_register</code>	Clears bits in the SR register
<code>__bic_SR_register_on_exit</code>	Clears bits in the SR register when an interrupt or monitor function returns
<code>__bis_SR_register</code>	Sets bits in the SR register
<code>__bis_SR_register_on_exit</code>	Sets bits in the SR register when an interrupt or monitor function returns
<code>__data16_read_addr</code>	Reads data to a 20-bit SFR register
<code>__data16_write_addr</code>	Writes data to a 20-bit SFR register
<code>__data20_read_type</code>	Reads data which has a 20-bit address
<code>__data20_write_type</code>	Writes data which has a 20-bit address
<code>__disable_interrupt</code>	Disables interrupts
<code>__enable_interrupt</code>	Enables interrupts
<code>__even_in_range</code>	Instructs the compiler to rely on the specified value being even and within the specified range
<code>__get_interrupt_state</code>	Returns the interrupt state

Table 30: Intrinsic functions summary

Intrinsic function	Description
<code>__get_R4_register</code>	Returns the value of the R4 register
<code>__get_R5_register</code>	Returns the value of the R5 register
<code>__get_SP_register</code>	Returns the value of the stack pointer
<code>__get_SR_register</code>	Returns the value of the SR register
<code>__get_SR_register_on_exit</code>	Returns the value of the processor status register when the current interrupt or monitor function returns
<code>__low_power_mode_n</code>	Enters a MSP430 low power mode
<code>__low_power_mode_off_on_exit</code>	Turns off low power mode when monitor or interrupt function returns
<code>__no_operation</code>	Inserts a NOP instruction
<code>__op_code</code>	Inserts a constant into the instruction stream
<code>__set_interrupt_state</code>	Restores the interrupt state
<code>__set_R4_register</code>	Writes a specific value to the R4 register
<code>__set_R5_register</code>	Writes a specific value to the R5 register
<code>__set_SP_register</code>	Writes a specific value to the SP register
<code>__swap_bytes</code>	Executes the SWPB instruction

Table 30: Intrinsic functions summary (Continued)

Descriptions of intrinsic functions

The following section gives reference information about each intrinsic function.

`__bcd_add_type`

Syntax

```
unsigned type __bcd_add_type(unsigned type x, unsigned type y);
```

where:

type Can be one of the types short, long, or long long

Description Performs a binary coded decimal addition. The parameters and the return value are represented as binary coded decimal (BCD) numbers. The following functions are supported:

Function	Return value
<code>__bcd_add_short</code>	Returns the sum of the two <code>short</code> parameters. The parameters and the return value are represented as four-digit BCD numbers.
<code>__bcd_add_long</code>	Returns the sum of the two <code>long</code> parameters. The parameters and the return value are represented as eight-digit BCD numbers.
<code>__bcd_add_long_long</code>	Returns the sum of the two <code>long long</code> parameters. The parameters and the return value are represented as sixteen-digit BCD numbers.

Table 31: Functions for binary coded decimal operations

Example

```
/* c = 0x19 */
c = __bcd_add_short(c, 0x01);
/* c = 0x20 */
```

`__bic_SR_register`

Syntax

```
void __bic_SR_register(unsigned short);
```

Description

Clears bits in the processor status register. The function takes an integer as its argument, that is, a bit mask with the bits to be cleared.

`__bic_SR_register_on_exit`

Syntax

```
void __bic_SR_register_on_exit(unsigned short);
```

Description

Clears bits in the processor status register when an interrupt or monitor function returns. The function takes an integer as its argument, that is, a bit mask with the bits to be cleared.

This intrinsic function is only available in interrupt and monitor functions.

__bis_SR_register

Syntax `void __bis_SR_register(unsigned short);`

Description Sets bits in the status register. The function takes an integer literal as its argument, that is, a bit mask with the bits to be set.

__bis_SR_register_on_exit

Syntax `void __bis_SR_register_on_exit(unsigned short);`

Description Sets bits in the processor status register when an interrupt or monitor function returns. The function takes an integer literal as its argument, that is, a bit mask with the bits to be set.

This intrinsic function is only available in interrupt and monitor functions.

__data16_read_addr

Syntax `unsigned long __data16_read_addr(unsigned short address);`

where:

address Specifies the address for the read operation

Description Reads data from a 20-bit SFR register located at the given 16-bit address. This intrinsic function is only useful on devices based on the MSP430X architecture.

__data16_write_addr

Syntax `void __data16_write_addr(unsigned short address,
unsigned long data);`

where:

address Specifies the address for the write operation

data The data to be written

Description

Writes a value to a 20-bit SFR register located at the given 16-bit address. This intrinsic function is only useful on devices based on the MSP430X architecture.

__data20_read_type

Syntax

```
unsigned type __data20_read_type(unsigned long address);
```

where:

address Specifies the address for the read operation
type Can be one of the types `char`, `short`, or `long`

Description

Reads data from the given 20-bit address. This intrinsic function is only useful on devices based on the MSP430X architecture and must be used when reading data that is located above the first 64 Kbytes of memory. The following functions are supported:

Function	Operation size	Alignment
unsigned char <code>__data20_read_char</code>	8	1
unsigned short <code>__data20_read_short</code>	16	2
unsigned long <code>__data20_read_long</code>	2*16	2

Table 32: Functions for reading data that has a 20-bit address

__data20_write_type

Syntax

```
void __data20_write_type(unsigned long address, unsigned type);
```

where:

address Specifies the address for the write operation
type Can be one of the types `char`, `short`, or `long`

Description

Writes a value to the given 20-bit address. This intrinsic function is only useful on devices based on the MSP430X architecture and must be used when writing data that is located above the first 64 Kbytes of memory. The following functions are supported:

Function	Operation size	Alignment
unsigned char <code>__data20_write_char</code>	8	1
unsigned short <code>__data20_write_short</code>	16	2

Table 33: Functions for writing data that has a 20-bit address

Function	Operation size	Alignment
unsigned long __data20_write_long	2*16	2

Table 33: Functions for writing data that has a 20-bit address (Continued)

__disable_interrupt

Syntax

```
void __disable_interrupt(void);
```

Description

Disables interrupts by inserting the `DINT` instruction followed by a `NOP` instruction. The `NOP` instruction ensures that interrupts have been disabled on the device before the execution resumes.

__enable_interrupt

Syntax

```
void __enable_interrupt(void);
```

Description

Enables interrupts by inserting the `EINT` instruction.

__even_in_range

Syntax

```
unsigned short __even_in_range(unsigned short value,
                               unsigned short upper_limit);
```

Parameters

<i>value</i>	The switch expression
<i>upper_limit</i>	The last value in the allowed range

Description

Instructs the compiler to rely on the specified value being even and within the specified range. The code will be generated accordingly and will only work if the requirement is fulfilled.

This intrinsic function can be used for getting optimal code for switch statements where you know that the only values possible are even values within a given range, for example an interrupt service routine for an Interrupt Vector Generator interrupt.

Example

```
switch (__even_in_range(TAIV, 10))
```

See also

Interrupt Vector Generator interrupt functions, page 17.

__get_interrupt_state

Syntax `__istate_t __get_interrupt_state(void);`

Description Returns the global interrupt state. The return value can be used as an argument to the `__set_interrupt_state` intrinsic function, which will restore the interrupt state.

Example

```
__istate_t s = __get_interrupt_state();
__disable_interrupt();
```

```
/* Do something */
```

```
__set_interrupt_state(s);
```

The advantage of using this sequence of code compared to using `__disable_interrupt` and `__enable_interrupt` is that the code in this example will not enable any interrupts disabled.

__get_R4_register

Syntax `unsigned short __get_R4_register(void);`

Description Returns the value of the R4 register. This intrinsic function is only available when the register is locked.

See also `--lock_r4`, page 136

__get_R5_register

Syntax `unsigned short __get_R5_register(void);`

Description Returns the value of the R5 register. This intrinsic function is only available when the register is locked.

See also `--lock_r5`, page 136

__get_SP_register

Syntax `unsigned short __get_SP_register(void);`

Description Returns the value of the stack pointer register SP.

__get_SR_register

Syntax `unsigned short __get_SR_register(void);`

Description Returns the value of the processor status register SR.

__get_SR_register_on_exit

Syntax `unsigned short __get_SR_register_on_exit(void);`

Description Returns the value of the processor status register SR that will be set when the current interrupt or monitor function returns.

This intrinsic function is only available in interrupt and monitor functions.

__low_power_mode_n

Syntax `void __low_power_mode_n(void);`

Description Enters a MSP430 low power mode, where *n* can be one of 0–4.

__low_power_mode_off_on_exit

Syntax `void __low_power_mode_off_on_exit(void);`

Description Turns off the low power mode when a monitor or interrupt function returns. This intrinsic function is only available in interrupt and monitor functions.

__no_operation

Syntax `void __no_operation(void);`

Description Inserts a NOP instruction.

__op_code

Syntax `void __op_code(unsigned short);`

Description Emits the 16-bit value into the instruction stream for the current function by inserting a DC16 constant.

__set_interrupt_state

Syntax	<code>void __set_interrupt_state(__istate_t);</code>
Descriptions	Restores the interrupt state by setting the value returned by the <code>__get_interrupt_state</code> function. For information about the <code>__istate_t</code> type, see <code>__get_interrupt_state</code> , page 203.

__set_R4_register

Syntax	<code>void __set_R4_register(unsigned short);</code>
Description	Writes a specific value to the R4 register. This intrinsic function is only available when R4 is locked.
See also	<code>--lock_r4</code> , page 136

__set_R5_register

Syntax	<code>void __set_R5_register(unsigned short);</code>
Description	Writes a specific value to the R5 register. This intrinsic function is only available when R5 is locked.
See also	<code>--lock_r5</code> , page 136

__set_SP_register

Syntax	<code>void __set_SP_register(unsigned short);</code>
Description	Writes a specific value to the SP stack pointer register. A warning message is issued if the compiler has used the stack in any way at the location where this intrinsic function is used.

__swap_bytes

Syntax	<code>unsigned short __swap_bytes(unsigned short);</code>
Description	Inserts an <code>SWAPB</code> instruction and returns the argument with the upper and lower parts interchanged.

Example

```
__swap_bytes(0x1234)  
returns 0x3412.
```

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 MSP430 IAR C/C++ Compiler adheres to the ISO/ANSI standard. The compiler also makes the following preprocessor-related features available to you:

- Predefined preprocessor symbols

These symbols allow you to inspect the compile-time environment, for example the time and date of compilation. For details, see *Descriptions of predefined preprocessor symbols*, page 208.

- User-defined preprocessor symbols defined using a compiler option

In addition to defining your own preprocessor symbols using the `#define` directive, you can also use the option `-D`, see *-D*, page 127.

- Preprocessor extensions

There are several preprocessor extensions, for example many `pragma` directives; for more information, see the chapter *Pragma directives* in this guide. Read also about the corresponding `_Pragma` operator and the other extensions related to the preprocessor, see *Descriptions of miscellaneous preprocessor extensions*, page 210.

- Preprocessor output

Use the option `--preprocess` to direct preprocessor output to a named file, see *--preprocess*, page 143.

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 234.

Descriptions of predefined preprocessor symbols

The following table describes the predefined preprocessor symbols:

Predefined symbol	Identifies
<code>__BASE_FILE__</code>	A string that identifies the name of the base source file (that is, not the header file), being compiled. See also <code>__FILE__</code> , page 208.
<code>__BUILD_NUMBER__</code>	A unique integer that identifies the build number of the compiler currently in use. The build number does not necessarily increase with a compiler that is released later.
<code>__CORE__</code>	An integer that identifies the chip architecture in use. The symbol reflects the <code>--core</code> option and is defined to <code>__430__</code> for the MSP430 architecture and to <code>__430X__</code> for the MSP430X architecture. These symbolic names can be used when testing the <code>__CORE__</code> symbol.
<code>__cplusplus</code>	An integer which is defined when the compiler runs in any of the C++ modes, otherwise it is undefined. When defined, its value is 199711L. This symbol can be used with <code>#ifdef</code> to detect whether the compiler accepts C++ code. It is particularly useful when creating header files that are to be shared by C and C++ code.*
<code>__DATE__</code>	A string that identifies the date of compilation, which is returned in the form "Mmm dd yyyy", for example "Oct 30 2005".*
<code>__embedded_cplusplus</code>	An integer which is defined to 1 when the compiler runs in any of the C++ modes, otherwise the symbol is undefined. This symbol can be used with <code>#ifdef</code> to detect whether the compiler accepts C++ code. It is particularly useful when creating header files that are to be shared by C and C++ code.*
<code>__FILE__</code>	A string that identifies the name of the file being compiled, which can be the base source file as well as any included header file. See also <code>__BASE_FILE__</code> , page 208.*
<code>__func__</code>	A string that identifies the name of the function in which the symbol is used. This is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see <code>-e</code> , page 132. See also <code>__PRETTY_FUNCTION__</code> , page 209.

Table 34: Predefined symbols

Predefined symbol	Identifies
<code>__FUNCTION__</code>	A string that identifies the name of the function in which the symbol is used. This is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see <code>-e</code> , page 132. See also <code>__PRETTY_FUNCTION__</code> , page 209.
<code>__IAR_SYSTEMS_ICC__</code>	An integer that identifies the IAR compiler platform. The current value is 6. Note that the number could be higher in a future version of the product. This symbol can be tested with <code>#ifdef</code> to detect whether the code was compiled by a compiler from IAR Systems.
<code>__ICC430__</code>	An integer that is set to 1 when the code is compiled with the MSP430 IAR C/C++ Compiler.
<code>__LINE__</code>	An integer that identifies the current source line number of the file being compiled, which can be the base source file as well as any included header file.*
<code>__PRETTY_FUNCTION__</code>	A string that identifies the function name, including parameter types and return type, of the function in which the symbol is used, for example <code>"void func(char)"</code> . This symbol is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see <code>-e</code> , page 132. See also <code>__func__</code> , page 208.
<code>__STDC__</code>	An integer that is set to 1, which means the compiler adheres to the ISO/ANSI C standard. This symbol can be tested with <code>#ifdef</code> to detect whether the compiler in use adheres to ISO/ANSI C.*
<code>__STDC_VERSION__</code>	An integer that identifies the version of ISO/ANSI C standard in use. The symbol expands to 199409L. This symbol does not apply in EC++ mode.*
<code>__SUBVERSION__</code>	An integer that identifies the version letter of the version number, for example the C in 4.21C, as an ASCII character.
<code>__TIME__</code>	A string that identifies the time of compilation in the form <code>"hh:mm:ss"</code> .*

Table 34: Predefined symbols (Continued)

Predefined symbol	Identifies
<code>__VER__</code>	An integer that identifies the version number of the IAR compiler in use. The value of the number is calculated in the following way: (100 * the major version number + the minor version number). For example, for compiler version 3.34, 3 is the major version number and 34 is the minor version number. Hence, the value of <code>__VER__</code> is 334.

Table 34: Predefined symbols (Continued)

* This symbol is required by the ISO/ANSI standard.

Note: The predefined symbol `__TID__` is recognized for backward compatibility, but it is recommended to use the symbols `__CORE__` and `__ICC430__` instead.

Descriptions of miscellaneous preprocessor extensions

The following section gives reference information about the preprocessor extensions that are available in addition to the predefined symbols, pragma directives, and ISO/ANSI directives.

NDEBUG

Description

This preprocessor symbol determines whether any assert macro you have written in your application shall be included or not in the built application.

If this symbol is not defined, all assert macros are evaluated. If the symbol is defined, all assert macros are excluded from the compilation. In other words, if the symbol is:

- **defined**, the assert code will *not* be included
- **not defined**, the assert code will be included

This means that if you have written any assert code and build your application, you should define this symbol to exclude the assert code from the final application.

Note that the assert macro is defined in the `assert.h` standard include file.



In the IAR Embedded Workbench IDE, the `NDEBUG` symbol is automatically defined if you build your application in the Release build configuration.

`_Pragma()`

Syntax

```
_Pragma("string")
```

where *string* follows the syntax of the corresponding pragma directive.

Description	This preprocessor operator is part of the C99 standard and can be used, for example, in defines and has the equivalent effect of the <code>#pragma</code> directive. Note: The <code>-e</code> option—enable language extensions—does not have to be specified.
Example	<pre>#if NO_OPTIMIZE #define NOOPT _Pragma("optimize=2") #else #define NOOPT #endif</pre>
See also	See the chapter <i>Pragma directives</i> .

#warning message

Syntax	<code>#warning message</code> where <i>message</i> can be any string.
Description	Use this preprocessor directive to produce messages. Typically, this is useful for assertions and other trace utilities, similar to the way the ISO/ANSI standard <code>#error</code> directive is used.

__VA_ARGS__

Syntax	<pre>#define P(...) __VA_ARGS__ #define P(x,y,...) x + y + __VA_ARGS__</pre> <code>__VA_ARGS__</code> will contain all variadic arguments concatenated, including the separating commas.
Description	Variadic macros are the preprocessor macro equivalents of <code>printf</code> style functions. <code>__VA_ARGS__</code> is part of the C99 standard.
Example	<pre>#if DEBUG #define DEBUG_TRACE(...) printf(S,__VA_ARGS__) #else #define DEBUG_TRACE(...) #endif ... DEBUG_TRACE("The value is:%d\n",value);</pre> will result in: <pre>printf("The value is:%d\n",value);</pre>

Library functions

This chapter gives an introduction to the C and C++ library functions. It also lists the header files used for accessing library definitions.

For detailed reference information about the library functions, see the online help system.

Introduction

The MSP430 IAR C/C++ Compiler provides two different libraries:

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

Note that different customization methods are normally needed for these two libraries. For additional information, see the chapter *The DLIB runtime environment* and *The CLIB runtime environment*, respectively.

For detailed information about the library functions, see the online documentation supplied with the product. There is also keyword reference information for the DLIB library functions. To obtain reference information for a function, select the function name in the editor window and press F1.

For additional information about library functions, see the chapter *Implementation-defined behaviour* in this guide.

HEADER FILES

Your application program gains access to library definitions through header files, which it incorporates using the `#include` directive. The definitions are divided into a number of different header files, each covering a particular functional area, letting you include just those that are required.

It is essential to include the appropriate header file before making any reference to its definitions. Failure to do so can cause the call to fail during execution, or generate error or warning messages at compile time or link time.

LIBRARY OBJECT FILES

Most of the library definitions can be used without modification, that is, directly from the library object files that are supplied with the product. For information about how to choose a runtime library, see *Basic settings for project configuration*, page 5. The linker will include only those routines that are required—directly or indirectly—by your application.

REENTRANCY

A function that can be simultaneously invoked in the main application and in any number of interrupts is reentrant. A library function that uses statically allocated data is therefore not reentrant. Most parts of the DLIB library are reentrant, but the following functions and parts are not reentrant:

<code>atexit</code>	Needs static data
heap functions	Need static data for memory allocation tables
<code>strerror</code>	Needs static data
<code>strtok</code>	Designed by ISO/ANSI standard to need static data
I/O	Every function that uses files in some way. This includes <code>printf</code> , <code>scanf</code> , <code>getchar</code> , and <code>putchar</code> . The functions <code>sprintf</code> and <code>sscanf</code> are not included.

In addition, some functions share the same storage for `errno`. These functions are not reentrant, since an `errno` value resulting from one of these functions can be destroyed by a subsequent use of the function before it has been read. Among these functions are:

```
exp, exp10, ldexp, log, log10, pow, sqrt, acos, asin, atan2,
cosh, sinh, strtod, strtol, strtoul
```

Remedies for this are:

- Do not use non-reentrant functions in interrupt service routines
- Guard calls to a non-reentrant function by a mutex, or a secure region, etc.

IAR DLIB Library

The IAR DLIB Library provides most of the important C and C++ library definitions that apply to embedded systems. These are of the following types:

- Adherence to a free-standing implementation of the ISO/ANSI standard for the programming language C. For additional information, see the chapter *Implementation-defined behavior* in this guide.

- Standard C library definitions, for user programs.
- Embedded C++ library definitions, for user programs.
- `CSTARTUP`, the module containing the start-up code. It is described in the chapter *The DLIB runtime environment* in this guide.
- Runtime support libraries; for example low-level floating-point routines.
- Intrinsic functions, allowing low-level use of MSP430 features. See the chapter *Intrinsic functions* for more information.

In addition, the IAR DLIB Library includes some added C functionality, partly taken from the C99 standard, see *Added C functionality*, page 218.

C HEADER FILES

This section lists the header files specific to the DLIB library C definitions. Header files may additionally contain target-specific definitions; these are documented in the chapter *Compiler extensions*.

The following table lists the C header files:

Header file	Usage
<code>assert.h</code>	Enforcing assertions when functions execute
<code>ctype.h</code>	Classifying characters
<code>errno.h</code>	Testing error codes reported by library functions
<code>float.h</code>	Testing floating-point type properties
<code>iso646.h</code>	Using Amendment 1— <code>iso646.h</code> standard header
<code>limits.h</code>	Testing integer type properties
<code>locale.h</code>	Adapting to different cultural conventions
<code>math.h</code>	Computing common mathematical functions
<code>setjmp.h</code>	Executing non-local goto statements
<code>signal.h</code>	Controlling various exceptional conditions
<code>stdarg.h</code>	Accessing a varying number of arguments
<code>stdbool.h</code>	Adds support for the <code>bool</code> data type in C.
<code>stddef.h</code>	Defining several useful types and macros
<code>stdio.h</code>	Performing input and output
<code>stdlib.h</code>	Performing a variety of operations
<code>string.h</code>	Manipulating several kinds of strings
<code>time.h</code>	Converting between various time and date formats
<code>wchar.h</code>	Support for wide characters
<code>wctype.h</code>	Classifying wide characters

Table 35: Traditional standard C header files—DLIB

C++ HEADER FILES

This section lists the C++ header files.

Embedded C++

The following table lists the Embedded C++ header files:

Header file	Usage
<code>complex</code>	Defining a class that supports complex arithmetic
<code>exception</code>	Defining several functions that control exception handling
<code>fstream</code>	Defining several I/O stream classes that manipulate external files
<code>iomanip</code>	Declaring several I/O stream manipulators that take an argument
<code>ios</code>	Defining the class that serves as the base for many I/O streams classes
<code>iosfwd</code>	Declaring several I/O stream classes before they are necessarily defined
<code>iostream</code>	Declaring the I/O stream objects that manipulate the standard streams
<code>istream</code>	Defining the class that performs extractions
<code>new</code>	Declaring several functions that allocate and free storage
<code>ostream</code>	Defining the class that performs insertions
<code>sstream</code>	Defining several I/O stream classes that manipulate string containers
<code>stdexcept</code>	Defining several classes useful for reporting exceptions
<code>streambuf</code>	Defining classes that buffer I/O stream operations
<code>string</code>	Defining a class that implements a string container
<code>stringstream</code>	Defining several I/O stream classes that manipulate in-memory character sequences

Table 36: Embedded C++ header files

The following table lists additional C++ header files:

Header file	Usage
<code>fstream.h</code>	Defining several I/O stream classes that manipulate external files
<code>iomanip.h</code>	Declaring several I/O stream manipulators that take an argument
<code>iostream.h</code>	Declaring the I/O stream objects that manipulate the standard streams
<code>new.h</code>	Declaring several functions that allocate and free storage

Table 37: Additional Embedded C++ header files—DLIB

Extended Embedded C++ standard template library

The following table lists the Extended EC++ standard template library (STL) header files:

Header file	Description
<code>algorithm</code>	Defines several common operations on sequences
<code>deque</code>	A deque sequence container
<code>functional</code>	Defines several function objects
<code>hash_map</code>	A map associative container, based on a hash algorithm
<code>hash_set</code>	A set associative container, based on a hash algorithm
<code>iterator</code>	Defines common iterators, and operations on iterators
<code>list</code>	A doubly-linked list sequence container
<code>map</code>	A map associative container
<code>memory</code>	Defines facilities for managing memory
<code>numeric</code>	Performs generalized numeric operations on sequences
<code>queue</code>	A queue sequence container
<code>set</code>	A set associative container
<code>slist</code>	A singly-linked list sequence container
<code>stack</code>	A stack sequence container
<code>utility</code>	Defines several utility components
<code>vector</code>	A vector sequence container

Table 38: Standard template library header files

Using standard C libraries in C++

The C++ library works in conjunction with 15 of the header files from the standard C library, sometimes with small alterations. The header files come in two forms—new and traditional—for example, `cassert` and `assert.h`.

The following table shows the new header files:

Header file	Usage
<code>cassert</code>	Enforcing assertions when functions execute
<code>cctype</code>	Classifying characters
<code>cerrno</code>	Testing error codes reported by library functions
<code>cfloat</code>	Testing floating-point type properties
<code>climits</code>	Testing integer type properties

Table 39: New standard C header files—DLIB

Header file	Usage
<code>locale</code>	Adapting to different cultural conventions
<code>cmath</code>	Computing common mathematical functions
<code>csetjmp</code>	Executing non-local goto statements
<code>csignal</code>	Controlling various exceptional conditions
<code>cstdarg</code>	Accessing a varying number of arguments
<code>cstddef</code>	Defining several useful types and macros
<code>stdio</code>	Performing input and output
<code>stdlib</code>	Performing a variety of operations
<code>cstring</code>	Manipulating several kinds of strings
<code>ctime</code>	Converting between various time and date formats

Table 39: New standard C header files—DLIB (Continued)

ADDED C FUNCTIONALITY

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

The following include files provide these features:

- `ctype.h`
- `inttypes.h`
- `math.h`
- `stdbool.h`
- `stdint.h`
- `stdio.h`
- `stdlib.h`
- `wchar.h`
- `wctype.h`

`ctype.h`

In `ctype.h`, the C99 function `isblank` is defined.

`inttypes.h`

This include file defines the formatters for all types defined in `stdint.h` to be used by the functions `printf`, `scanf`, and all their variants.

`math.h`

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

The following C99 macro symbols are defined:

HUGE_VALF, HUGE_VALL, INFINITY, NAN, FP_INFINITE, FP_NAN, FP_NORMAL, FP_SUBNORMAL, FP_ZERO, MATH_ERRNO, MATH_ERREXCEPT, math_errhandling.

The following C99 macro functions are defined:

fpclassify, signbit, isfinite, isinf, isnan, isnormal, isgreater, isless, islessequal, islessgreater, isunordered.

The following C99 type definitions are added:

float_t, double_t.

stdbool.h

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

stdint.h

This include file provides integer characteristics.

stdio.h

In `stdio.h`, the following C99 functions are defined:

`vscanf`, `vfscanf`, `vsscanf`, `vsnprintf`, `snprintf`

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

The following functions providing I/O functionality for libraries built without `FILE` support are defined:

`__write_array` Corresponds to `fwrite` on `stdout`.

`__ungetchar` Corresponds to `ungetc` on `stdout`.

`__gets` Corresponds to `fgets` on `stdin`.

stdlib.h

In `stdlib.h`, the following C99 functions are defined:

`_Exit`, `llabs`, `lldiv`, `strtoll`, `strtoull`, `atoll`, `strtod`, `strtold`.

The function `strtod` has added functionality from the C99 standard. For reference information about this functions, see the library reference available from the **Help** menu.

The `__qsorttbl1` function is defined; it provides sorting using a bubble sort algorithm. This is useful for applications that have a limited stack.

wchar.h

In `wchar.h`, the following C99 functions are defined:

`vfwscanf`, `vswscanf`, `vscanf`, `wcstof`, `wcstolb`.

wctype.h

In `wctype.h`, the C99 function `iswblank` is defined.

IAR CLIB Library

The IAR CLIB Library provides most of the important C library definitions that apply to embedded systems. These are of the following types:

- Standard C library definitions available for user programs. These are documented in this chapter.
- The system startup code. It is described in the chapter *The CLIB runtime environment* in this guide.
- Runtime support libraries; for example low-level floating-point routines.
- Intrinsic functions, allowing low-level use of MSP430 features. See the chapter *Intrinsic functions* for more information.

LIBRARY DEFINITIONS SUMMARY

This following table lists the header files specific to the CLIB library:

Header file	Description
<code>assert.h</code>	Assertions
<code>cctype.h*</code>	Character handling
<code>errno.h</code>	Error return values
<code>float.h</code>	Limits and sizes of floating-point types
<code>iccbutl.h</code>	Low-level routines
<code>limits.h</code>	Limits and sizes of integral types
<code>math.h</code>	Mathematics

Table 40: IAR CLIB Library header files

Header file	Description
<code>setjmp.h</code>	Non-local jumps
<code>stdarg.h</code>	Variable arguments
<code>stdbool.h</code>	Adds support for the <code>bool</code> data type in C
<code>stddef.h</code>	Common definitions including <code>size_t</code> , <code>NULL</code> , <code>ptrdiff_t</code> , and <code>offsetof</code>
<code>stdio.h</code>	Input/output
<code>stdlib.h</code>	General utilities
<code>string.h</code>	String handling

Table 40: IAR CLIB Library header files (Continued)

*** The functions `isxxx`, `toupper`, and `tolower` declared in the header file `ctype.h` evaluate their argument more than once. This is not according to the ISO/ANSI standard.**

Segment reference

The MSP430 IAR C/C++ Compiler places code and data into named segments which are referred to by the IAR XLINK Linker. Details about the segments are required for programming assembler language modules, and are also useful when interpreting the assembler language output from the compiler.

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

Summary of segments

The table below lists the segments that are available in the MSP430 IAR C/C++ Compiler:

Segment	Description
CODE	Holds the program code.
CSTACK	Holds the stack used by C or C++ programs.
CSTART	Holds the startup code.
DATA16_AC	Holds located constant data.
DATA16_AN	Holds located uninitialized data.
DATA16_C	Holds constant data.
DATA16_I	Holds static and global initialized variables.
DATA16_ID	Holds initial values for static and global variables in DATA16_I.
DATA16_N	Holds <code>__no_init</code> static and global variables.
DATA16_Z	Holds zero-initialized static and global variables.
DIFUNCT	Holds pointers to code, typically C++ constructors, that should be executed by the system startup code before <code>main</code> is called.
HEAP	Holds the heap data used by <code>malloc</code> and <code>free</code> .
INTVEC	Contains the reset and interrupt vectors.
ISR_CODE	Holds interrupt functions when compiling for the MSP430X architecture.
REGVAR_AN	Holds <code>__regvar</code> located uninitialized data.
RESET	Holds the reset vector.

Table 41: Segment summary

Descriptions of segments

This section gives reference information about each segment.

The segments are placed in memory by using the segment placement linker directives `-Z` and `-P`, for sequential and packed placement, respectively. Some segments cannot use packed placement, as their contents must be contiguous.

In each description, the segment memory type—`CODE`, `CONST`, or `DATA`—indicates whether the segment should be placed in ROM or RAM memory; see Table 3, *XLINK segment memory types*, page 24.

For information about the `-Z` and the `-P` directives, see the *IAR Linker and Library Tools Reference Guide*.

For information about how to define segments in the linker command file, see *Customizing the linker command file*, page 25.

For detailed information about the extended keywords mentioned here, see the chapter *Extended keywords*.

CODE

Description	Holds program code. However, when compiling for the MSP430X architecture, the code for system initialization and interrupt functions is not placed in the <code>CODE</code> segment.
Segment memory type	<code>CODE</code>
Memory placement	<code>--core=430:</code> <code>0x0002-0xFFFF</code> <code>--core=430X:</code> <code>0x00002-0xFFFFF</code>
Access type	Read-only

CSTACK

Description	Holds the internal data stack.
Segment memory type	<code>DATA</code>
Memory placement	<code>0x0001-0xFFFE</code>
Access type	Read/write
See also	<i>The stack</i> , page 29.

CSTART

Description	<p>Holds the startup code.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be contiguous. Instead, when you define this segment in the linker command file, the <code>-z</code> directive must be used.</p>
Segment memory type	CODE
Memory placement	0x0002-0xFFFF
Access type	Read-only

DATA16_AC

Description	<p>Holds located constant data.</p> <p>Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. <i>Located</i> means being placed at an absolute location using the <code>@</code> operator or the <code>#pragma location</code> directive.</p>
-------------	---

DATA16_AN

Description	<p>Holds located uninitialized data.</p> <p>Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. <i>Located</i> means being placed at an absolute location using the <code>@</code> operator or the <code>#pragma location</code> directive.</p>
-------------	--

DATA16_C

Description	Holds constant data.
Segment memory type	CONST
Memory placement	0x0001-0xFFFE
Access type	Read-only

DATA16_I

Description	<p>Holds static and global initialized variables initialized by copying from the segment DATA16_ID at application startup.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be contiguous. Instead, when you define this segment in the linker command file, the <code>-z</code> directive must be used.</p>
Segment memory type	DATA
Memory placement	0x0001-0xFFFFE
Access type	Read/write

DATA16_ID

Description	<p>Holds initial values for static and global variables in the DATA16_I segment. These values are copied from DATA16_ID to DATA16_I at application startup.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be contiguous. Instead, when you define this segment in the linker command file, the <code>-z</code> directive must be used.</p>
Segment memory type	CONST
Memory placement	0x0001-0xFFFFE
Access type	Read-only

DATA16_N

Description	Holds static and global <code>__no_init</code> variables.
Segment memory type	DATA
Placement	0x0001-0xFFFFE
Access type	Read/write

DATA16_Z

Description	<p>Holds zero-initialized static and global variables.</p> <p>This segment cannot be placed in memory by using the <code>-P</code> directive for packed placement, because the contents must be contiguous. Instead, when you define this segment in the linker command file, the <code>-z</code> directive must be used.</p>
Segment memory type	DATA
Memory placement	0x0001-0xFFFFE
Access type	Read/write

DIFUNCT

Description	Holds the dynamic initialization vector used by C++.
Segment memory type	CONST
Placement	0x0001-0xFFFFE
Access type	Read-only

HEAP

Description	Holds the heap used for dynamically allocated data, in other words data allocated by <code>malloc</code> and <code>free</code> , and in C++, <code>new</code> and <code>delete</code> .
Segment memory type	DATA
Memory placement	0x0001-0xFFFFE
Access type	Read/write
See also	<i>The heap</i> , page 39.

INTVEC

Description	Holds the interrupt vector table generated by the use of the <code>__interrupt</code> extended keyword in combination with the <code>#pragma vector</code> directive.
Segment memory type	CODE
Placement	0xFFE0-0xFFFF or 0xFFC0-0xFFFE depending on the device.
Access type	Read-only

ISR_CODE

Description	Holds interrupt functions when compiling for the MSP430X architecture. This segment is not used when compiling for the MSP430 architecture.
Segment memory type	CODE
Memory placement	0x0002-0xFFFF
Access type	Read-only

REGVAR_AN

Description	Holds <code>__regvar</code> located uninitialized data. Segments containing located data need no further configuration because they have already been assigned addresses prior to linking. <i>Located</i> means being placed at an absolute location using the <code>@</code> operator or the <code>#pragma location</code> directive.
-------------	---

RESET

Description	Holds the reset vector.
Segment memory type	CODE
Memory placement	0xFFFFE-0xFFFF
Access type	Read-only

Implementation-defined behavior

This chapter describes how the MSP430 IAR C/C++ Compiler handles the implementation-defined areas of the C language.

ISO 9899:1990, the International Organization for Standardization standard - *Programming Languages - C* (revision and redesign of ANSI X3.159-1989, American National Standard), changed by the ISO Amendment 1:1994, *Technical Corrigendum 1*, and *Technical Corrigendum 2*, contains an appendix called *Portability Issues*. The ISO appendix lists areas of the C language that ISO leaves open to each particular implementation.

Note: The MSP430 IAR C/C++ Compiler adheres to a freestanding implementation of the ISO standard for the C programming language. This means that parts of a standard library can be excluded in the implementation.

Descriptions of implementation-defined behavior

This section follows the same order as the ISO appendix. Each item covered includes references to the ISO chapter and section (in parenthesis) that explains the implementation-defined behavior.

TRANSLATION

Diagnostics (5.1.1.3)

Diagnostics are produced in the form:

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

where *filename* is the name of the source file in which the error was encountered, *linenumber* is the line number at which the compiler detected the error, *level* is the level of seriousness of the message (remark, warning, error, or fatal error), *tag* is a unique tag that identifies the message, and *message* is an explanatory message, possibly several lines.

ENVIRONMENT

Arguments to main (5.1.2.2.1)

The function called at program startup is called `main`. There is no prototype declared for `main`, and the only definition supported for `main` is:

```
int main(void)
```

To change this behavior for the IAR CLIB runtime environment, see *Customizing system initialization*, page 74. To change this behavior for the IAR DLIB runtime environment, see *Customizing system initialization*, page 52.

Interactive devices (5.1.2.3)

The streams `stdin` and `stdout` are treated as interactive devices.

IDENTIFIERS

Significant characters without external linkage (6.1.2)

The number of significant initial characters in an identifier without external linkage is 200.

Significant characters with external linkage (6.1.2)

The number of significant initial characters in an identifier with external linkage is 200.

Case distinctions are significant (6.1.2)

Identifiers with external linkage are treated as case-sensitive.

CHARACTERS

Source and execution character sets (5.2.1)

The source character set is the set of legal characters that can appear in source files. The default source character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the source character set will be the host computer's default character set.

The execution character set is the set of legal characters that can appear in the execution environment. The default execution character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the execution character set will be the host computer's default character set. The IAR DLIB Library needs a multibyte character scanner to support a multibyte execution character set. The IAR CLIB Library does not support multibyte characters.

See *Locale*, page 57.

Bits per character in execution character set (5.2.4.2.1)

The number of bits in a character is represented by the manifest constant `CHAR_BIT`. The standard include file `limits.h` defines `CHAR_BIT` as 8.

Mapping of characters (6.1.3.4)

The mapping of members of the source character set (in character and string literals) to members of the execution character set is made in a one-to-one way. In other words, the same representation value is used for each member in the character sets except for the escape sequences listed in the ISO standard.

Unrepresented character constants (6.1.3.4)

The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or in the extended character set for a wide character constant generates a diagnostic message, and will be truncated to fit the execution character set.

Character constant with more than one character (6.1.3.4)

An integer character constant that contains more than one character will be treated as an integer constant. The value will be calculated by treating the leftmost character as the most significant character, and the rightmost character as the least significant character, in an integer constant. A diagnostic message will be issued if the value cannot be represented in an integer constant.

A wide character constant that contains more than one multibyte character generates a diagnostic message.

Converting multibyte characters (6.1.3.4)

The only locale supported—that is, the only locale supplied with the IAR C/C++ Compiler—is the ‘C’ locale. If you use the command line option `--enable_multibytes`, the IAR DLIB Library will support multibyte characters if you add a locale with multibyte support or a multibyte character scanner to the library. The IAR CLIB Library does not support multibyte characters.

See *Locale*, page 57.

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 150, 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 152, 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 154, for information about `size_t`.

Conversion from/to pointers (6.3.4)

See *Casting*, page 154, for information about casting of data pointers and function pointers.

`ptrdiff_t` (6.3.6, 7.1.1)

See *ptrdiff_t*, page 154, 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 150, for information about the alignment requirement for data objects.

Sign of 'plain' bitfields (6.5.2.1)

A 'plain' `int` bitfield is treated as a signed `int` bitfield. All integer types are allowed as bitfields.

Allocation order of bitfields within a unit (6.5.2.1)

Bitfields are allocated within an integer from least-significant to most-significant bit.

Can bitfields straddle a storage-unit boundary (6.5.2.1)

Bitfields cannot straddle a storage-unit boundary for the chosen bitfield integer type.

Integer type chosen to represent enumeration types (6.5.2.2)

The chosen integer type for a specific enumeration type depends on the enumeration constants defined for the enumeration type. The chosen integer type is the smallest possible.

QUALIFIERS

Access to volatile objects (6.5.3)

Any reference to an object with volatile qualified type is an access.

DECLARATORS

Maximum numbers of declarators (6.5.4)

The number of declarators is not limited. The number is limited only by the available memory.

STATEMENTS

Maximum number of case statements (6.6.4.2)

The number of case statements (case values) in a switch statement is not limited. The number is limited only by the available memory.

PREPROCESSING DIRECTIVES

Character constants and conditional inclusion (6.8.1)

The character set used in the preprocessor directives is the same as the execution character set. The preprocessor recognizes negative character values if a 'plain' character is treated as a `signed` character.

Including bracketed filenames (6.8.2)

For file specifications enclosed in angle brackets, the preprocessor does not search directories of the parent files. A parent file is the file that contains the `#include` directive. Instead, it begins by searching for the file in the directories specified on the compiler command line.

Including quoted filenames (6.8.2)

For file specifications enclosed in quotes, the preprocessor directory search begins with the directories of the parent file, then proceeds through the directories of any grandparent files. Thus, searching begins relative to the directory containing the source file currently being processed. If there is no grandparent file and the file has not been found, the search continues as if the filename was enclosed in angle brackets.

Character sequences (6.8.2)

Preprocessor directives use the source character set, with the exception of escape sequences. Thus, to specify a path for an include file, use only one backslash:

```
#include "mydirectory\myfile"
```

Within source code, two backslashes are necessary:

```
file = fopen("mydirectory\\myfile", "rt");
```

Recognized pragma directives (6.8.6)

In addition to the pragma directives described in the chapter *Pragma directives*, the following directives are recognized but will have no effect:

```
alignment
ARGSUSED
baseaddr
basic_template_matching
can_instantiate
codeseg
cspy_support
define_type_info
do_not_instantiate
function
hdrstop
instantiate
keep_definition
memory
module_name
none
no_pch
NOTREACHED
```

```

once
__printf_args
public_equ
__scanf_args
system_include
VARARGS
warnings

```

Default `__DATE__` and `__TIME__` (6.8.8)

The definitions for `__TIME__` and `__DATE__` are always available.

IAR DLIB LIBRARY FUNCTIONS

The information in this section is valid only if the runtime library configuration you have chosen supports file descriptors. See the chapter *The DLIB runtime environment* for more information about runtime library configurations.

NULL macro (7.1.6)

The `NULL` macro is defined to 0.

Diagnostic printed by the `assert` function (7.2)

The `assert()` function prints:

```
filename:linenr expression -- assertion failed
```

when the parameter evaluates to zero.

Domain errors (7.5.1)

`NaN` (Not a Number) will be returned by the mathematic functions on domain errors.

Underflow of floating-point values sets `errno` to `ERANGE` (7.5.1)

The mathematics functions set the integer expression `errno` to `ERANGE` (a macro in `errno.h`) on underflow range errors.

`fmod()` functionality (7.5.6.4)

If the second argument to `fmod()` is zero, the function returns `NaN`; `errno` is set to `EDOM`.

`signal()` (7.7.1.1)

The signal part of the library is not supported.

Note: Low-level interface functions exist in the library, but will not perform anything. Use the template source code to implement application-specific signal handling. See *Signal and raise*, page 60.

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 56.

`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 56.

`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 56.

`%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 59.

system() (7.10.4.5)

How the command processor works depends on how you have implemented the `system` function. See *Environment interaction*, page 59.

Message returned by strerror() (7.11.6.2)

The messages returned by `strerror()` depending on the argument is:

Argument	Message
EZERO	no error
EDOM	domain error

Table 42: Message returned by `strerror()`—IAR DLIB library

Argument	Message
ERANGE	range error
EFPOS	file positioning error
EILSEQ	multi-byte encoding error
<0 >99	unknown error
all others	error <i>nnn</i>

Table 42: Message returned by `strerror()`—IAR DLIB library (Continued)

The time zone (7.12.1)

The local time zone and daylight savings time implementation is described in *Time*, page 60.

clock() (7.12.2.1)

From where the system clock starts counting depends on how you have implemented the `clock` function. See *Time*, page 60.

IAR CLIB LIBRARY FUNCTIONS

NULL macro (7.1.6)

The `NULL` macro is defined to `(void *) 0`.

Diagnostic printed by the assert function (7.2)

The `assert()` function prints:

Assertion failed: *expression*, file *Filename*, line *linenumber*

when the parameter evaluates to zero.

Domain errors (7.5.1)

`HUGE_VAL`, the largest representable value in a double floating-point type, will be returned by the mathematic functions on domain errors.

Underflow of floating-point values sets `errno` to `ERANGE` (7.5.1)

The mathematics functions set the integer expression `errno` to `ERANGE` (a macro in `errno.h`) on underflow range errors.

fmod() functionality (7.5.6.4)

If the second argument to `fmod()` is zero, the function returns zero (it does not change the integer expression `errno`).

signal() (7.7.1.1)

The signal part of the library is not supported.

Terminating newline character (7.9.2)

`stdout` stream functions recognize either `newline` or `end of file (EOF)` as the terminating character for a line.

Blank lines (7.9.2)

Space characters written to the `stdout` stream immediately before a newline character are preserved. There is no way to read the line through the `stdin` stream that was written through the `stdout` stream.

Null characters appended to data written to binary streams (7.9.2)

There are no binary streams implemented.

Files (7.9.3)

There are no other streams than `stdin` and `stdout`. This means that a file system is not implemented.

remove() (7.9.4.1)

There are no other streams than `stdin` and `stdout`. This means that a file system is not implemented.

rename() (7.9.4.2)

There are no other streams than `stdin` and `stdout`. This means that a file system is not implemented.

%p in printf() (7.9.6.1)

The argument to a `%p` conversion specifier, print pointer, to `printf()` is treated as having the type `'char *'`. The value will be printed as a hexadecimal number, similar to using the `%x` conversion specifier.

%p in scanf() (7.9.6.2)

The `%p` conversion specifier, scan pointer, to `scanf()` reads a hexadecimal number and converts it into a value with the type `'void *'`.

Reading ranges in scanf() (7.9.6.2)

A - (dash) character is always treated explicitly as a - character.

File position errors (7.9.9.1, 7.9.9.4)

There are no other streams than `stdin` and `stdout`. This means that a file system is not implemented.

Message generated by `perror()` (7.9.10.4)

`perror()` is not supported.

Allocating zero bytes of memory (7.10.3)

The `calloc()`, `malloc()`, and `realloc()` functions accept zero as an argument. Memory will be allocated, a valid pointer to that memory is returned, and the memory block can be modified later by `realloc`.

Behavior of `abort()` (7.10.4.1)

The `abort()` function does not flush stream buffers, and it does not handle files, because this is an unsupported feature.

Behavior of `exit()` (7.10.4.3)

The `exit()` function does not return.

Environment (7.10.4.4)

Environments are not supported.

`system()` (7.10.4.5)

The `system()` function is not supported.

Message returned by `strerror()` (7.11.6.2)

The messages returned by `strerror()` depending on the argument are:

Argument	Message
EZERO	no error
EDOM	domain error
ERANGE	range error
<0 >99	unknown error
all others	error No.xx

Table 43: Message returned by `strerror()`—IAR CLIB library

The time zone (7.12.1)

The time zone function is not supported.

clock() (7.12.2.1)

The `clock()` function is not supported.

A

abort
 implementation-defined behavior (CLIB) 241
 implementation-defined behavior (DLIB) 238
 system termination (DLIB) 51
 absolute location
 data, placing at (@) 34
 language support for 161
 #pragma location 188
 algorithm (STL header file) 217
 alignment 149
 forcing stricter (#pragma data_alignment) 184
 in structures (#pragma pack) 191
 in structures, causing problems 104
 of an object (__ALIGNOF__) 161
 of data types 150
 restrictions for inline assembler 79
 alignment (pragma directive) 235
 __ALIGNOF__ (operator) 161
 anonymous structures 105
 anonymous symbols, creating 163
 applications
 building, overview of 4
 startup and termination (CLIB) 73
 startup and termination (DLIB) 50
 architectures, MSP430 and MSP430X 6
 ARGFRAME (assembler directive) 89
 ARGSUSED (pragma directive) 235
 arrays
 designated initializers in 165
 hints 103
 implementation-defined behavior 233
 incomplete at end of structs 164
 non-lvalue 167
 of incomplete types 166
 single-value initialization 168
 asm, __asm (language extension) 163

assembler code
 calling from C 80
 calling from C++ 82
 inline 79
 assembler directives, using in inline assembler code 79
 assembler labels, making public (--public_equ) 143
 assembler language interface 77
 calling convention. *See* assembler code
 assembler list file, generating 134
 assembler output file 81
 assembler, inline 163
 asserts 61
 implementation-defined behavior of, (CLIB) 239
 implementation-defined behavior of, (DLIB) 236
 including in application 210
 assert.h (CLIB header file) 220
 assert.h (DLIB header file) 215
 atoll, C99 extension 219
 atomic operations 18
 __monitor 176
 attributes
 object 173
 type 171
 auto variables 12
 at function entrance 85
 programming hints for efficient code 106
 using in inline assembler code 79

B

Barr, Michael xix
 baseaddr (pragma directive) 235
 __BASE_FILE__ (predefined symbol) 208
 basic type names, using in preprocessor expressions
 (--migration_preprocessor_extensions) 136
 basic_template_matching (pragma directive) 235
 __bcd_add_long (intrinsic function) 199
 __bcd_add_long_long (intrinsic function) 199
 __bcd_add_short (intrinsic function) 199

__bic_SR_register (intrinsic function)	199
__bic_SR_register_on_exit (intrinsic function).	199
binary streams (CLIB)	240
binary streams (DLIB)	237
bis_nmi_ie1 (pragma directive)	182
__bis_SR_register (intrinsic function)	200
__bis_SR_register_on_exit (intrinsic function).	200
bit negation	108
bitfields	
data representation of	151
hints	103
implementation-defined behavior of	233
non-standard types in	161
specifying order of members (#pragma bitfields).	183
bold style, in this guide	xx
bool (data type).	150
adding support for in CLIB	221
adding support for in DLIB	215
making available in C code	219
bubble sort function, defined in stdlib.h	220
__BUILD_NUMBER__ (predefined symbol)	208

C

C and C++ linkage	83
C/C++ calling convention. <i>See</i> calling conventions	
C header files	215
call frame information	90
in assembler list file	81
in assembler list file (-IA)	135
call stack	90
callee-save registers, stored on stack.	12
calling convention	83
C++, requiring C linkage	82
calloc (library function)	13
<i>See also</i> heap	
implementation-defined behavior of (CLIB)	241
implementation-defined behavior of (DLIB)	238
can_instantiate (pragma directive)	235

cassert (DLIB header file).	217
casting	
of pointers and integers	154
operators in Extended EC++	94
operators missing from Embedded C++	94
cctype (DLIB header file)	217
cerrno (DLIB header file)	217
cexit (system termination code)	
in CLIB	73
in DLIB	50
placement in segment.	32
CFI (assembler directive)	90
cfloat (DLIB header file).	217
char (data type)	150
changing default representation (--char_is_signed)	126
signed and unsigned.	151
characters, implementation-defined behavior of	230
character-based I/O	
in CLIB	70
in DLIB	53
overriding in runtime library	47
--char_is_signed (compiler option).	126
classes.	95
CLIB.	7, 220
reference information. <i>See</i> the online help system	
summary of definitions	220
climits (DLIB header file).	217
locale (DLIB header file)	218
clock (CLIB library function), implementation-defined behavior of	242
clock (DLIB library function), implementation-defined behavior of	239
clock.c	60
__close (DLIB library function)	56
cmath (DLIB header file)	218
code	
interruption of execution	15
verifying linked result	37
code motion (compiler transformation).	102
disabling (--no_code_motion)	138

- code segments, used for placement 32
- CODE (segment) 224
 - using 32
- codeseg (pragma directive) 235
- command line options
 - part of invocation syntax 115
 - passing to compiler 116
 - See also* compiler options
- command prompt icon, in this guide xxi
- comments
 - after preprocessor directives 168
 - C++ style, using in C code 163
- common block (call frame information) 90
- common subexpr elimination (compiler transformation) . 101
 - disabling (--no_cse) 138
- compilation date
 - exact time of (__TIME__) 209
 - identifying (__DATE__) 208
- compiler
 - environment variables 116
 - invocation syntax 115
 - output from 117
- compiler listing, generating (-l) 134
- compiler object file
 - including debug information in (--debug, -r) 128
 - specifying filename of (-o) 141
- compiler optimization levels 100
- compiler options 121
 - passing to compiler 116
 - reading from file (-f) 133
 - setting 121
 - specifying parameters 123
 - summary 124
 - syntax 121
 - typographic convention xx
- compiler platform, identifying 209
- compiler subversion number 209
- compiler transformations 99
- compiler version number 210
- compiling
 - from the command line 4
 - syntax 115
- complex numbers, supported in Embedded C++ 94
- complex (library header file) 216
- compound literals 163
- computer style, typographic convention xx
- configuration
 - basic project settings 5
 - __low_level_init 52
- configuration symbols, in library configuration files 49
- consistency, module 64
- constants, placing in named segment 183
- constseg (pragma directive) 183
- const, declaring objects 157
- const_cast (cast operator) 94
- contents, of this guide xviii
- conventions, typographic xx
- copyright notice ii
- __CORE__ (predefined symbol) 208
- core
 - identifying 208
 - specifying on command line 127
- core (compiler option) 127
- __core (runtime model attribute) 65
- __cplusplus (predefined symbol) 208
- csetjmp (DLIB header file) 218
- csignal (DLIB header file) 218
- cspy_support (pragma directive) 235
- CSTACK (segment)
 - example 29
 - See also* stack
- CSTART (segment) 32, 225
- cstartup (system startup code) 32, 73
 - customizing 53
 - overriding in runtime library 47
- cstartup.s43 50
- cstdlib (DLIB header file) 218
- cstdlib (DLIB header file) 218

cstdio (DLIB header file)	218
cstdlib (DLIB header file)	218
cstring (DLIB header file)	218
ctime (DLIB header file)	218
ctype.h (library header file)	215, 220
added C functionality	218
C++	
<i>See also</i> Embedded C++ and Extended Embedded C++	
absolute location	34–35
calling convention	82
dynamic initialization in	33
features excluded from EC++	93
header files	216
language extensions	97
special function types	21
static member variables	34–35
support for	3
terminology	xx
C++ names, in assembler code	82
C++-style comments	163
C-SPY	
low-level interface	62, 74
STL container support	96
?C_EXIT (assembler label)	75
?C_GETCHAR (assembler label)	74
C_INCLUDE (environment variable)	116
?C_PUTCHAR (assembler label)	74
C99 standard, added functionality from	160, 218

D

data	
alignment of	149
located, declaring extern	34
placing	33, 184, 223
at absolute location	34
representation of	149
storage	11
verifying linked result	37

data block (call frame information)	90
data segments	27
data types	150
floating point	152
in C++	157
integers	150
dataseg (pragma directive)	184
data_alignment (pragma directive)	184
__data16 (extended keyword)	175
DATA16_AC (segment)	225
DATA16_C (segment)	225
DATA16_I (segment)	226
DATA16_ID (segment)	226
DATA16_N (segment)	226
__data16_read_addr (intrinsic function)	200
DATA16_Z (segment)	227
__data20_read_char (intrinsic function)	201
__data20_read_long (intrinsic function)	201
__data20_read_short (intrinsic function)	201
__data20_write_char (intrinsic function)	201
__data20_write_long (intrinsic function)	202
__data20_write_short (intrinsic function)	201
__DATE__ (predefined symbol)	208
date (library function), configuring support for	60
--debug (compiler option)	128
debug information, including in object file	128, 144
declarations	
empty	168
in for loops	162
Kernighan & Ritchie	108
of functions	83
declarations and statements, mixing	162
declarators, implementation-defined behavior	234
define_type_info (pragma directive)	235
delete (keyword)	13
--dependencies (compiler option)	128
deque (STL header file)	217
destructors and interrupts, using	97

- diagnostic messages 119
 - classifying as errors 129
 - classifying as remarks 129
 - classifying as warnings 130
 - disabling warnings 141
 - disabling wrapping of 141
 - enabling remarks 145
 - listing all used 130
 - suppressing 130
 - diagnostics_tables (compiler option) 130
 - diag_default (pragma directive) 185
 - diag_error (compiler option) 129
 - diag_error (pragma directive) 185
 - diag_remark (compiler option) 129
 - diag_remark (pragma directive) 185
 - diag_suppress (compiler option) 130
 - diag_suppress (pragma directive) 186
 - diag_warning (compiler option) 130
 - diag_warning (pragma directive) 186
 - DIFUNCT (segment) 33, 227
 - DINT (assembler instruction) 202
 - directives
 - function for static overlay 89
 - pragma 9, 181
 - directory, specifying as parameter 122
 - __disable_interrupt (intrinsic function) 202
 - disclaimer ii
 - DLIB 7, 214
 - building customized library 40
 - configurations 41
 - configuring 40, 131
 - debug support 41
 - reference information. *See* the online help system
 - runtime environment 39
 - dlib_config (compiler option) 131
 - Dlib_defaults.h (library configuration file) 49
 - dl430Custom.h (library configuration file) 49
 - document conventions xx
 - documentation, library 213
 - domain errors, implementation-defined behavior 236, 239
 - double (compiler option) 131
 - double (data type) 152
 - configuring size of floating-point type 7
 - __double_size (runtime model attribute) 65
 - double_t, C99 extension 219
 - do_not_instantiate (pragma directive) 235
 - dynamic initialization 50, 73
 - in C++ 33
 - dynamic memory 13
- ## E
- ec++ (compiler option) 132
 - EC++ header files 216
 - edition, of this guide ii
 - eec++ (compiler option) 132
 - EINT (assembler instruction) 202
 - Embedded C++ 93
 - differences from C++ 93
 - enabling 132
 - function linkage 83
 - language extensions 93
 - overview 93
 - Embedded C++ Technical Committee xx
 - embedded systems, IAR special support for 9
 - __embedded_cplusplus (predefined symbol) 208
 - __enable_interrupt (intrinsic function) 202
 - enable_multibytes (compiler option) 133
 - entry label, program 51
 - enumerations, implementation-defined behavior 233
 - enums
 - data representation 151
 - forward declarations of 166
 - environment
 - implementation-defined behavior 230
 - runtime
 - CLIB 69
 - DLIB 39

environment variables	
C_INCLUDE	116
QCC430	116
epilogue	189
EQU (assembler directive)	143
errno.h (library header file)	215, 220
error messages	119
classifying	129
error return codes	118
__even_in_range (intrinsic function)	202
exception handling, missing from Embedded C++	93
exception vectors	33
exception (library header file)	216
_Exit (library function)	51
exit (library function)	51
implementation-defined behavior	238, 241
_exit (library function)	51
__exit (library function)	51
export keyword, missing from Extended EC++	96
extended command line file	133
Extended Embedded C++	94
enabling	132
standard template library (STL)	217
extended keywords	171
enabling	132
overview	9
summary	174
syntax	172
__data16	175
__interrupt	16, 175
<i>See also</i> INTVEC (segment)	
using in pragma directives	195
__intrinsic	176
__monitor	109, 176
__noreturn	176
__no_init	111, 176
__raw	177
example	17
__root	178

__save_reg20	178
__task	178
__trap	
<i>See also</i> INTVEC (segment)	
extern "C" linkage	95

F

-f (compiler option)	133
fatal error messages	120
fgetpos (library function)	
implementation-defined behavior	238
field width, library support for	71
__FILE__ (predefined symbol)	208
file dependencies, tracking	128
file paths, specifying for #include files	134
file systems	240
filename	
of object file	141
specifying as parameter	122
float (data type)	152
floating point type, configuring size of double	7
floating-point constants	
hexadecimal notation	165
hints	104
floating-point expressions,	
using in preprocessor extensions	136
floating-point format	152
hints	103–104
implementation-defined behavior	232
special cases	153
32-bits	152
64-bits	153
floating-point numbers, library support for	71
float.h (library header file)	215, 220
float_t, C99 extension	219
fmod (library function),	
implementation-defined behavior	236, 239
for loops, declarations in	162

- formats
 - floating-point values 152
 - standard IEEE (floating point) 152
 - `_formatted_write` (library function) 45, 71
 - `fpclassify`, C99 extension 219
 - `FP_INFINITE`, C99 extension 219
 - `FP_NAN`, C99 extension 219
 - `FP_NORMAL`, C99 extension 219
 - `FP_SUBNORMAL`, C99 extension 219
 - `FP_ZERO`, C99 extension 219
 - fragmentation, of heap memory 13
 - `free` (library function). *See also* heap 13
 - `fstream` (library header file) 216
 - `fstream.h` (library header file) 216
 - `ftell` (library function), implementation-defined behavior . 238
 - Full DLIB (library configuration) 41
 - `__func__` (predefined symbol) 169, 208
 - `FUNCALL` (assembler directive) 89
 - `__FUNCTION__` (predefined symbol) 169, 209
 - function declarations, Kernighan & Ritchie 108
 - function directives for static overlay 89
 - function inlining (compiler transformation) 102
 - disabling (`--no_inline`) 139
 - function prototypes 107
 - enforcing 145
 - function type information, omitting in object output 142
 - `FUNCTION` (assembler directive) 89
 - function (pragma directive) 235
 - functional (STL header file) 217
 - functions 15
 - C++ and special function types 21
 - declaring 83, 107
 - executing 11
 - inlining 102, 107, 162, 187
 - interrupt 15, 18
 - intrinsic 77, 107
 - monitor 18
 - omitting type info 142
 - parameters 85
 - placing in memory 33, 35
 - recursive
 - avoiding 107
 - storing data on stack 12–13
 - reentrancy (DLIB) 214
 - related extensions 15
 - return values from 87
 - special function types 15
 - verifying linked result 37
- ## G
- `getchar` (library function) 71
 - `getenv` (library function), configuring support for 59
 - `getzone` (library function), configuring support for 60
 - `getzone.c` 60
 - `__get_interrupt_state` (intrinsic function) 203
 - `__get_R4_register` (intrinsic function) 203
 - `__get_R5_register` (intrinsic function) 203
 - `__get_SP_register` (intrinsic function) 203
 - `__get_SR_register` (intrinsic function) 204
 - `__get_SR_register_on_exit` (intrinsic function) 204
 - global variables, initialization 28
 - glossary xvii
 - Guidelines for the Use of the
C Language in Vehicle Based Software 137
 - guidelines, reading xvii
- ## H
- Harbison, Samuel P. xix
 - hardware multiplier 61
 - hardware support in compiler 40
 - `hash_map` (STL header file) 217
 - `hash_set` (STL header file) 217
 - `hdrstop` (pragma directive) 235
 - header files
 - C 215
 - C++ 216

- __interrupt (extended keyword) 16, 175
 - using in pragma directives 195
- interrupt functions 15
 - placement in memory 33
- interrupt state, restoring 205
- interrupt vector table 16
 - in linker command file 33
 - INTVEC segment 228
- interrupt vectors, specifying with pragma directive 195
- interrupts
 - disabling 176
 - during function execution 18
 - processor state 12
 - using with EC++ destructors 97
- intptr_t (integer type) 154
- __intrinsic (extended keyword) 176
- intrinsic functions 107
 - overview 77
 - summary 197
 - __bcd_add_long 199
 - __bcd_add_long_long 199
 - __bcd_add_short 199
 - __bic_SR_register 199
 - __bic_SR_register_on_exit 199
 - __bis_SR_register 200
 - __bis_SR_register_on_exit 200
 - __data16_read_addr 200
 - __data20_read_char 201
 - __data20_read_long 201
 - __data20_read_short 201
 - __data20_write_char 201
 - __data20_write_long 202
 - __data20_write_short 201
 - __disable_interrupt 202
 - __enable_interrupt 202
 - __even_in_range 202
 - __get_interrupt_state 203
 - __get_R4_register 203
 - __get_R5_register 203
 - __get_SP_register 203
 - __get_SR_register 204
 - __get_SR_register_on_exit 204
 - __low_power_mode_n 204
 - __low_power_mode_off_on_exit 204
 - __no_operation 204
 - __op_code 204
 - __set_interrupt_state 205
 - __set_R4_register 205
 - __set_R5_register 205
 - __set_SP_register 205
 - __swap_bytes 205
- intrinsics.h (header file) 197
- inttypes.h, added C functionality 218
- INTVEC (segment) 33, 228
- intwri.c (library source code) 72
- invocation syntax 115
- iomanip (library header file) 216
- iomanip.h (library header file) 216
- ios (library header file) 216
- iosfwd (library header file) 216
- iostream (library header file) 216
- iostream.h (library header file) 216
- isblank, C99 extension 218
- isfinite, C99 extension 219
- isgreater, C99 extension 219
- isinf, C99 extension 219
- isless, C99 extension 219
- islessgreater, C99 extension 219
- isless, C99 extension 219
- isnan, C99 extension 219
- isnormal, C99 extension 219
- ISO/ANSI standard
 - compiler extensions 159
 - C++ features excluded from EC++ 93
 - library compliance with 7, 213
 - specifying strict usage 147
- iso646.h (library header file) 215

ISR_CODE (segment)	228
using	32
istream (library header file)	216
isunordered, C99 extension	219
iswblank, C99 extension	220
italic style, in this guide	xx
iterator (STL header file)	217
I/O debugging, support for	62
I/O module, overriding in runtime library	47
I/O, character-based	70

K

keep_definition (pragma directive)	235
Kernighan & Ritchie function declarations	108
disallowing	145
Kernighan, Brian W.	xx
keywords, extended	9

L

-l (compiler option)	81, 134
labels	168
assembler, making public	143
__program_start	51
Labrosse, Jean J.	xx
Lajoie, Josée	xx
language extensions	
descriptions	159
Embedded C++	93
enabling	132, 187
language overview	3
language (pragma directive)	187
libraries	
CLIB	69
definition of	4
DLIB, building	40
runtime	42
standard template library	217

library configuration files	
DLIB	41
Dlib_defaults.h	49
dl430Custom.h	49
modifying	49
specifying	131
library documentation	213
library features, missing from Embedded C++	94
library functions	213
reference information	xix
summary	
CLIB	220
DLIB	215
abort (CLIB)	241
abort (DLIB)	238
calloc	238, 241
clock	239, 242
exit	238, 241
fgetpos	238
fmod	236, 239
free	13
ftell	238
getchar	71
malloc	13, 238, 241
perror	238, 241
printf	71, 237, 240
choosing formatter	45
putchar	71
realloc	13, 238, 241
remove	56, 237, 240
rename	56, 237, 240
scanf	72, 237, 240
choosing formatter	46
signal	236
sprintf	71
choosing formatter	45
sscanf	72
choosing formatter	46
strerror	238, 241

- system 238, 241
 - time zone 239, 242
 - __close 56
 - __lseek 56
 - __open 56
 - __read 56
 - __write 56
 - library header files 213
 - library modules
 - creating 135
 - overriding 47
 - library object files 214
 - library options, setting 9
 - library project template 8, 48
 - library_module (compiler option) 135
 - lightbulb icon, in this guide xxi
 - limits.h (library header file) 215, 220
 - __LINE__ (predefined symbol) 209
 - linkage, C and C++ 83
 - linker command files 24
 - customizing 25, 29, 32
 - using the -P command 26
 - using the -Z command 26
 - linker map file 37
 - linker segment. *See* segment
 - linking
 - from the command line 5
 - required input 4
 - Lippman, Stanley B. xx
 - list (STL header file) 217
 - listing, generating 134
 - literals, compound 163
 - literature, recommended xix
 - llabs, C99 extension 219
 - lldiv, C99 extension 219
 - local variables, *See* auto variables
 - locale support
 - DLIB 57
 - adding 58
 - changing at runtime 58
 - removing 58
 - locale.h (library header file) 215
 - located data segments 31
 - located data, declaring extern 34
 - location (pragma directive) 34, 188
 - LOCFRAME (assembler directive) 89
 - lock_r4 (compiler option) 136
 - lock_r5 (compiler option) 136
 - long double (data type) 152
 - long float (data type), synonym for double 167
 - long long (data type), restrictions 151
 - loop overhead, reducing 140
 - loop unrolling (compiler transformation) 101
 - disabling 140
 - loop-invariant expressions 102
 - low-level processor operations 160, 197
 - accessing 77
 - __low_level_init 51
 - customizing 52
 - low_level_init.c 50, 73
 - __low_power_mode_n (intrinsic function) 204
 - __low_power_mode_off_on_exit (intrinsic function) 204
 - __lseek (library function) 56
- ## M
- macros, variadic 211
 - main (function), definition 230
 - malloc (library function)
 - implementation-defined behavior 238, 241
 - malloc (library function). *See also* heap 13
 - Mann, Bernhard xx
 - map (STL header file) 217
 - map, linker 37
 - math.h (library header file) 215, 220
 - math.h, added C functionality 218
 - MATH_ERREXCEPT, C99 extension 219
 - math_errhandling, C99 extension 219

MATH_ERRNO, C99 extension	219
_medium_write (library function)	71
memory	
allocating in C++	13
dynamic	13
heap	13
non-initialized	110
RAM, saving	107
releasing in C++	13
stack	12
saving	107
static	11
used by executing functions	11
used by global or static variables	11
memory consumption, reducing	71
memory management, type-safe	93
memory placement, using type definitions	172
memory segment. <i>See</i> segment	
memory (pragma directive)	235
memory (STL header file)	217
message (pragma directive)	188
messages	
disabling	146
forcing	188
--migration_preprocessor_extensions (compiler option)	136
MISRA C rules	
checking for adherence to	137
logging	137
--misrac (compiler option)	137
--misrac_verbose (compiler option)	137
module consistency	64
rtmodel	193
module map, in linker map file	37
module name, specifying	138
module summary, in linker map file	37
--module_name (compiler option)	138
module_name (pragma directive)	235
__monitor (extended keyword)	109, 176
monitor functions	18, 176

MSP430 architecture	6
MSP430X architecture	6
multibyte character support	133
multiple inheritance, missing from Embedded C++	93
mutable attribute	
in Extended EC++	94, 96
missing from Embedded C++	94

N

names block (call frame information)	90
namespace support	
in Extended EC++	94, 96
missing from Embedded C++	94
NAN, C99 extension	219
NDEBUG (preprocessor symbol)	210
new (keyword)	13
new (library header file)	216
new.h (library header file)	216
none (pragma directive)	235
non-initialized variables, hints for	111
non-scalar parameters, avoiding	107
NOP (assembler instruction)	204
__noreturn (extended keyword)	176
Normal DLIB (library configuration)	41
Not a number (NaN)	153
NOTREACHED (pragma directive)	235
--no_code_motion (compiler option)	138
--no_cse (compiler option)	138
no_epilogue (pragma directive)	189
__no_init (extended keyword)	111, 176
--no_inline (compiler option)	139
__no_operation (intrinsic function)	204
no_pch (pragma directive)	235
--no_typedefs_in_diagnostics (compiler option)	139
--no_unroll (compiler option)	140
--no_warnings (compiler option)	141
--no_wrap_diagnostics (compiler option)	141
NULL	221

NULL (macro), implementation-defined behavior . . . 236, 239
 numeric (STL header file) 217

O

-o (compiler option) 141
 object attributes 173
 object filename, specifying 141
 object module name, specifying 138
 object_attribute (pragma directive) 111, 189
 offsetof 221
 --omit_types (compiler option) 142
 once (pragma directive) 236
 --only_stdout (compiler option) 142
 __open (library function) 56
 operators
 @ 34, 161
 & 107
 ? 98
 _Pragma 211
 __ALIGNOF__ 161
 __segment_begin 161
 __segment_end 161
 optimization
 code motion, disabling 138
 common sub-expression elimination, disabling 138
 configuration 7
 disabling 101
 function inlining, disabling 139
 hints 106
 loop unrolling, disabling 140
 size, specifying 148
 speed, specifying 145
 summary 100
 techniques 101
 types and levels 100
 type-based alias analysis 102
 type-based alias analysis (compiler option)
 disabling 139

 using inline assembler code 79
 using pragma directive 189
 optimize (pragma directive) 189
 option parameters 121
 options, compiler. *See* compiler options
 __op_code (intrinsic function) 204
 Oram, Andy xix
 ostream (library header file) 216
 output files, from XLINK 5
 output (linker)
 specifying 5
 specifying file name 5
 output (preprocessor) 143
 output, supporting non-standard 72
 overhead, reducing 101–102

P

pack (pragma directive) 155, 190
 packed structure types 155
 parameters
 function 85
 hidden 85
 non-scalar, avoiding 107
 register 85
 rules for specifying a file or directory 122
 specifying 123
 stack 85–86
 typographic convention xx
 part number, of this guide ii
 permanent registers 84
 perror (library function),
 implementation-defined behavior 238, 241
 --pic (compiler option) 143
 placement
 code and data 223
 in named segments 35
 pointers
 casting 154

implementation-defined behavior	233
mixing types of	167
polymorphism, in Embedded C++	93
porting, code containing pragma directives	182
position independent code	6
_Pragma (predefined symbol)	210
pragma directives	9
summary	181
alignment	235
ARGSUSED	235
baseaddr	235
basic_template_matching	235
bis_nmi_ie1	182
bitfields	152, 183
can_instantiate	235
codeseg	235
constseg	183
cspy_support	235
dataseg	184
data_alignment	184
define_type_info	235
diag_default	185
diag_error	185
diag_remark	185
diag_suppress	186
diag_warning	186
do_not_instantiate	235
function	235
hdrstop	235
include_alias	186
inline	187
instantiate	235
keep_definition	235
language	187
location	34, 188
memory	235
message	188
module_name	235
none	235

NOTREACHED	235
no_epilogue	189
no_pch	235
object_attribute	111, 189
once	236
optimize	189
pack	155, 190
public_equ	236
required	192
rtmodel	193
segment	194
system_include	236
type_attribute	194
VARARGS	236
vector	16, 195
warnings	236
__printf_args	236
__scanf_args	236
precision arguments, library support for	71
predefined symbols	
overview	10
summary	208
_Pragma	210
__BASE_FILE__	208
__BUILD_NUMBER__	208
__CORE__	208
__cplusplus	208
__DATE__	208
__embedded_cplusplus	208
__FILE__	208
__FUNCTION__	169, 209
__func__	169, 208
__IAR_SYSTEMS_ICC__	209
__ICC430__	209
__LINE__	209
__PRETTY_FUNCTION__	209
__STDC_VERSION__	209
__STDC__	209
__SUBVERSION__	209

__TIME__ 209
 __VER__ 210
 --preinclude (compiler option) 142
 --preprocess (compiler option) 143
 preprocessing directives, implementation-defined
 behavior 234
 preprocessor
 output 143
 overview 207
 preprocessor extensions
 compatibility 136
 #warning message 211
 __VA_ARGS__ 211
 preprocessor symbols 208
 defining 127
 NDEBUG 210
 preserved registers 84
 __PRETTY_FUNCTION__ (predefined symbol). 209
 primitives, for special functions 15
 print formatter, selecting 45
 printf (library function). 45, 71
 choosing formatter 45
 configuration symbols 55
 customizing 72
 implementation-defined behavior. 237, 240
 selecting. 72
 processor operations
 accessing 77
 low-level 160, 197
 program entry label. 51
 programming hints 106
 __program_start (label). 51
 prototypes, enforcing 145
 ptrdiff_t (integer type). 154, 221
 PUBLIC (assembler directive) 143
 publication date, of this guide ii
 --public_equ (compiler option) 143
 public_equ (pragma directive) 236
 putchar (library function) 71
 putenv (library function), absent from DLIB 59

Q

QCC430 (environment variable) 116
 qualifiers, implementation-defined behavior. 234
 queue (STL header file) 217

R

-r (compiler option). 144
 raise (library function), configuring support for 60
 raise.c 60
 RAM memory, saving. 107
 range errors, in linker 37
 __raw (extended keyword) 177
 example 17
 __read (library function). 56
 read formatter, selecting 46, 73
 __read (library function)
 customizing 53
 reading guidelines. xvii
 reading, recommended xix
 realloc (library function)
 implementation-defined behavior. 238, 241
 realloc (library function). *See also* heap 13
 recursive functions
 avoiding 107
 storing data on stack 12–13
 --reduce_stack_usage (compiler option) 144
 reentrancy (DLIB). 214
 reference information, typographic convention. xx
 register parameters 85
 registered trademarks ii
 registers
 assigning to parameters 86
 callee-save, stored on stack 12
 for function returns 87
 implementation-defined behavior. 233
 in assembler-level routines. 83
 preserved 84

R4	
excluding from use (--lock_R4)	136
getting the value of (__get_R4_register)	203
reserving for register variables (--regvar_R4)	144
writing to (__set_R4_register)	205
R5	
excluding from use (--lock_R5)	136
getting the value of (__get_R5_register)	203
reserving for register variables (--regvar_R5)	144
writing to (__set_R5_register)	205
scratch	84
SP	
getting the value of (__get_SP_register)	203
writing to (__set_SP_register)	205
SR	
getting the value of on exit	204
getting the value of (__get_SR_register)	204
REGVAR_AC (segment)	228
--regvar_r4 (compiler option)	144
--regvar_r5 (compiler option)	144
__reg_4 (runtime model attribute)	65
__reg_5 (runtime model attribute)	65
reinterpret_cast (cast operator)	94
remark (diagnostic message)	
classifying	129
enabling	145
--remarks (compiler option)	145
remarks (diagnostic message)	119
remove (library function)	56
implementation-defined behavior	237, 240
rename (library function)	56
implementation-defined behavior	237, 240
__ReportAssert (library function)	61
required (pragma directive)	192
--require_prototypes (compiler option)	145
RESET (segment)	228
return values, from functions	87
Ritchie, Dennis M.	xx
__root (extended keyword)	178

routines, time-critical	77, 160, 197
RTMODEL (assembler directive)	65
rtmodel (pragma directive)	193
rtti support, missing from STL	94
__rt_version (runtime model attribute)	66
runtime environment	
CLIB	69
DLIB	39
setting options	9
runtime libraries	
choosing	8
introduction	213
CLIB	69
naming convention	70
DLIB	42
choosing	43
customizing without rebuilding	44
naming convention	43
overriding modules in	47
runtime model attributes	64
__core	65
__double_size	65
__reg_4	65
__reg_5	65
__rt_version	66
runtime model definitions	193
runtime type information, missing from Embedded C++	93
R4 <i>See</i> registers	
R5 <i>See</i> registers	

S

-s (compiler option)	145
--save_reg20 (compiler option)	146
__save_reg20 (extended keyword)	178
scanf (library function)	72
choosing formatter	46
configuration symbols	55
implementation-defined behavior	237, 240

- scratch registers 84
- segment base name 27
- segment map, in linker map file 37
- segment memory types, in XLINK 24
- segment names
 - declaring 194
- segment operators 161
- segment (pragma directive). 194
- segments 223
 - code 32
 - data 27
 - definition of 23
 - initialized data 28
 - introduction 23
 - located data 31
 - naming 27
 - packing in memory 26
 - placing in sequence 26
 - static memory 27
 - summary 223
 - too long for address range 37
 - too long, in linker. 37
- CODE 32, 224
- CSTACK
 - example. 29
- CSTART 32, 225
- DATA16_AC 225
- DATA16_AN 225
- DATA16_C 225
- DATA16_I 226
- DATA16_ID 226
- DATA16_N 226
- DATA16_Z. 227
- DIFUNCT 33, 227
- HEAP. 30, 227
- INTVEC 33, 228
- ISR_CODE 32, 228
- REGVAR_AN 228
- RESET. 228
- __segment_begin (extended operator). 161
- __segment_end (extended operator) 161
- semaphores
 - C example 18
 - C++ example 19
 - operations on 176
- set (STL header file) 217
- setjmp.h (library header file). 215, 221
- setlocale (library function) 58
- settings, basic for project configuration 5
 - __set_interrupt_state (intrinsic function) 205
 - __set_R4_register (intrinsic function). 205
 - __set_R5_register (intrinsic function). 205
 - __set_SP_register (intrinsic function). 205
- severity level, of diagnostic messages 119
 - specifying 120
- SFR (special function registers) 109
 - declaring extern 34
- shared object. 118
- signal (library function)
 - configuring support for 60
 - implementation-defined behavior. 236
- signal.c 60
- signal.h (library header file) 215
- signbit, C99 extension. 219
- signed char (data type) 150–151
 - specifying 126
- signed int (data type). 150
- signed long long (data type) 150
- signed long (data type) 150
- signed short (data type). 150
- silent (compiler option) 146
- silent operation, specifying 146
- 64-bits (floating-point format) 153
- size optimization, specifying. 148
- sizeof, using in preprocessor extensions 136
- size_t (integer type) 154, 221
- skeleton code, creating for assembler language interface . . 80
- slist (STL header file) 217

<code>_small_write</code> (library function)	71
<code>snprintf</code> , C99 extension	219
source files, list all referred	134
SP <i>See</i> registers	
special function registers (SFR)	109
special function types	15
overview	10
speed optimization, specifying	145
<code>sprintf</code> (library function)	45, 71
choosing formatter	45
customizing	72
SR <i>See</i> registers	
<code>sscanf</code> (library function)	72
choosing formatter	46
<code>sstream</code> (library header file)	216
stack	12, 29
advantages and problems using	12
changing default size of	29
cleaning after function return	87
contents of	12
function usage	11
internal data	224
layout	86
saving space	107
size	30
stack parameters	85–86
stack pointer	12
stack segment, placing in memory	30
stack (STL header file)	217
standard error	142
standard input	53
standard output	53
specifying	142
standard template library (STL)	
in Extended EC++	94, 96, 217
missing from Embedded C++	94
startup code	
placement of	32
<i>See also</i> CSTART	

startup, system	
CLIB	73
DLIB	51
statements, implementation-defined behavior	234
static data, in linker command file	29
static memory	11
static memory segments	27
static overlay	89
static variables	
initialization	28
taking the address of	107
<code>static_cast</code> (cast operator)	94
std namespace, missing from EC++	
and Extended EC++	97
<code>stdarg.h</code> (library header file)	215, 221
<code>stdbool.h</code> (library header file)	150, 215, 221
added C functionality	219
<code>__STDC__</code> (predefined symbol)	209
<code>__STDC_VERSION__</code> (predefined symbol)	209
<code>stddef.h</code> (library header file)	151, 215, 221
<code>stderr</code>	56, 142
<code>stdexcept</code> (library header file)	216
<code>stdin</code>	56
implementation-defined behavior	237, 240
<code>stdint.h</code> , added C functionality	219
<code>stdio.h</code> (library header file)	215, 221
<code>stdio.h</code> , additional C functionality	219
<code>stdlib.h</code> (library header file)	215, 221
<code>stdlib.h</code> , additional C functionality	219
<code>stdout</code>	56, 142
implementation-defined behavior	237, 240
Steele, Guy L.	xix
STL	96
<code>streambuf</code> (library header file)	216
streams, supported in Embedded C++	94
<code>strerror</code> (library function)	
implementation-defined behavior	238, 241
<code>--strict_ansi</code> (compiler option)	147
<code>string</code> (library header file)	216
strings, supported in Embedded C++	94

- string.h (library header file) 215, 221
 - Stroustrup, Bjarne xx
 - strstream (library header file) 216
 - strtod (library function), configuring support for 61
 - strtod, in stdlib.h 220
 - strtof, C99 extension 219
 - strtold, C99 extension 219
 - strtoll, C99 extension 219
 - strtoull, C99 extension 219
 - structs 164
 - anonymous 161
 - structure types
 - alignment 155
 - layout of 155
 - packed 155
 - structures
 - aligning 191
 - anonymous 105
 - implementation-defined behavior 233
 - packing and unpacking 105
 - __SUBVERSION__ (predefined symbol) 209
 - support, technical 120
 - SWAPB (assembler instruction) 205
 - __swap_bytes (intrinsic function) 205
 - switch statements, hints for using 111
 - symbol names, using in preprocessor extensions 136
 - symbols
 - anonymous, creating 163
 - including in output 192
 - listing in linker map file 37
 - overview of predefined 10
 - preprocessor, defining 127
 - syntax
 - compiler invocation 115
 - compiler options 121
 - extended keywords 172
 - system startup
 - CLIB 73
 - customizing 52
 - DLIB 51
 - system termination
 - CLIB 73
 - DLIB 51
 - C-SPY interface 52
 - system (library function)
 - configuring support for 59
 - implementation-defined behavior 238, 241
 - system_include (pragma directive) 236
- ## T
- __task (extended keyword) 178
 - technical support, IAR Systems 120
 - template support
 - in Embedded C++ 93
 - in Extended EC++ 94, 96
 - Terminal I/O window 74
 - making available 63
 - terminal output, speeding up 63
 - termination, of system
 - CLIB 73
 - DLIB 51
 - terminology xvii, xx
 - 32-bits (floating-point format) 152
 - this (pointer) 82
 - __TIME__ (predefined symbol) 209
 - time zone (library function)
 - implementation-defined behavior 239, 242
 - time (library function), configuring support for 60
 - time-critical routines 77, 160, 197
 - time.c 60
 - time.h (library header file) 215
 - tips, programming 106
 - tools icon, in this guide xx
 - trademarks ii
 - transformations, compiler 99
 - translation, implementation-defined behavior 229
 - trap vectors, specifying with pragma directive 195

type attributes	171
specifying	194
type definitions, used for specifying memory storage	172
type information, omitting	142
type qualifiers, const and volatile	156
typedefs	
excluding from diagnostics	139
repeated	167
using in preprocessor extensions	136
type-based alias analysis (compiler transformation)	102
disabling	139
type-safe memory management	93
type_attribute (pragma directive)	194
typographic conventions	xx

U

uintptr_t (integer type)	154
underflow range errors, implementation-defined behavior	236, 239
unions	
anonymous	105, 161
implementation-defined behavior	233
unsigned char (data type)	150–151
changing to signed char	126
unsigned int (data type)	150
unsigned long long (data type)	150
unsigned long (data type)	150
unsigned short (data type)	150
utility (STL header file)	217

V

VARARGS (pragma directive)	236
variable type information, omitting in object output	142
variables	
auto	12
defined inside a function	12
global, placement in memory	11

hints for choosing	106
local, <i>See</i> auto variables	
non-initialized	111
omitting type info	142
placing at absolute addresses	35
placing in named segments	35
static	
placement in memory	11
taking the address of	107
static and global, initializing	28
vector (pragma directive)	16, 195
vector (STL header file)	217
__VER__ (predefined symbol)	210
version, IAR Embedded Workbench	ii
version, of compiler	209–210
vfscanf, C99 extension	219
vfwscanf, C99 extension	220
void, pointers to	166
volatile (keyword)	109
volatile, declaring objects	156
vscanf, C99 extension	219
vsprintf, C99 extension	219
vsscanf, C99 extension	219
vswscanf, C99 extension	220
vwsscanf, C99 extension	220

W

#warning message (preprocessor extension)	211
warnings	119
classifying	130
disabling	141
exit code	147
warnings (pragma directive)	236
--warnings_affect_exit_code (compiler option)	118
--warnings_are_errors (compiler option)	147
wchar.h (library header file)	215
wchar.h, added C functionality	220
wchar_t (data type), adding support for in C	151

wcstof, C99 extension	220
wcstolb, C99 extension	220
wctype.h (library header file)	215
wctype.h, added C functionality	220
web sites, recommended	xx
write formatter, selecting	72–73
__write (library function)	56
customizing	53

X

XLINK errors	
range error	37
segment too long	37
XLINK output files	5
XLINK segment memory types	24
xreportassert.c	61

Z

-z (compiler option)	148
----------------------	-----

Symbols

#include files, specifying	116, 134
#warning message (preprocessor extension)	211
-D (compiler option)	127
-e (compiler option)	132
-f (compiler option)	133
-I (compiler option)	134
-l (compiler option)	81, 134
-o (compiler option)	141
-r (compiler option)	144
-s (compiler option)	145
-z (compiler option)	148
--char_is_signed (compiler option)	126
--core (compiler option)	127
--debug (compiler option)	128
--dependencies (compiler option)	128

--diagnostics_tables (compiler option)	130
--diag_error (compiler option)	129
--diag_remark (compiler option)	129
--diag_suppress (compiler option)	130
--diag_warning (compiler option)	130
--dlib_config (compiler option)	131
--double (compiler option)	131
--ec++ (compiler option)	132
--eec++ (compiler option)	132
--enable_multibytes (compiler option)	133
--error_limit (compiler option)	133
--header_context (compiler option)	134
--library_module (compiler option)	135
--lock_r4 (compiler option)	136
--lock_r5 (compiler option)	136
--migration_preprocessor_extensions (compiler option)	136
--misrac (compiler option)	137
--misrac_verbose (compiler option)	137
--module_name (compiler option)	138
--no_code_motion (compiler option)	138
--no_cse (compiler option)	138
--no_inline (compiler option)	139
--no_tbaa (compiler option)	139
--no_typedefs_in_diagnostics (compiler option)	139
--no_unroll (compiler option)	140
--no_warnings (compiler option)	141
--no_wrap_diagnostics (compiler option)	141
--omit_types (compiler option)	142
--only_stdout (compiler option)	142
--pic (compiler option)	143
--preinclude (compiler option)	142
--preprocess (compiler option)	143
--reduce_stack_usage (compiler option)	144
--regvar_r4 (compiler option)	144
--regvar_r5 (compiler option)	144
--remarks (compiler option)	145
--require_prototypes (compiler option)	145
--save_reg20 (compiler option)	146
--silent (compiler option)	146

--strict_ansi (compiler option)	147	__disable_interrupt (intrinsic function)	202
--warnings_affect_exit_code (compiler option)	118, 147	__double_size (runtime model attribute)	65
--warnings_are_errors (compiler option)	147	__embedded_cplusplus (predefined symbol)	208
?C_EXIT (assembler label)	75	__enable_interrupt (intrinsic function)	202
?C_GETCHAR (assembler label)	74	__even_in_range (intrinsic function)	202
?C_PUTCHAR (assembler label)	74	__exit (library function)	51
@ (operator)	34, 161	__FILE__ (predefined symbol)	208
__write_buffered (DLIB library function)	63	__FUNCTION__ (predefined symbol)	169, 209
_Exit (library function)	51	__func__ (predefined symbol)	169, 208
_exit (library function)	51	__gets, in stdio.h.	219
_Exit, C99 extension	219	__get_interrupt_state (intrinsic function)	203
_formatted_write (library function)	45, 71	__get_R4_register (intrinsic function)	203
_medium_write (library function)	71	__get_R5_register (intrinsic function)	203
_Pragma (predefined symbol)	210	__get_SP_register (intrinsic function)	203
_small_write (library function)	71	__get_SR_register (intrinsic function)	204
__ALIGNOF__ (operator)	161	__get_SR_register_on_exit (intrinsic function)	204
__asm (language extension)	163	__IAR_SYSTEMS_ICC__ (predefined symbol)	209
__BASE_FILE__ (predefined symbol)	208	__ICC430__ (predefined symbol)	209
__bcd_add_long (intrinsic function)	199	__interrupt (extended keyword)	16, 175
__bcd_add_long_long (intrinsic function)	199	using in pragma directives	195
__bcd_add_short (intrinsic function)	199	__intrinsic (extended keyword)	176
__bic_SR_register (intrinsic function)	199	__LINE__ (predefined symbol)	209
__bic_SR_register_on_exit (intrinsic function)	199	__low_level_init	51
__bis_SR_register (intrinsic function)	200	low_level_init, customizing	52
__bis_SR_register_on_exit (intrinsic function)	200	__low_power_mode_n (intrinsic function)	204
__BUILD_NUMBER__ (predefined symbol)	208	__low_power_mode_off_on_exit (intrinsic function)	204
__close (library function)	56	__lseek (library function)	56
__core (runtime model attribute)	65	__monitor (extended keyword)	109, 176
__CORE__ (predefined symbol)	208	__noreturn (extended keyword)	176
__cplusplus (predefined symbol)	208	__no_init (extended keyword)	111, 176
__data16 (extended keyword)	175	__no_operation (intrinsic function)	204
__data16_read_addr (intrinsic function)	200	__open (library function)	56
__data20_read_char (intrinsic function)	201	__op_code (intrinsic function)	204
__data20_read_long (intrinsic function)	201	__PRETTY_FUNCTION__ (predefined symbol)	209
__data20_read_short (intrinsic function)	201	__printf_args (pragma directive)	236
__data20_write_char (intrinsic function)	201	__program_start (label)	51
__data20_write_long (intrinsic function)	202	__qsortbbl, C99 extension	220
__data20_write_short (intrinsic function)	201	__raw (extended keyword)	177
__DATE__ (predefined symbol)	208	example	17

__read (library function)	56
customizing	53
__reg_4 (runtime model attribute)	65
__reg_5 (runtime model attribute)	65
__ReportAssert (library function)	61
__root (extended keyword)	178
__rt_version (runtime model attribute)	66
__save_reg20 (extended keyword)	178
__scanf_args (pragma directive)	236
__segment_begin	161
__segment_end (extended operators)	161
__set_interrupt_state (intrinsic function)	205
__set_R4_register (intrinsic function)	205
__set_R5_register (intrinsic function)	205
__set_SP_register (intrinsic function)	205
__STDC_VERSION__ (predefined symbol)	209
__STDC__ (predefined symbol)	209
__SUBVERSION__ (predefined symbol)	209
__swap_bytes (intrinsic function)	205
__task (extended keyword)	178
__TIME__ (predefined symbol)	209
__ungetchar, in stdio.h	219
__VA_ARGS__ (preprocessor extension)	211
__VER__ (predefined symbol)	210
__write (library function)	56
__write (library function), customizing	53
__write_array, in stdio.h	219

Numerics

32-bits (floating-point format)	152
64-bits (floating-point format)	153