dsPIC IAR C/EC++ Compiler

Reference Guide

for Microchip's

dsPIC Microcontroller Family

COPYRIGHT NOTICE

© Copyright 2002 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, IAR Embedded Workbench, IAR XLINK Linker, IAR XAR Library Builder, IAR XLIB Librarian, IAR MakeApp, and IAR PreQual are trademarks owned by IAR Systems. C-SPY is a trademark registered in Sweden by IAR Systems. IAR visualSTATE is a registered trademark owned by IAR Systems.

dsPIC and Microchip are registered trademarks of Microchip Corporation.

Microsoft and Windows are registered trademarks of Microsoft Corporation. Adobe and Acrobat Reader are registered trademarks of Adobe Systems Incorporated. CodeWright is a registered trademark of Starbase Corporation.

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

EDITION NOTICE

First edition: April 2002

Part number: CDSPIC-1

Contents

Tables xi

Preface xiii

Who should read this guide xiii

How to use this guide xiii

What this guide contains xiii

Other documentation xv

Further reading xv

Document conventions xv

Typographic conventions xvi

Part 1: Using the compiler 1

Introduction 3

Building applications 3

Compiling 3

Linking 3

Data storage 4

Optimization techniques 4

IAR language extension overview 4

Special function types 5

Extended keywords 5

#pragma directives 5

Predefined symbols 5

Intrinsic functions 5

Inline assembler 6

Runtime libraries 6
Embedded C++ overview 6
Customization 9
Processor variant 9
Data model 9
Runtime libraries 10
Data storage 13
Stack, static, and heap memory 13
The stack and auto variables 13
Static memory 15
Dynamic memory on the heap 15
Memory access methods and memory types 16
Memory access methods 16
Memory types 17
Pointers 17
Pointers and memory types 18
Structure types and memory types 19
Embedded C++ and memory types 19
Non-initialized memory 20
Located variables 21
Absolute location placement 21
Segment placement 21
Accessing special function registers 22
Anonymous structs and unions 23
Functions 25
Special function types 25
Interrupt functions 25
Segment placement 26
Assembler language interface 27
Introduction 27
Example of assembler function 27

Calling convention ... 30
Function declarations ... 30
C and C++ linkage ... 30
Function parameters ... 31
Returning a value from a function ... 32
Permanent versus scratch registers ... 32
Examples ... 33
Monitor functions ... 33
Calling functions ... 34
Assembler instructions used for calling functions ... 34
Special function types ... 34
Runtime model attributes ... 35
Specifying runtime attributes ... 35
Predefined runtime attributes ... 36
Calling assembler routines from C ... 36
Creating skeleton code ... 37
Calling assembler routines from Embedded C++ ... 40
Function directives ... 41
Syntax ... 41
Parameters ... 41
Description ... 42
Segments and memory ... 43
What is a segment? ... 43
Linker segment type ... 43
Placeholder segments ... 44
Placing segments in memory ... 44
The contents of the linker command file ... 44
Customizing a linker command file ... 45
Data segments ... 46
Static memory segments ... 46
The heap ... 50
Located data ... 51

Code segments 51
Startup code 51
Normal code 51
Exception vectors 51
Embedded C++ dynamic initialization 51
Runtime environment 53
The cstartup.s59 file 53
System startup 53
System termination 53
__low_level_init 54
Customizing cstartup.s59 54
Modules and segment parts 55
Call frame information 56
Modifying the cstartup.s59 file 56
Input and output 57
The IAR CLIB library 57
The IAR DLIB library 60
C-SPY debugger interface 62
The debugger terminal I/O window 62
Efficient coding techniques 63
Programming hints 63
Optimizing for size or speed 63
Saving stack space and RAM memory 64
Using efficient data types 64
Module compatibility 64
Part 2: Compiler reference 65
Data representation 67
Fundamentals 67
Alignment 67
Byte order 67

Data types 67
Integer types 67
Floating-point types 68
Pointers 70
Size 70
Casting 70
Structure types 70
Alignment 70
General layout 70
Data types in Embedded C++ 71
Segment reference 73
Summary of segments 73
Descriptions of segments 74
Compiler options 83
Setting compiler options 83
Specifying parameters 83
Specifying environment variables 84
Error return codes 85
Options summary 85
Descriptions of options 87
Extended keywords 103
Summary of extended keywords 103
Using extended keywords 103
Data storage 104
Functions 105

Descriptions of extended keywords 105

#pragma directives 109

Summary of #pragma directives 109

Descriptions of #pragma directives 110

Predefined symbols 117

Summary of predefined symbols 117

Descriptions of predefined symbols 118

Intrinsic functions 121

Intrinsic functions summary 121

DSP-related intrinsic functions 121

General intrinsic functions 124

Descriptions of intrinsic functions 124

Library functions 133

IAR CLIB library 133

Library object files 133

Header files 133

Library definitions summary 134

IAR DLIB library 134

Library object files 135

Header files 135

Library definitions summary 135

Diagnostics 139

Severity levels 139

Setting the severity level 140

Internal error 140

Part 3: Portability 141

Implementation-defined behavior 143

Descriptions of implementation-defined behavior 143

Translation 143

Environment 144
Identifiers 144
Characters 144
Integers 145
Floating point 146
Arrays and pointers 147
Registers 147
Structures, unions, enumerations, and bitfields 147
Qualifiers 148
Declarators 148
Statements 148
Preprocessing directives 148
IAR CLIB library functions 150
IAR DLIB library functions 153

IAR C extensions 157

Why should language extensions be used? 157
Descriptions of language extensions 157

Index 167

Tables

1: Typographic conventions used in this guide xvi
2: Mapping of processor options 9
3: Data model characteristics 10
4: Runtime libraries 11
5: Memory types 17
6: Example of runtime model attributes 35
7: Runtime model attributes 36
8: XLINK segment types 44
9: Linker command file example 45
10: Memory types 47
11: Segment groups 47
12: Segments in segment groups 47
13: Stack types 50
14: I/O files 61
15: Integer types 67
16: Floating-point types 68
17: Segment summary 73
18: Environment variables 85
19: Error return codes 85
20: Compiler options summary 85
21: Available data models 88
22: Generating a compiler list file (-l) 93
23: Directing preprocessor output to file (--preprocess) 97
24: Specifying speed optimization (-s) 99
25: Mapping of processor options 99
26: Specifying size optimization (-z) 100
27: Extended keywords summary 103
28: #pragma directives summary 109
29: Predefined symbols summary 117
30: Inspecting the data model using predefined symbols 118
31: DSP-related intrinsic functions summary 121

32: General intrinsic functions summary ... 124
33: IAR C Library header files... 134
34: Miscellaneous IAR C Library header files ... 134
35: Traditional standard C library header files ... 135
36: Embedded C++ library header files ... 136
37: New standard C library header files... 137
38: Traditional C++ library header files ... 138
39: Message returned by strerror()—IAR CLIB library ... 152
40: Message returned by strerror()—IAR DLIB library ... 156

Preface

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

- The architecture and instruction set of the dsPIC microcontroller. Refer to the documentation from Microchip for information about the dsPIC microcontroller
- The C or Embedded C++ programming language
- The operating system of your host machine.

How to use this guide

When you first begin using dsPIC IAR C/EC++ Compiler, you should read *Part 1: Using the compiler* in this reference guide.

When you are thoroughly familiar with the dsPIC IAR C/EC++ Compiler and have already configured your project, you can focus more on *Part 2: Compiler reference*.

If you are new to using the IAR toolkit, we recommend that you first read the initial chapters of the *dsPIC IAR Embedded Workbench™ IDE User Guide*. They include comprehensive information about the installation of all IAR tools and give product overviews, as well as tutorials that can help you get started.

What this guide contains

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

Part 1: Using the compiler

- *Introduction* gives an overview of the compiler techniques that allow an application to take full advantage of the dsPIC microcontroller: code and data storage features, optimization techniques, and language extensions.

- *Customization* describes the available customization options: processor option, data model, and runtime libraries.
- *Data storage* describes how data can be stored in memory, with an emphasis on the different memory types.
- *Functions* describes the different ways code can be generated. Interrupt functions are also covered.
- *Assembler language interface* contains information about memory access methods, how parameters are passed in registers and the C and Embedded C++ calling conventions. Runtime model attributes are also described here.
- *Segments and memory* describes the concept of segments, introduces the linker command file, and describes how code and data are placed in memory.
- *Runtime environment* describes system initialization, introduces the `cstartup` file, and describes some low-level I/O routines in the runtime library.
- *Efficient coding techniques* gives hints about programming for the dsPIC IAR C/EC++ Compiler.

Part 2: Compiler reference

- *Data representation* describes the available data types, pointers, and structure types.
- *Segment reference* gives reference information about the compiler's use of segments.
- *Compiler options* explains how to set the compiler options, gives a summary of the options, and contains detailed reference information for each compiler option.
- *Extended keywords* gives reference information about each of the dsPIC-specific keywords that are extensions to the standard C language.
- *#pragma directives* gives reference information about the `#pragma` directives.
- *Predefined symbols* gives reference information about the predefined preprocessor symbols.
- *Intrinsic functions* gives reference information about the functions that can use dsPIC-specific low-level features.
- *Library functions* gives an introduction to the C or Embedded C++ library functions, and summarizes the header files.
- *Diagnostics* describes how the dsPIC IAR C/EC++ Compiler diagnostic system works.

Part 3: Portability

- *Implementation-defined behavior* describes how IAR C handles the implementation-defined areas of the C language.
- *IAR C extensions* describes the IAR extensions to the ISO/ANSI standard for the C programming language.

Other documentation

The complete set of IAR Systems development tools for the dsPIC 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 *dsPIC IAR Embedded Workbench™ IDE User Guide*
- Programming for the dsPIC IAR Assembler, refer to the *dsPIC IAR Assembler Reference Guide*
- Using the linker and library tools, refer to the *IAR Linker and Library Tools Reference Guide*.
- Using the Embedded C++ Library, refer to the *C++ Library Reference*, available from the IAR Embedded Workbench IDE **Help** menu.

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

FURTHER READING

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

- Barr, Michael, and Andy Oram, ed. *Programming Embedded Systems in C and C++*. O'Reilly & Associates.
- 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.
- Mann, Bernhard. *C für Mikrocontroller*. Franzis-Verlag. [Written in German.]
- Stroustrup, Bjarne. *The C++ Programming Language*. Addison-Wesley.

We recommend that you visit the websites of Microchip and IAR Systems:

- The Microchip website, **www.microchip.com**, contains information and news about the dsPIC microcontrollers.
- The IAR website, **www.iar.com**, holds dsPIC application notes and other product information.

Document conventions

Whenever the dsPIC microcontroller is mentioned, all derivatives of the dsPIC core are included, unless otherwise specified.

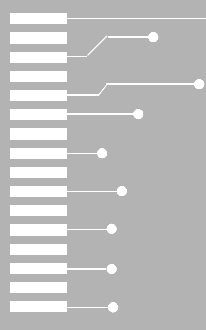
When, in this text, we refer to the programming language C, the text also applies to Embedded C++, unless it is explicitly mentioned.

TYPOGRAPHIC CONVENTIONS

This guide uses the following typographic conventions:

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

Table 1: Typographic conventions used in this guide



Part 1: Using the compiler

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

- Introduction
- Customization
- Data storage
- Functions
- Assembler language interface
- Segments and memory
- Runtime environment
- Efficient coding techniques.

Introduction

The dsPIC IAR C/EC++ Compiler supports C and Embedded C++ for Microchip's dsPIC microcontroller.

This chapter first describes how an application is built by introducing the concepts of compiling and linking.

Then the compiler is introduced, including an overview of the techniques that enable applications to take full advantage of the dsPIC microcontroller. In the following chapters the techniques will be studied in more detail.

Building applications

A typical application is built from a number of source files and libraries. The source files could be written in C, Embedded C++, or assembler language. They are compiled into object files by the dsPIC IAR C/EC++ Compiler or the dsPIC 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 or Embedded C++ standard library. Libraries can also be built using the IAR XLIB Librarian or the IAR XAR Library Builder, or 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.

COMPILING

In the command line interface, the following line compiles the source file `myfile.c` into the object file `myfile.r59` using the default settings:

```
iccdspic myfile.c
```

LINKING

The IAR XLINK Linker is used to build the final application. Normally XLINK requires the following:

- A number of object files and possibly some libraries
- The standard library containing the runtime environment and the standard language functions
- A linker command file that describes the memory layout of the target system.

In the IAR Embedded Workbench, XLINK is started automatically when you choose the **Build** option.

In the command line interface, the following line can be used to start XLINK:

```
xlink myfile.r59 myfile2.r59 -f lnkdspic.xcl cldspic0lf.r59
```

In this example, `myfile.r59` and `myfile2.r59` are object files, `lnkdspic.xcl` is the linker command file, and `cldspic0lf.r59` is the runtime library.

Data storage

One of the characteristics of the dsPIC microcontroller is that there is a tradeoff regarding the way memory is accessed, ranging from cheap access to data memory areas up to more expensive access methods that can access any location.

One of the decisions a developer of embedded systems must make is to decide where the different memory access methods should be used.

The dsPIC IAR C/EC++ Compiler allows you to set a default memory access method using data models. It also allows the access method to be specified explicitly for each individual variable.

The *Data storage* chapter covers memory access methods in greater detail.

Optimization techniques

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

The user can control the level of optimization and decide if the basic approach is to optimize for speed or for size.

For more information about the optimization of the dsPIC IAR C/EC++ Compiler, see the chapter *Efficient coding techniques*.

IAR language extension overview

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

SPECIAL FUNCTION TYPES

The dsPIC IAR C/EC++ Compiler supports interrupt functions. This allows developers to write a complete application without being forced to write any part of it in assembler language.

EXTENDED KEYWORDS

The dsPIC IAR C/EC++ Compiler provides a set of keywords that can be used for controlling the behavior of the program. There are, for example, keywords for controlling the memory type for individual variables as well as for declaring special function types.

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

The command line option `-e` makes the extended keywords available, and reserves them so that they cannot be used as variable names. See page 90 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 dsPIC IAR C/EC++ 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, the processor variant.

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

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 in-line code, either as a single instruction or as a short sequence of instructions.

For detailed reference information, see the chapter *Intrinsic functions*.

INLINE ASSEMBLER

The `asm` keyword assembles and inserts the supplied assembler statement in-line on a line-by-line basis. For example:

```
asm("MOV W0,W1");
```

Note: The `asm` keyword reduces the compiler's ability to optimize the code. We recommend the use of modules written in assembler language instead of inline assembler, since the function call to an assembler routine causes less performance reduction.

Runtime libraries

The dsPIC IAR C/EC++ Compiler supports two runtime libraries:

- The IAR CLIB Library, which is a small, efficient library well-suited for 8- and 16-bit processors. This library is not fully ANSI compliant, and does not fully support IEEE 754 floating points.
- The IAR DLIB Library, which supports ANSI C and Embedded C++.

Note: CLIB is the default library unless you run the compiler in EC++mode.

Embedded C++ overview

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

The following EC++ features are provided:

- 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.

Excluded features in C++ are those that introduce overheads 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 existent development tools.

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

- Templates
- Multiple inheritance
- Exception handling
- Runtime type information
- New cast syntax (operators `dynamic_cast`, `static_cast`, `reinterpret_cast`, and `const_cast`)
- Namespaces.

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

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

Customization

This chapter describes the configuration of the dsPIC IAR C/EC++ Compiler. This includes an overview of the processor variants and data models. The last section describes the standard runtime libraries that are included and how they correspond to the compiler options.

You should read this chapter before you read the remaining chapters in *Part 1: Using the compiler* and the chapters in *Part 2: Compiler reference*.

Processor variant

The dsPIC IAR C/EC++ Compiler supports both dsPIC microcontroller cores. The processor option reflects the presence of dsp in the target microcontroller. When you select a particular processor option for your project, several target-specific parameters are tuned to best suit that derivative.

The following table shows the mapping of processor options and which dsPIC cores they support:

Processor option	Supported dsPIC core
`-v0`	DSP instructions
`-v1`	No DSP instructions

Table 2: Mapping of processor options

Your program may use only one processor option at a time, and the same processor option must be used by all user and library modules in order to maintain consistency.

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

Use the `--cpu` or `-v` option to specify the dsPIC core; see the chapter *Compiler options* for syntax information.

Data model

The data model specifies the data memory which is used for storing:

- Non-stacked variables, i.e. global data and variables declared as static
- Dynamically allocated data, for example data allocated with `malloc` or, in Embedded C++, the operator `new`.

Your choice of processor option determines which memory models are available. The following table summarizes the characteristics of the different memory models:

Memory model	Data model option	Default memory type	Default pointer type	Const variable/ string literal placement
small	`--data_model=s`	mem	__mem	Data memory
large (default)	`--data_model=l`	mem	__ptr	Code memory

Table 3: Data model characteristics

Your program may use only one data model at a time, and the same model must be used by all user modules and all library modules. If you do not specify a data model option, the compiler will use the large data model.

The default memory type is always `mem`, regardless of data model. The difference is the default data pointer, which is `__mem` for the small data model, and `__ptr` for the large data model.

The default memory attribute can—for each individual variable—be overridden by the use of extended keywords or `#pragma` directives.

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

Use the `__data_model` option to specify the data model for your project. See the *Compiler options* chapter for more syntax information.

Runtime libraries

The runtime library includes the runtime environment and the C and Embedded C++ standard libraries. The linker will include only those routines that are required—directly or indirectly—by your application.

When building an application all parts must use the same customization settings. This also applies to the runtime library. For the dsPIC IAR C/EC++ Compiler this means that there is a runtime library for each combination of data and cpu models.

The runtime library names are constructed in the following way:

```
<type>dspic<cpu><data model><double size>.r59
```

where

- *`type`* can be `dl` for the IAR DLIB library, or `cl` for the IAR CLIB library, respectively
- *`cpu`* is either `0` or `1`, matching the `--cpu/-v` option
- *`data model`* is either `s` or `l`, for small and large data, respectively

- `double size` is either `f` or `d`, for 32-bit and 64-bit, respectively.

The following table shows the mapping of runtime libraries, data models, and code models:

Library file	Processor option	Data model	Double size
`cldspic0sf.r59`	0	Small	32
`cldspic1sd.r59`	1	Small	64
`cldspic0sf.r59`	0	Small	32
`cldspic1sd.r59`	1	Small	64
`cldspic0lf.r59`	0	Large	32
`cldspic1ld.r59`	1	Large	64
`cldspic0lf.r59`	0	Large	32
`cldspic1ld.r59`	1	Large	64

Table 4: Runtime libraries

In order to support both dsPIC microcontroller cores, a large number of runtime libraries are supplied. The library files are located in the `dspic` directory.

Note: The CLIB library is the default unless you run the compiler in Embedded C++ mode.

For information about the mapping of the `-v` processor option and dsPIC core, see Table 2, *Mapping of processor options*, page 9.

The IAR Embedded Workbench will include the correct runtime library based on the options you select; see the *dsPIC IAR Embedded Workbench™ IDE User Guide* for additional information.

Specify which runtime library to use on the XLINK command line; see the *IAR Linker and Library Tools Reference Guide*.

Data storage

This chapter first describes the fundamental ways data can be stored in memory: on the stack, in static (global) memory, or in heap memory. Then the different memory access methods and corresponding memory types are described.

Memory types are discussed in relation to pointers, structures, Embedded C++ class objects, and non-initialized memory. Placement in memory of global and static variables is then described. Finally, the structure types struct and union are discussed.

Stack, static, and heap memory

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 execution of the application. Variables that are either global or declared static are placed in this kind of memory.
- 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—and 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.

Auto variables are not allowed to have memory attributes or the `__no_init` attribute.

The stack is not only used for storing variables declared in the program; it can also contain:

- Local variables and parameters not stored in registers
- Temporary results of expressions
- The return value of functions (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 may 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 its data. Unlike a heap, a stack will never become fragmented or suffer from memory leaks.

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

Potential problems

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

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

Another problem is the risk of running out of stack. This will happen when one function calls another, which in turn calls a third, and so forth, 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.

STATIC MEMORY

All global and static variables will be placed in static memory. The word "static" in this context means that the amount of memory allocated for this type of variables does not change while the application is running.

The dsPIC microcontroller can access memory in different ways. The access methods range from generic but expensive methods that can access the full memory space to cheap methods that can access limited memory areas.

The following memory types and corresponding keywords exist:

- Full memory addressing (`__ptr`) (pointer only)
- datamem (`__mem`)
- xmem (`__xmem`)
- ymem (`__ymem`)
- sfr (`__sfr`)
- Pointer to constant in code memory (`__constptr`).

To place a variable in a memory area, you can declare it by use of extended keywords or `#pragma` directives, as in these examples:

```
__sfr int x;

#pragma type_attribute=__xmem
int y;
```

See *Memory access methods and memory types*, page 16, for a description of the limitations and advantages of each of these methods.

DYNAMIC MEMORY ON THE HEAP

Memory for objects allocated on the heap will live until they 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 Embedded 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

Systems that are using heap-allocated objects must be designed very carefully, since it is easy to end up in a situation where it is not possible to allocate objects on the heap, either because there is not enough free memory on the heap or because it is fragmented.

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

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 size of the free objects 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.

Memory access methods and memory types

This section describes the concept of access methods and the corresponding memory types used by the dsPIC IAR C/EC++ Compiler to access data. For each memory type the capabilities and limitations are discussed.

MEMORY ACCESS METHODS

The dsPIC microcontroller has two separate memory spaces. Data memory, which can be accessed efficiently, and code memory, which requires more code space and execution time to access. Code memory is only used for const declared variables, string literals, and initializer data. In the small data model, all const variables are placed in `mem`, and the resulting code is faster and more compact. Const declared variables can be placed in any memory by combining the `const` keyword with a memory specifier, e.g. `const __mem int a=34;`

The data memory for the dsPIC is divided into different zones, depending on access methods. The SFR memory is the area located between addresses 0 through 8191, also called the Access space. This area is usually used by SFRs. The X memory is normally used only with DSP code, and usually together with Y memory. The X memory accesses the whole 64Kbytes data memory, except the Y memory area. The Y memory is normally used only with DSP code, and usually together with X memory.

Do not place variables in either X or Y memory unless used in DSP code, since it places restrictions on register usage, and could result in larger and slower code. The `mem` memory is the whole 64Kbytes area, and SFR, X, and Y memory are subsets of the `mem` memory.

The code memory is an 8Mword large memory used for code, constants, and initializer data. Data placed in code memory is accessed using table read instructions. This type of access is much slower than memory access in data memory, and should generally be avoided to gain speed for the application.

Example

The example below defines three variables—`alpha`, `beta`, and `gamma`—to be placed in xmem, sfr, and in the default memory type, respectively. Note that the `#pragma` directive only controls the memory placement of the next defined variable.

```
int __xmem alpha;
#pragma type_attribute=__sfr
int beta;
int gamma;
```

MEMORY TYPES

Name	Keyword	Address range	Object size	Cost*
SFR	`__sfr`	0-8192	8 Kbytes	Same or lower
X memory	`__xmem`	0-65535 (excluding Y memory)	64 Kbytes	Same
Y memory	`__ymem`	Subset of data memory	<64 Kbytes	Same
Data memory	`__mem`	0-65535	64 Kbytes	
Const in code memory	`__constptr`	0-8M	4 Mbytes	Higher

Table 5: Memory types

* The cost column reflects cost relative to accessing data memory.

The chapter *Assembler language interface* covers this in more detail.

Pointers

In C, pointers are used for referring to the location of data or variables. This section discusses pointers in the presence of multiple memory types.

POINTERS AND MEMORY TYPES

In general, a pointer has a type. For example, a pointer that has the type `int *` points to an integer.

In the dsPIC IAR C/EC++ Compiler, a pointer also points to some kind of memory. The type of the memory is specified using a memory type keyword before the asterisk. For example, a pointer that points to an integer stored in sfr memory has the type `int __sfr *`.

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

Variables as pointers

If a variable is declared as a pointer, the memory type of the variable itself can be specified.

Examples

Below is a series of examples with descriptions. First some integer variables are defined and then pointer variables are introduced. Finally a function accepting a pointer to an integer in xmem memory is declared. The function returns a pointer to an integer in sfr memory.

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

In order to read the examples above, start from the left and add one qualifier at each step:

`int`	The basic type is an integer.
`int __xmem`	It is stored in xmem memory.
`int __xmem *`	This is a pointer to it.
`int __xmem * __sfr`	The pointer is stored in sfr memory.

Structure types and memory types

When a variable is defined, it will be placed in a memory of a certain type. Normally the default memory type is used but another memory type can be specified. For structures, the entire object is placed in the same memory type. It is not possible to place individual structure members in different memory types.

Example

In the example below, the variable `gamma` is a struct placed in sfr memory.

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

The following declaration is incorrect:

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

Embedded C++ and memory types

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

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

Also note that when calling class methods, the `this` pointer uses the default pointer type. This means that a pointer to the object must be convertable to the default pointer type. The restrictions that apply to the default pointer type also apply to the `this` pointer.

For the dsPIC microcontroller, all pointer types can be converted to the default pointer type, except `__constptr` to `__ptr` in the small data model.

Example

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

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

  __sfr static int gamma;
};


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

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

Non-initialized memory

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 *Segments and memory* 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 with the application is turned off.

For information about the __no_init keyword, see page 107. For information about the #pragma object_attribute, see page 113.

Located variables

Global and static variables can be explicitly placed at absolute addresses or in named segments using the @ operator or #pragma location. The variables must be declared either __no_init or const. If declared const, it is legal for them to have initializers.

Embedded C++ static member variables can be placed just like any other static variable.

ABSOLUTE LOCATION PLACEMENT

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 fulfil the alignment requirement for this type of variable.

Example

```
__no_init char alpha @ 0x1000; /* OK */

#pragma location=0x1004
const int beta;                /* OK */

const int gamma @ 0x1008 = 3;  /* OK */

int delta @ 0x100C;            /* Error, neither */
                               /* "__no_init" nor "const". */

const int epsilon @ 0x100F;    /* Error, misaligned. */
```

SEGMENT PLACEMENT

It is possible to place variables into named segments using either the @ operator or the #pragma location directive. The segment is specified as a string literal.

Example

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

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

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

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

ACCESSING SPECIAL FUNCTION REGISTERS

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

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

Example

```
__no_init __sfr volatile union
{
  unsigned short SR;
  struct
  {
    unsigned short C:1;
    unsigned short DC:1;
    unsigned short Z:2;
    unsigned short OV:1;
    unsigned short N:1;
    unsigned short :1;
    unsigned short RA:1;
    unsigned short DA:1;
    unsigned short :1;
    unsigned short :1;
    unsigned short SAB:1;
    unsigned short OAB:1;
    unsigned short SB:1;
    unsigned short SA:1;
    unsigned short OB:1;
    unsigned short OA:1;
  };
} @ 0x0042;
```

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

```
// whole register access
SR = 0x1020;

// Bit accesses
C = 1;
```

The header files are also suitable to use as templates when creating new header files for other dsPIC derivatives.

Anonymous structs and unions

An *anonymous struct or union* is a struct or union object that is declared without a name. Its members are promoted to the surrounding scope. An anonymous struct or union may not have a tag.

Notice that anonymous struct and unions are only available when language extensions are enabled in the dsPIC IAR C/EC++ Compiler. In EC++, however, anonymous unions are part of the language and thus always available.

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

Use the -e compiler option to enable language extensions. See *-e*, page 90 for additional information.

Example

In the example below, 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()
{
  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 is also allowed. This is for instance used for declaring I/O registers, as in the following example:

```
union
  {
    char IOPORT;
    struct
      {
          char way: 1;
          char out: 1;
      };
  } @ 0x400;
```

This declares an I/O register byte `IOPORT` at address `0x400`. The I/O register has 2 bits declared, `way` and `out`.

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

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

Functions

This chapter describes the interrupt special function type, and how to place functions into named segments.

Special function types

This section describes the interrupt special function type. The dsPIC IAR C/EC++ Compiler allows an application to fully take advantage of these powerful dsPIC features without forcing the developers to implement anything in assembler language.

INTERRUPT FUNCTIONS

In embedded systems, the use of interrupts is a method of detecting external events immediately, for example, the pressing of a button.

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

The dsPIC microcontroller allows many interrupt routines to be specified. Each interrupt routine will be associated with a vector address. The vector address corresponds to the *address* as specified in the dsPIC microcontroller documentation from the chip manufacturer.

Note: An interrupt function must have a return type of `void` and it may not specify any parameters.

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

```
#pragma vector=0x70
__interrupt void my_interrupt_routine()
{
  /* Do something */
}
```

When an interrupt function is defined with a vector, 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 dsPIC 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.

Segment placement

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

Example

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

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

Assembler language interface

This chapter describes how to write library functions in assembler language that work together with an application written in C. It covers the calling convention used by the dsPIC IAR C/EC++ Compiler, the runtime attributes, and how to write a code skeleton for calling an assembler routine from C. Some differences between the C and Embedded C++ calling conventions are also pointed out. The chapter ends with a few words about function directives.

Introduction

This section defines a simple assembler function as a practical example of how assembler functions are written. The example demonstrates some of the problems that you as a developer are faced with: How are parameters and the return value communicated from and to C? How are global variables accessed from assembler? How do I call other functions from assembler?

Note: This is just an example. There is no need to write a piece of code of this kind in assembler. The code generated from the compiler is just as efficient.

The next sections will cover these and other questions in more detail.

EXAMPLE OF ASSEMBLER FUNCTION

The example that we will use in the rest of this section is the assembler equivalence of the following C function:

```
__no_init volatile int OUTPIN @ 0x1000;
int f(int);

extern int base;

int example(int value)
{
  value += f(base);
  OUTPIN = value;
  return value;
}
```

The function, called `example`, has one parameter, `value`. The function starts by calling the function `f` (which could be written in C or assembler) with one parameter, the value of the global variable `base`. The result of this call is then added to the variable `value`.

The result of the addition is stored at the variable `OUTPIN`, which is a typical memory-mapped I/O register. The same value is also used as the return value of this function.

When planning the design of the assembler routine, the first question we must ask ourselves is: Is it possible to call the function `f` and at the same time keep all values we need, without being forced to store some of them in memory?

When calling a function, about half of the registers must be restored to their original content, whereas the other half can be used as scratch registers.

When calling function `f`, we need to keep the values of all the non-scratch registers. Clearly, this is not possible without writing something to the memory. The best place to temporarily store some values is the stack.

`value`, on the other hand, will be needed several times. If we store it on the stack, we would be forced to access the stack several times. This is undesirable. Instead, we can select one of the permanent registers and write its content to the stack. The value of the parameter `value` can then be moved to this permanent register in order to survive the call to `f`.

The non-scratch register is `W10` and the parameter register is `W0`. The `RETURN` instruction transfers the control back to the calling function.

```
example:
    MOV     W10,[W15++]
    MOV     W0,W10

    ...

    MOV     [--W15],W10
    RETURN
```

Next we need to load the value of the global variable base into `W0`, the register that should contain the parameter of `f`. First we must inform the assembler that there is an external label `base`. The assembler directive `EXTERN` is used for this. The directive is normally placed before the function itself:

```
    EXTERN  base
```

To load the code from memory, the following line us used. The value of `base` will be loaded directly into `R1`.

```
MOV     #base,W0
MOV.b   [W0],W0
```

The next step is to call the function `f`. This is performed using the `CALL` assembler instruction. Here the parameter to `f` is in `W0`, as expected. Again we must inform the assembler of the presence of `f`:

```
EXTERN  f
CALL    f
```

The return value of the function `f` can now be found in register `W0`. The next step is to add this to `value`, now to be found in `W10`. In addition, we take the opportunity to use `W0` as a result register, since the result of the addition is also the return value of this instruction. The register `W0` is the location where the return value should be placed.

```
ADD     W0,W10,W0
```

The result should also be stored at the location of the memory-mapped I/O register. Here we use the physical address, but it would also have been possible to use a symbolic label.

```
MOV     W0,4096
```

This finishes the actual code. What else is needed? First, the code must be stored in a segment. Then the label `example` must be exported from the module, and finally the assembler directive `END` must be specified to mark the end of the file.

To summarize, the complete assembler source code is:

```
    NAME    example
    RSEG    CODE:CODE:NOROOT(2)
    PUBLIC  example
    EXTERN  base
    EXTERN  f

example:
    MOV     W10,[W15++]
    MOV     W0,W10
    MOV     #base,W0
    MOV.b   [W0],W0
    CALL    f
    ADD     W0,W10,W0
    MOV     W0,4096
    MOV     [--W15],W10
    RETURN

    END
```

Calling convention

A calling convention is the way one 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.

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. Hence it must be able to deduce the calling convention from this information, as described below.

C AND C++ LINKAGE

In Embedded 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 EC++. The following is an example of a declaration that will declare a function with C linkage in both C and Embedded C++:

```
#ifdef __cplusplus
extern "C"
{
#endif
   int f(int);
#ifdef __cplusplus
}
#endif
```

FUNCTION PARAMETERS

When deciding how to pass parameters to a function, each parameter is considered in turn. The method selected is primarily based on the type of the parameter. It is also based on the availability of parameter registers. Passing parameters to registers is faster than placing them on the stack.

Register parameters versus stack parameters

Parameters can be passed to a function using two basic methods: in registers or on the stack. Clearly it is much more efficient to use registers than to take a detour via memory.

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:

- Structures—structs, unions, and classes, if larger than 4 bytes
- Unnamed parameters to variable length functions, in other words functions declared as `foo(`*`param1`*`, ...)`, for instance `printf`.

Hidden parameters

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

- If the function returns a structure, the memory location where to store the structure is passed as an extra parameter. Notice that it is always treated as the first parameter.
- If the function is a non-static Embedded 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

Register parameters use a pool of the following registers: `W0-W9`, `W14`.

8- and 16-bit parameters take the first available register, and 24/32-bit parameters take the first available register pair of `W0:W1`, `W2:W3`, `W4:W5`, `W6:W7`, or `W8:W9`.

8/16-bit values are returned in `W0`.

24/32-bit values are returned in `W0:W1`.

40-bit values are returned in `A`.

64-bit values are returned in `W0:W1:W2:W3`.

Stack parameters

Stack parameters are stored in the main memory starting at the location pointed to by the stack pointer. Above the stack pointer (towards high memory) there is free space that the called function can use. The first stack parameter is stored at the location pointed to by the stack pointer. The next one is stored at the next location on the stack that is dividable by two, etc.

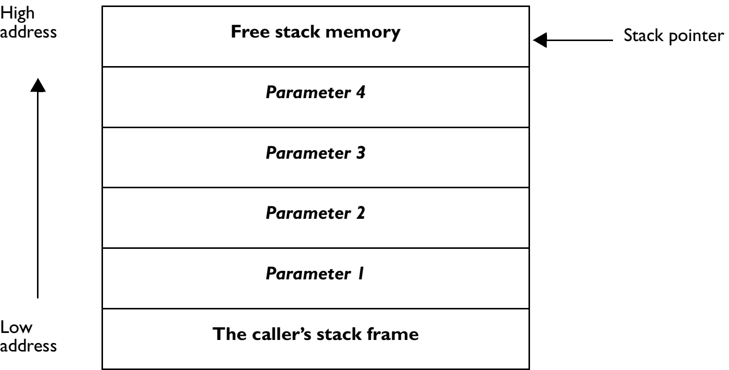


Figure 1: Storing stack parameters in memory

RETURNING A VALUE FROM A FUNCTION

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

Structures

If a structure is returned that is larger than 4 bytes, the caller of the function is responsible for allocating memory for the return value. A pointer to the memory is passed as a "hidden" first parameter.

The called function must return the value of the location.

PERMANENT VERSUS SCRATCH REGISTERS

The `W10` through `W13` registers are *permanent registers*, while the `W0-W9` and `W14` registers are available as *scratch registers*.

EXAMPLES

The following section shows a series of examples of declarations and the corresponding calling convention. The complexity of the examples increase towards the end.

Example 1

Assume that we have the following function declaration:

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

This function takes one parameter in register `W0` and the return value is passed back to its caller in register `W0`.

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:

```
add1:
     ADD #1,W0
     RETURN
```

Example 2

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

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

The calling function must reserve five bytes on the top of the stack and copy the value of the `struct` to that location. The integer parameter `y` is passed in register `W0`. The register the y parameter is passed on is dependent on the size of the default memory pointer.

Write an empty function with correct parameter and generate an assembly file as skeleton code.

MONITOR FUNCTIONS

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

For additional information, see *__monitor*, page 106.

Calling functions

ASSEMBLER INSTRUCTIONS USED FOR CALLING FUNCTIONS

This section presents the assembler instructions that can be used for calling and returning from functions on the dsPIC microcontroller.

`CALL` *`label`*

This is `CALL`, the most commonly used assembler instruction for calling functions. The location that the called function should return to (i.e. the location immediately after this instruction) is stored on the stack.

SPECIAL FUNCTION TYPES

In this section we will describe the special function type interrupt. The runtime environment that is generated is also described. When writing these special functions in assembler language, you must explicitly provide a similar runtime environment.

Interrupt functions

In the dsPIC microcontroller, when an interrupt occurs, the following happens:

- `PC` is stored on the stack along with the lower byte of the Status register
- `PC` is set to the value stored in the interrupt vector
- Registers used by the function are stored on the stack. This includes the `W0-W14`, `A`, `B`, and `SFR` registers that are associated with the `DO` and `REPEAT` instructions, as well as the `SFR CORCON`.

Other `SFR` registers used by the function, or the functions called by the function, must be saved by the user.

If an interrupt for vector `0x60`, for example, occurs, the processor will start to execute code at address `0x60`. We denote the memory area that is used as start locations for interrupts, the *interrupt vector table*. The content of the interrupt vector is normally a branch instruction jumping to the interrupt routine.

When an interrupt function with a vector is defined in C, the dsPIC IAR C/EC++ Compiler will generate both the code for the function and a small piece of code that will be placed in the interrupt vector table.

Runtime model attributes

This section introduces the concept of runtime attributes, a mechanism designed to prevent modules that are not compatible from being linked together into an application.

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.

Runtime attributes that start with two underscores are reserved to be used by IAR Systems. Any other runtime attribute is available to application developers to ensure consistency between modules.

Example

Study the object files below that could (but do not have to) define the two runtime attributes `color` and `taste`:

Object file	Color	Taste
`File1`	`blue`	not defined
`File2`	`red`	not defined
`File3`	`red`	`*`
`File4`	`red`	`spicy`
`File5`	`red`	`lean`

Table 6: Example of runtime model attributes

In this case `File1` cannot be linked with any of the other files, since the runtime attribute `color` does not match. Also, `File4` and `File5` can not be linked together since 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 both.

SPECIFYING RUNTIME ATTRIBUTES

Runtime attributes can be specified for a module written in assembler language by using the `RTMODEL` directive.

Example

```
RTMODEL color, red
```

PREDEFINED RUNTIME ATTRIBUTES

The following table shows the runtime model attributes that are available for the dsPIC IAR C/EC++ Compiler. These can be included in assembler code or in mixed C or Embedded C++ and assembler code, and will at link time be used by XLINK to ensure consistency between modules.

runtime model attribute	Value	Description
`__rt_version`	*n*	This runtime key is always present in all modules generated by the dsPIC IAR C/EC++ Compiler. If a major change in the runtime attribute scheme occurs, the value of this key changes
`__data_model`	`s` or `l`	Reflects the data model option this module is intended to be used together with.
`__double_size`	`32`	Sets the double size to 32 bits
`__long_double_size`	`64`	Sets the double size to 64 bits

Table 7: Runtime model attributes

The easiest way to find the proper settings for the `RTMODEL` directive is to compile a C or Embedded C++ module and examine the list file.

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

Calling assembler routines from C

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

- Conform to the calling convention described on page 30
- 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 a code skeleton 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 created by the C compiler. Notice that you must create a skeleton 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 only needs to declare the variables required and perform simple accesses to them.

Example

In this example, the assembler routine takes an `integer` and a `double`, and then returns a `char`:

```
int globInt;
double globDouble;

int func(int arg1, double arg2)
{
  int locInt = arg1;
  globInt = arg1;
  globDouble = arg2;
  return locInt;
}

void main(void)
{
  int locInt = globInt;
  globInt = func(locInt, globDouble);
}
```

Note: A low optimization level is used 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.

In the IAR Embedded Workbench, specify list options on file level. Select the file in the Project window. Then choose **Project>Options**. In the **ICCDSPIC** category, select **Override inherited settings**. 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:

```
iccdspic shell -lA . -s3
```

The `-lA` option creates an assembler language output file including C or Embedded C++ source lines as assembler comments. The `.` (period) specifies that the assembler file should be named in the same way as the C or Embedded C++ module, i.e. `shell`, but with the filename extension `s59`.

The result is the assembler source file `shell.s59`, which contains the declarations, function call, function return, and variable accesses.

Viewing the output file

The output file contains the following important information:

- The calling conventions
- The return values
- The global variables
- The function parameters
- How to create space on the stack (auto variables)

The following list shows an example of an assembler output file with source comments. The compiler option used to generate this particular list file example was `-s3`. Note that the first parameter uses the first available parameter register `W0`. The second parameter is passed in the first available register pair; `W2,W3`.

```
        NAME shell

        RTMODEL "__data_model", "l"
        RTMODEL "__double_size", "32"
        RTMODEL "__long_double_size", "32"
        RTMODEL "__rt_version", "1"

        RSEG CSTACK:DATA:NOROOT(1)

        EXTERN `__INIT_MEM_Z`
        EXTERN `?CLDSPIC_1_00_L00`

        PUBLIC func
        FUNCTION func,0203H
        LOCFRAME CSTACK, 2, STACK
        PUBLIC globDouble
        PUBLIC globInt
        PUBLIC main
        FUNCTION main,021a03H
        LOCFRAME CSTACK, 2, STACK

        RSEG MEM_Z:DATA:NOROOT(1)
// C:\src\dspic\test\shell.c
//    1 int globInt;
globInt:
```

```
    DS 2
    REQUIRE `__INIT_MEM_Z`

        RSEG MEM_Z:DATA:NOROOT(1)
//    2 double globDouble;
globDouble:
    DS 4
    REQUIRE `__INIT_MEM_Z`
//    3

        RSEG CODE:CODE:NOROOT(2)
//    4 int func(int arg1, double arg2)
func:
        REQUIRE `?CLDSPIC_1_00_L00`
//    5 {
        ; * Stack frame (at entry) *
        ; Param size: 0
        ; Return address size: 4
        ; Saved register size: 0
        ; Auto size: 0
//    6   int locInt = arg1;
        MOV       W0,W1
//    7   globInt = arg1;
        MOV       W0,globInt
//    8   globDouble = arg2;
        MOV       W2,globDouble
        MOV       W3,globDouble+2
//    9   return locInt;
        MOV       W1,W0
        RETURN
//   10 }
//   11

        RSEG CODE:CODE:NOROOT(2)
//   12 void main(void)
main:
        FUNCALL main, func
        REQUIRE `?CLDSPIC_1_00_L00`
//   13 {
        ; * Stack frame (at entry) *
        ; Param size: 0
        ; Return address size: 4
        ; Auto size: 0
//   14   int locInt = globInt;
        MOV       globInt,W0
//   15   globInt = func( locInt, globDouble);
        ; Setup parameters for call to function func
```

```
        MOV       globDouble,W2
        MOV       globDouble+2,W3
        CALL      func
        MOV       W0,globInt
//   16 }
        RETURN

        END

// 26 words in segment CODE
//  6 bytes in segment MEM_Z
//
// 26 words of CODE memory
//  6 bytes of DATA memory
//
//Errors: none
//Warnings: none
```

For information about the runtime model attributes used in this example, see the table *Runtime model attributes*, page 36.

Calling assembler routines from Embedded C++

The C calling convention does not apply to Embedded C++ functions. Most importantly, a function name is not sufficient to identify an Embedded 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 above description. An assembler routine may therefore be called from Embedded 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 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 EC++ files cannot, in general, be passed through the assembler.
- It is impossible to refer to or define EC++ functions that do not have C linkage in assembler.

Function directives

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

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

SYNTAX

```
FUNCTION <label>,<value>
ARGFRAME &lt;segment&gt;, <size>, <type>
LOCFRAME &lt;segment&gt;, <size>, <type>
FUNCALL <caller>, <callee>
```

PARAMETERS

label	Label to be decared as function.

value	Function information.
segment	Segment in which argument frame or local frame is to be stored.
size	Size of argument frame or local frame.
type	Type of argument or local frame; either `STACK` or `STATIC`.
caller	Caller to a function.
callee	Called function.

DESCRIPTION

`FUNCTION` declares the *label* name to be a function. *value* encodes extra information about the function.

`FUNCALL` declares that the function *caller* calls the function *callee*. *callee* can be omitted to indicate an indirect function call.

`ARGFRAME` and `LOCFRAME` declare how much space the frame of the function uses in different memories. `ARGFRAME` declares the space used for the arguments to the function, `LOCFRAME` the space for locals. *segment* is the segment in which the space resides. *size* is the number of bytes used. *type* is either `STACK` or `STATIC`, for stack-based allocation and static overlay allocation, respectively.

`ARGFRAME` and `LOCFRAME` always occur immediately after a `FUNCTION` or `FUNCALL` directive.

After a `FUNCTION` directive for an external function, there can only be `ARGFRAME` directives, which indicate the maximum argument frame usage of any call to that function. After a `FUNCTION` directive for a defined function, there can be both `ARGFRAME` and `LOCFRAME` directives.

After a `FUNCALL` directive, there will first be `LOCFRAME` directives declaring frame usage in the calling function at the point of call, and then `ARGFRAME` directives declaring argument frame usage of the called function.

Segments and memory

This chapter introduces the concept of segments and describe 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 chapter also contains an overview of the linker command file, which is used for controlling the placement of segments in memory.

Note that the information in this chapter is conceptual; it is strictly generic and not related to any particular compiler unless stated otherwise. For product-specific details, see the linker command file included in your product package.

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

What is a segment?

A segment is a piece of data or code that should be mapped to a physical location in memory. The segment could either be placed in RAM or in ROM. Segments that are placed in RAM do not have any content, they only occupy space.

The compiler has a number of predefined segments for different purposes. Each segment has a name describing the contents of the segment. In addition, you can define your own segments.

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. It is important to remember that, from the linker's point of view, all segments are equal, they are simply named parts of memory.

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

LINKER SEGMENT TYPE

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

XLINK supports a number of other segment types than the ones described below. However, most of them exist to support other types of microcontrollers.

By default the compiler uses only the following XLINK segment types:

XLINK segment type	Description
CODE	Contains executable code, constants and initializer data
DATA	Contains data placed in RAM

Table 8: XLINK segment types

PLACEHOLDER SEGMENTS

The runtime environment of the compiler uses *placeholder segments*, empty segments that are used for marking a location in memory. Any type of segment can be used for placeholder segments.

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.

Since the chip-specific details are specified in the linker command file and not in the source code, the linker command file also ensures code portability. Basically, you can use the same source code with different derivatives just by rebuilding the code using an appropriate linker command file.

The `config` directory contains at least one ready-made linker command files. The file contains the information required by the linker and is ready to be used. If, for example, your application uses external RAM, you will only need to provide details about the external RAM memory area. Remember not to change the original file. We recommend that you make a copy and modify the copy instead.

Notice that the supplied linker command file includes comments explaining the entire contents.

THE CONTENTS OF THE LINKER COMMAND FILE

In particular, the linker command file specifies:

- The placement of segments
- The stack size
- The heap size used by the IAR DLIB library.

The linker command file contains three different types of XLINK command line options.

- The CPU used, for example:

  `-cdspic`

  This specifies that we are using the dsPIC microcontroller.
- Definitions of constants used later in the file. These are defined using the `-D` option.
- 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, whereas the latter will try to rearrange them in order to make better use of memory. The `-P` option is useful when the memory used to place one segment type is not continuous.

CUSTOMIZING A LINKER COMMAND FILE

The examples below show the general principles for how to set up a linker command file. The target system is assumed to have the following fictitious memory layout:

Range	Type
0x2000—0xCFFF	ROM
0x20000—0x3FFFF	ROM
0x0—0x1FFF	RAM
0x10000-0x11FFF	RAM

Table 9: Linker command file example

The ROM can be used to store `CONST` and `CODE` memory. The RAM memory can contain segments of `DATA` type.

The only change you will normally have to make to the supplied linker command file is to suit the details of the target hardware memory map.

Example 1

The following will place the segments `MYSEGMENTA` and `MYSEGMENTB` in `CODE` memory (that is ROM) in the memory range of `0x2000-0x2FFF`.

```
-Z(CONST)MYSEGMENTA,MYSEGMENTB=2000-2FFF
```

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(CODE)MYSEGMENTA=2000-2FFF
-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(CODE)MYSMALLSEGMENT=2000-20FF
-Z(CODE)MYLARGESEGMENT=2000-2FFF
```

Even though it is not strictly required, make sure to always specify the end of memory ranges. If you do this, the IAR XLINK Linker will alert you if your segments do not fit. If you do not specify the end of memory ranges, you will not be alerted by the linker. See the *IAR Linker and Library Tools Reference Guide* for more details.

Example 2

The following example 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 the XLINK `-P` option, which will make efficient use of the memory area.

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 are declared static, as described in *Memory access methods and memory types*, page 16.

This section describes how the segment types correspond to segment groups, and the segments that are part of the segment groups.

Segment naming

The fictitious example started in *Customizing a linker command file*, page 45, uses the following memory types:

Memory type	Range
sfr	`0-0x1FFF`
mem	`0-0xFFFF`

Table 10: Memory types

The static memory types in this fictitious example correspond to the following basic segment groups. The first part of the name of a segment in each segment group corresponds to the segment keyword:

Segment group	First part of name
sfr	`SFR`
mem	`MEM`

Table 11: Segment groups

The variables declared in each of the groups can be divided into the following categories:

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

When an application is started, the `cstartup` module initializes memory 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.

For each of the segment groups, some of the following segments exist:

Usage	Type	Suffix
Zero-initialized data	`DATA`	`Z`
Non-zero initialized data	`DATA`	`I`
Initializers for the above	`CODE`	`ID`
Non-initialized data	`DATA`	`N`

Table 12: Segments in segment groups

The names of the actual segments are *NAME_SUFFIX*. For example, the segment MEM_Z contains the mem variables that should be initialized to zero when the system starts.

Initialized data

The data in the ROM segment with suffix ID is copied to the corresponding I segment when the system starts.

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.

SFR_I	0x1000-0x10FF and 0x1200-0x12FF
SFR_ID	0x4000-0x41FF

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

SFR_I	0x1000-0x10FF and 0x1200-0x12FF
SFR_ID	0x4000-0x40FF and 0x4200-0x42FF

The ID segment can, for all segment groups, be placed anywhere in memory, since it is not accessed using the corresponding access method.

sfr

The SFR segments must be placed in the theoretical memory range 0-0x1FFF. In this example these segments are placed in the available RAM area 0x0000–0x1FFF.

The segment SFR_ID can be placed anywhere in memory.

mem

The MEM segments data must be placed in the theoretical memory range 0-0xFFFF, which is anywhere in this example.

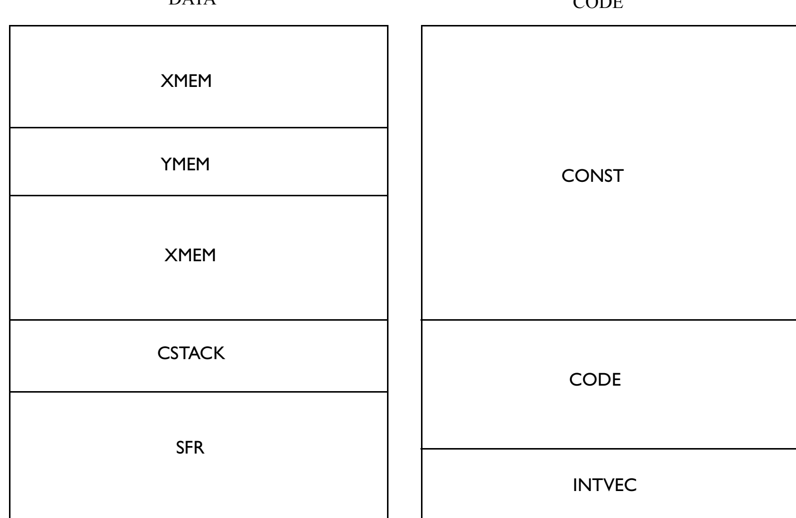
The segment MEM_ID can be placed anywhere in memory.

The linker command file

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

```
// The ROM segments
-Z(CODE)SFR_ID,MEM_ID=2000-CFFF

// The RAM segments
-Z(DATA)SFR_I,SFR_Z,SFR_N=0-1FFF
-Z(DATA)MEM_I,MEM_Z,MEM_N=10000-11FFF

// Constants
-P(CODE)CONST=2000-CFFF
```

This gives the following placement of index segments:



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. The `cstartup` module initializes the stack pointer to the end of the stack segment called `CSTACK`.

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

```
-D_CSTACK_SIZE=size
```

Note that the size is written hexadecimally.

At the end of the linker file the actual segment is defined in the memory area available for the stack:

```
-Z(DATA)CSTACK+_CSTACK_SIZE=start-end
```

Stack size

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 small, the stack will normally overwrite the variable storage which is likely to result in program failure. If the given stack size is too large, RAM will be wasted.

Stack types

The dsPIC IAR C/EC++ Compiler supports two types of stacks: the interrupt stack and the program stack.

The default linker command file contains the following definition:

Stack type	Segment name	Size in hex	Memory area
User stack	CSTACK	1000	800-2000

Table 13: Stack types

THE HEAP

The heap contains dynamically allocated data. Initially, the free memory of the heap will be the memory that is placed in the segment `HEAP`. This segment is only included in the application if dynamic memory allocation is actually used.

The size of the heap is defined in much the same manner as the size of the stack:

```
-D_HEAP_SIZE=size
```

and

```
-Z(DATA)HEAP+_HEAP_SIZE=start-end
```

LOCATED DATA

A variable that has been explicitly placed at an address, for example by using the compiler `@` syntax, will be placed in the `<memtype>_A` segment. It is used for items declared as `__no_init`. The individual segment part of the segment knows its location in the memory space and it does not have to be specified in the linker command file.

Code segments

This section contains descriptions of the segments used for storing code and the interrupt vector table.

STARTUP CODE

The segment `ICODE` contains code used during system setup. The startup code should be placed at the location where the chip starts executing code after a reset.

In this example, the following line in the linker command file will place the `ICODE` segment at address `0x2000`:

```
-Z(CODE)ICODE=0x2000
```

NORMAL CODE

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

```
-Z(CODE)CODE=2000-BFFF
```

EXCEPTION VECTORS

The exception vectors are typically placed in the segment `INTVEC`.

Embedded C++ dynamic initialization

In Embedded 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.

This segment can be placed anywhere in memory, for example:

```
-Z(CODE)DIFUNCT=0000-1FFFF
```

Runtime environment

This chapter will describe the cstartup file which handles system initialization and termination. We will present how an application can control what happens before main is called, either by providing a custom __low_level_init routine, or by changing the cstartup file.

The standard library uses a small set of low-level input and output routines as a base for a wide range of I/O routines. This chapter describes how the low-level routines can be replaced by an application, so that it can use the standard function to—for example—communicate with the outside world or providing a memory-based file system.

This chapter also covers the methods used for communicating with the IAR C-SPY™ Debugger.

The cstartup.s59 file

This section will cover what actions the runtime environment performs during startup and termination of applications. In the next couple of sections customization is discussed.

SYSTEM STARTUP

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

- The stack pointer (`SP`) is initialized
- The custom-provided function `__low_level_init` is called, allowing 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
- Global Embedded C++ objects are constructed
- The `main` function is called, which starts the application.

SYSTEM TERMINATION

An application can perform a normal termination in two different ways:

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

Since the 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 C. It calls a small function _exit provided by the cstartup file.

The _exit function will perform the following operations:

- Call functions registered to be executed when the application ends. This includes Embedded C++ destructors for static and global variables, and functions registered with the standard C function atexit
- All open files are closed
- __exit is called
- When __exit is reached the system is stopped. 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.

__low_level_init

Some applications may need to initialize I/O registers, 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 before the data segments are initialized. The value returned by __low_level_init determines whether data segments are initialized. If the function returns 0, the data segment will not be initialized.

A skeleton for this function is supplied in the low_level_init.c file, which is installed with the product.

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

Customizing cstartup.s59

The cstartup.s59 file itself is well commented and is not described in detail in this guide. This section however presents some general techniques used in the file; it covers some background knowledge that can be useful when modifying the cstartup.s59 file. and then describes how the customized cstartup.s59 file could be used.

MODULES AND SEGMENT PARTS

In order to understand how the cstartup code is designed, it is imperative to have a clear understanding of modules and segment parts, and how the IAR XLINK Linker™ treats them.

An assembler module starts with a MODULE directive and ends with an ENDMOD directive. Inside the module a number of segment parts reside. Each segment part begins with an RSEG directive.

When XLINK builds an application, it starts with a small number of modules that have been declared as root. It then continues to include all modules that are referred from the already included modules. XLINK then discards unused segment parts.

Segment parts, REQUIRE, and the falling-through trick

The cstartup.s59 file has been designed to use the mechanism described above so that as little as possible of unused code will be included in the linked application.

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

A piece of code or data is not included if it is not used or referred to with the REQUIRE assembler directive.

The segment parts of the CSTART module defined in the cstartup.s59 file are guaranteed to be placed immediately after each other. XLINK will not change the order of the segment parts or modules since the segments are placed using the -Z option.

The above lets the cstartup.s59 file specify code in subsequent segment parts and modules that are designed so that some of the parts may not be included by XLINK. The following example shows this technique:

```
    MODULE   doSomething

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

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

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

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


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

?end_of_test:
    RETURN         // This is included if ?do_something above
                   // is included.
    ENDMOD
```

CALL FRAME INFORMATION

When debugging an application, the IAR C-SPY Debugger is capable of displaying the call stack, i.e. the functions that have called the current function. In order to ensure that the call stack is correctly displayed when executing code written in assembler language, information about the the call frame must be provided. This is done by use of the assembler directive `CFI`. For more information, see the *dsPIC IAR Assembler Reference Guide*.

MODIFYING THE CSTARTUP.S59 FILE

Do not modify the `cstartup.s59` file unless required by your application. If you need to modify it, we recommend that you follow the overall procedure for creating a modified copy of the `cstartup.s59` file and adding it to your project.

In the IAR Embedded Workbench

1. Copy the assembler source file `cstartup.s59`, which is supplied in the product directory, to your project directory. Make any required modifications to the copy and save the file under the same name.
2. Add the file `cstartup.s59` to your project.
3. Select the option **Ignore CSTARTUP in library** on the **Include** page in the **XLINK** category of project options. See the *dsPIC IAR Embedded Workbench™ IDE User Guide* for additional information.
4. Rebuild your project.

From the command line

1. Copy the assembler source file `cstartup.s59`, which is supplied in the product directory, to your project directory. Make any required modifications to the copy.

2. Use the assembler option `-D` to specify the memory and code model symbols, for example:

```
adspic cstartup -DMEMORY_MODEL=[s|l]
```

   This will create an object module file named `Cstartup.r59`.

3. Specify the XLINK option `-C` in front of the name of the library to ignore the standard `Cstartup` file. See *Linking*, page 3. Then link your application.

Input and output

The dsPIC IAR C/EC++ Compiler includes two different sets of runtime libraries, with different I/O functions.

THE IAR CLIB LIBRARY

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 upon the following files:

- `putchar.c`, which serves as the low-level part of functions such as `printf`
- `getchar.c`, which serves at the low-level part of functions such as `scanf`. To customize `getchar.c`, use the method described for `putchar.c`.

Customizing putchar in the IAR Embedded Workbench

The following section describes the procedure for adding a customized version of `putchar` to your project:

1. Copy `putchar.c` to your project directory.

2. Make the required additions to the source `putchar.c`, and save it under the same name (or create your own routine using `putchar.c` as a model).

3. Add the customized `putchar.c` as a program module to your project.

4. Rebuild your project.

Customizing putchar from the command line

The following section describes the procedure for replacing the original C library with a library containing a customized version of `putchar`:

1. Make the required additions to the source `putchar.c`, and save it under the same name (or create your own routine using `putchar.c` as a model).

   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. It can, for example, depend on the selected processor variant.

2. Compile the modified putchar using the appropriate processor variant and the `--library_module` option, for example:

   ```
   iccdspic putchar --library_module
   ```

   This will create a replacement object module file named `putchar.r59`.

   Note: The code model and/or data model must be the same for `putchar` as for the rest of your code.

3. Test the modified module before installing it in the library by using the `-A` XLINK option. Place the following line into your linker command file, before the library reference:

   ```
   -A putchar
   ```

   This causes your version of `putchar.r59` to load instead of the one in the *`library`* library. For information about the XLINK options, see the *IAR Linker and Library Tools Reference Guide*.

4. Add the new `putchar` module to the appropriate runtime library module, replacing the original.

   Note: Be sure to save your original library file before you overwrite the `putchar` module.

   For example, to add the new `putchar` module to the correct library, use the following command:

   ```
   xlib
   ```

```
def-cpu dspic
rep-mod putchar library
exit
```

The library module *library* will now have the modified putchar instead of the original one.

def-cpu and rep-mod are abbreviations of the XLIB commands DEFINE-CPU and REPLACE-MODULES. Module names are case sensitive in XLIB. For additional information about the IAR XLIB Librarian, see the *IAR Linker and Library Tools Reference Guide.*

Note: If you use XLIB to replace the original putchar module, you must repeat the steps described above whenever you change the library, for example when you upgrade to a new version of the product.

printf and sprintf

The printf and sprintf functions use a common formatter called _formatted_write. The ANSI standard 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 functionality as _formatted_write, except that floating-point numbers are not supported. Any attempt to use a %f, %g, %G, %e, and %E specifier will produce the error:

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

_medium_write is considerably smaller than _formatted_write.

_small_write

As for _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% of the size of _formatted_write.

Specifying the write formatter version

In the linker command files provided with the product, the _small_write formatter can be selected with the following line:

```
-e_small_write=_formatted_write
```

In the IAR XLINK Linker, the full ANSI version is default. To use full ANSI, remove the line.

To select _medium_write, replace the line with:

```
-e_medium_write=_formatted_write
```

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 and/or non-standard output devices.

For such applications, a highly reduced version of the entire printf function (without sprintf) is supplied in source form in the file intwri.c. This file can be modified to your requirements and the compiled module inserted into the library in place of the original using the procedure described for putchar; for additional information, see *Customizing putchar from the command line*, page 58.

Scanf and sscanf

In a similar way to the printf and sprintf functions, scanf and sscanf use a common formatter called _formatted_read. The ANSI standard 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 in the standard C library.

_medium_read

The _medium_read formatter has the same functionality as _formatted_read, except that floating-point numbers are not supported. _medium_read is considerably smaller than _formatted_read.

Specifying the read formatter version

In the linker command files provided with the product, the _medium_read formatter can be selected with the following line:

```
-e_medium_read=_formatted_read
```

In the IAR XLINK Linker, the full ANSI version is default. To use full ANSI, remove the line.

THE IAR DLIB LIBRARY

This library contains a large number of powerful functions for I/O operations. In order to simplify adaption to specific hardware, all I/O functions call a small set of primitive functions, each designed to accomplish one particular task; for example, __open acts as if it opens a file and __write outputs a number of characters.

The primitive I/O files are located in the product directory.

I/O function	File	Description
__close()	close.c	Close a file.
__lseek()	lseek.c	Set the file position indicator.
__open()	open.c	Open a file.
__read()	read.c	Read a character buffer.
__readchar()	readchar.c	Read a character.
__write()	write.c	Write a character buffer.
__writechar()	writechar.c	Write a character.
remove()	remove.c	Remove a file.
rename()	rename.c	Rename a file.

Table 14: I/O files

I/O functions

The primitive I/O functions 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 functions using whatever facilities the hardware environment provides.

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.

The default implementation of the primitive I/O functions maps the I/O streams associated with `stdin` and `stdout` to the debugger; all other operations are ignored.

Example

The code in the following example uses memory-mapped I/O to write to an LCD display.

```
int __writechar(int handle, unsigned char c)
{
  /* Write only if it is to standard output. */
  if (handle == 1)
  {
    unsigned char * LCD_IO;
    LCD_IO = (unsigned char *)address;
    * LCD_IO = c;
    return c;
  }
  else
```

```
  {
    return -1;
  }
}
```

C-SPY debugger interface

The low-level debugger interface is used for communicating between the debugged application 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`, `?C_VIRTUAL_IO`, and `?C_GETCHAR` are 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. Should no input be 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 **Debug info with terminal I/O** selected; see the *dsPIC IAR Embedded Workbench™ IDE User Guide*.

Termination

The debugger stops executing if the execution of the application reaches the special label `?C_EXIT`.

Efficient coding techniques

This chapter provides hints on how to write efficient code, an overview of Embedded C++, and information about the settings required to make object files compatible and thus linkable.

Programming hints

This section contains recommendations on how to write efficient code for the dsPIC microcontroller.

OPTIMIZING FOR SIZE OR SPEED

The dsPIC IAR C/EC++ Compiler allows you to generate code that is optimized either for size or for speed, at a selectable optimization level. Both compiler options and `#pragma` directives are available for specifying the preferred type and level of optimization:

- The chapter *Compiler options* in *Part 2: Compiler reference* contains reference information about the following command line options, which are used for specifying optimization type and level: `--no_code_motion`, `--no_cse`, `--no_inline`, `--no_unroll`, `-s[3|6|9]`, and `-z[3|6|9]`.
  Refer to the *dsPIC IAR Embedded Workbench™ IDE User Guide* for information about the compiler options available in the IAR Embedded Workbench.
- Refer to *#pragma optimize*, page 114, for information about the `#pragma` directives that can be used for specifying optimization type and level.

Normally you would use the same optimization level for an entire project or file, but the `#pragma optimize` directive allows you to fine-tune the optimization for a specific code section, for example a time-critical function.

Optimization hints include the following:

- Sensible use of the memory attributes (see the chapter *Extended keywords*) can improve the speed and reduce the code size in critical applications.
- Small local functions may be inlined by the compiler if declared `static`, which allows further optimizations. This feature can be turned off with the `--no_inline` option. See page 95 for information about this option.
- Module-local variables (i.e. variables declared `static`) are preferred over global variables.
- Avoid taking the address of local and static variables.
- Avoid using inline assembler. Instead, try writing the code in C or Embedded C++, use intrinsic functions, or write a separate assembler module.

The purpose of optimization is either to reduce the code size or to improve the execution speed. Speed optimization alternatives sometimes also reduce the code size.

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. We therefore recommend that you use a low optimization level during the development phase of your project, and a high optimization level for the release version.

SAVING STACK SPACE AND RAM MEMORY

- Avoid long call chains and recursive functions in order to save stack space.
- Declare variables with a *short* life span as auto variables. When the life spans for these variables end, they will be popped from the stack and the previously occupied memory can than be reused. Globally declared variables will occupy data memory during the whole program execution. Be careful with auto variables, though, as the stack size can exceed its limits.
- Avoid passing large non-scalar parameters; in order to save stack space, you should instead pass them as pointers.

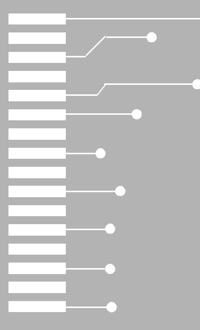
USING EFFICIENT DATA TYPES

- When declaring functions, use prototypes. They allow the compiler to generate efficient code and provide type checking of parameters.
- Floating-point types are ineffective. If possible, try to use integers instead. If you have to use floating-point types, notice that 32-bit floats are more efficient than 64-bit type doubles.
- Using bitfields larger than 1 bit generates code that is both larger and slower than if non-bitfield integers were used.

Module compatibility

When building an application, the following options should be the same for all modules:

- Processor option (`-v`)
- Data model (`--data_model`)
- Double size (`--64bit_doubles`)



Part 2: Compiler reference

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

- Data representation
- Segment reference
- Compiler options
- Extended keywords
- #pragma directives
- Predefined symbols
- Intrinsic functions
- Library functions
- Diagnostics.

Data representation

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

See the chapter *Efficient coding techniques* for information about which data types and pointers provide the most efficient code.

Fundamentals

There are some basic facts that you need to know before you can use the dsPIC IAR C/EC++ Compiler with ease.

ALIGNMENT

The alignment of a data object controls how it can be stored in memory. Objects with alignment 2 must for example be stored at an address dividable by 2. The reason for alignment is that the dsPIC microcontroller can access 2-byte objects using one assembler instruction only when the object is stored at such addresses.

BYTE ORDER

The dsPIC microcontroller stores data in memory using the Little Endian method. This means that the lowest byte is stored at the lowest address in memory.

Data types

The dsPIC IAR C/EC++ Compiler supports all ISO/ANSI C basic data types. Signed variables are stored in two's complement form.

INTEGER TYPES

The following table gives the size and range of each C/EC++ integer data type:

Data type	Size	Range	Alignment
signed char	8 bits	-128 to 127	1
unsigned char	8 bits	0 to 255	1
int, short	16 bits	-32768 to 32767	2
unsigned int, unsigned short, wchar_t	16 bits	0 to 65535	2

Table 15: Integer types

Data type	Size	Range	Alignment
long	32 bits	-2^{31} to $2^{31}-1$	2
unsigned int, unsigned long	32 bits	0 to $2^{32}-1$	2
long long	40 bits	-2^{39} to $2^{39}-1$	2
unsigned long long	40 bits	0 to $2^{40}-1$	2

Table 15: Integer types

Enum type

The `enum` keyword creates each object with the shortest signed or unsigned integer type required to contain its value.

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. Notice, however, that the library is compiled with `char` types as unsigned.

Bitfields

In ISO/ANSI C, `int` and `unsigned int` can be used as base type for integer bitfields. In the dsPIC IAR C/EC++ Compiler, any integer type can be used as base type.

Bitfields in expressions will have the same data type as the integer base type.

By default the dsPIC IAR C/EC++ 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

Floating-point values are represented by 32- and 64-bit numbers in standard IEEE format. The ranges and sizes for the different floating-point types are:

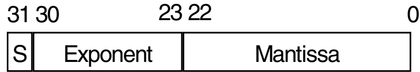
Type	Range (+/-)	Decimals	Exponent	Mantissa
float	±1.18E-38 to ±3.39E+38	7	8	23
double	±2.23E-308 to ±1.79E+308	15	11	52

Table 16: Floating-point types

Double can be either 32 or 64 bits in size, depending on the `--64bit_doubles` command line option.

32-bit floating-point format

The memory layout of 4-byte floating-point numbers is:



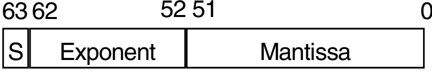
The value of the number is:

$(-1)^{S} * 2^{(Exponent-127)} * 1.Mantissa$

The precision of the float operators (`+`, `-`, `*`, and `/`) is approximately 7 decimal digits.

64-bit floating-point format

The memory layout of 8-byte floating-point numbers is:



The value of the number is:

$(-1)^{S} * 2^{(Exponent-1023)} * 1.Mantissa$

The precision of the float operators (`+`, `-`, `*`, and `/`) is approximately 15 decimal digits.

Special cases

For both 32-bit and 64-bit floating-point formats:

- Zero is represented by zero mantissa and exponent. The sign bit signifies positive or negative zero.
- Infinity is represented by setting the exponent to the highest value and the mantissa to zero. The sign bit signifies positive or negative infinity.
- Not a number (`NaN`) is represented by setting the exponent to the highest positive value and the mantissa to a non-zero value. The value of the sign bit is ignored.
- Denormalized numbers are used to represent values smaller than what can be represented by normal values. The drawback is that the precision will decrease with smaller values. The exponent is set to 0 to signify that that the number is denormalized even though the number is treated as the exponent would have been 1. Unlike normal numbers, denormalized numbers do not have an implicit 1 as MSB of the mantissa. The value of a denormalized number is:

  ```
  (-1)S * 2(1-BIAS) * 0.Mantissa
  ```

  where `BIAS` is 127 and 1023 for 32-bit and 64-bit floats, respectively.

Pointers

The dsPIC IAR C/EC++ Compiler has six basic types of pointers: sfr, xmem, ymem, mem, constptr, and ptr.

SIZE

The sfr, xmem, ymem, and mem pointers are 2 bytes, while the constptr and ptr pointers are 3 bytes.

CASTING

Casts between pointers have the following characteristics:

- 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 data pointers to function pointers and vice versa is illegal.
- Casting function pointers to integer types would give an undefined result.

size_t

`size_t` is the unsigned integer type required to hold the maximum size of an object. In the dsPIC IAR C/EC++ Compiler, the size of `size_t` is 32 bits.

ptrdiff_t

`ptrdiff_t` is the type of the signed integer required to hold the difference between two pointers to elements of the same array. In the dsPIC IAR C/EC++ Compiler, the size of `ptrdiff_t` is 32 bits.

Structure types

Structure members are stored sequentially in the order in which they are declared: the first member has the lowest memory address.

ALIGNMENT

Both a `struct` and a `union` inherits the aligment requirements of its 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).

For example:

```
struct {
  short s; /* stored in byte 0 and 1 */
  char c;  /* stored in byte 2 */
  long l;  /* stored in byte 4, 5, 6, and 7 */
  char c2; /* stored in byte 8 */
} s;
```

The following diagram shows the layout in memory:

s.s	s.c	pad	s.l	s.c2	pad
2 bytes	1 byte	1 byte	4 bytes	1 byte	3 bytes

The alignment of the `struct` is 4 bytes and its size is 12 bytes.

Data types in Embedded C++

In Embedded C++, all plain C data types are represented in the same way as described earlier in this chapter. However, if any Embedded C++ features are used for a type, no assumptions can be made concerning the data representation. This means, for example, that it is not legal to write assembler code that accesses class members.

Segment reference

The dsPIC IAR C/EC++ Compiler places code and data into named segments which are referred to by the IAR XLINK Linker™. Details about the segments are required for programming assembler language modules, and are also useful when interpreting the assembler language output from the compiler.

For information about how to define segments in the linker command file, see *Customizing a linker command file*, page 45.

Summary of segments

The following table lists the segments that are available in the dsPIC IAR C/EC++ Compiler. Notice that *located* denotes absolute location using the `@` operator or the `#pragma location` directive.

Segment	Description
CODE	Holds the user program code.
CONST	Holds the constants and string literals placed in code.
CSTACK	Holds the data stack.
DIFUNCT	Holds pointers to constructor blocks that should be executed by `cstartup` before `main` is called.
HEAP	Holds the heap data used by `malloc` and `free`.
ICODE	Holds the startup code.
INTVEC	Contains the reset and interrupt vectors.
MEM_A	Holds located `__mem` variables.
MEM_I	Holds initialized `__mem` variables.
MEM_ID	Holds initializer data for initialized `__mem` variables.
MEM_N	Holds `__no_init__mem` variables.
MEM_Z	Holds zero-initialized `__mem` variables.
RCODE	Holds the library code.
SFR_A	Holds located `__sfr` variables.
SFR_I	Holds initialized `__sfr` variables.
SFR_ID	Holds initializer data for initialized `__sfr` variables.
SFR_N	Holds `__no_init__sfr` variables.
SFR_Z	Holds zero-initialized `__sfr` variables.

Table 17: Segment summary

Segment	Description
XMEM_A	Holds located __xmem variables.
XMEM_I	Holds initialized __xmem variables.
XMEM_ID	Holds initializer data for initialized __xmem variables.
XMEM_N	Holds __no_init__xmem variables.
XMEM_Z	Holds zero-initialized __xmem variables.
YMEM_A	Holds located __ymem variables.
YMEM_I	Holds initialized __ymem variables.
YMEM_ID	Holds initializer data for initialized __ymem variables.
YMEM_N	Holds __no_init__ymem variables.
YMEM_Z	Holds zero-initialized __ymem variables.

Table 17: Segment summary (Continued)

Descriptions of segments

The following section gives reference information about each segment.

The linker segment type CODE, or DATA indicate whether the segment should be placed in ROM or RAM memory areas.

This chapter mentions many of the extended keywords. For detailed information about the keywords, see the chapter *Extended keywords*.

CODE Holds user program code.

Linker segment type

CODE

Memory range

This segment can be placed anywhere in memory.

CONST Holds the constants and string literals placed in code.

Linker segment type

CODE

Memory range

This segment can be placed anywhere in memory.

CSTACK

Holds the internal data stack. This segment and its length are normally defined in the linker command file with the following command:

```
-Z(DATA)CSTACK+nn=start
```

where *nn* is the size of the stack specified as a hexadecimal number and *start* is the first memory location.

Linker segment type

DATA

Memory range

This segment can be placed anywhere in memory.

DIFUNCT

Holds the dynamic initialization vector used by Embedded C++.

Linker segment type

CODE

Memory range

This segment can be placed anywhere in memory.

HEAP

Holds dynamically allocated data.

This segment and its length is normally defined in the linker command file by the command:

```
-Z(DATA)HEAP+nn=start
```

where *nn* is the length and *start* is the location.

Linker segment type

DATA

Memory range

This segment can be placed anywhere in memory.

ICODE Holds the start-up code.

Linker segment type

CODE

Memory range

This segment can be placed anywhere in memory.

INTVEC Holds the interrupt vector table generated by the use of the `__interrupt` extended keyword.

Linker segment type

CODE

Memory range

Must start at address `0`.

MEM_A Holds located `__mem` variables.

Linker segment type

DATA

Memory range

This segment can be placed anywhere in memory.

MEM_I Holds initialized `__mem` variables.

Linker segment type

DATA

Memory range

This segment can be placed anywhere in memory.

MEM_ID Holds initializer data for initialized __mem variables.

Linker segment type

CODE

Memory range

This segment can be placed anywhere in memory.

MEM_N Holds __no_init__mem variables.

Linker segment type

DATA

Memory range

This segment can be placed anywhere in memory.

MEM_Z Holds zero-initialized __mem variables.

Linker segment type

DATA

Memory range

This segment can be placed anywhere in memory.

RCODE Holds library code.

Linker segment type

CODE

Memory range

This segment can be placed anywhere in memory.

`SFR_A` Holds located `__sfr` variables.

Linker segment type

`DATA`

Memory range

`0-1FFF`

`SFR_I` Holds initialized `__sfr` variables.

Linker segment type

`DATA`

Memory range

`0-1FFF`

`SFR_ID` Holds initializer data for initialized `__sfr` variables.

Linker segment type

`CODE`

Memory range

This segment can be placed anywhere in memory.

`SFR_N` Holds `__no_init__sfr` variables.

Linker segment type

`DATA`

Memory range

`0-1FFF`

SFR_Z Holds zero-initialized __sfr variables.

Linker segment type

DATA

Memory range

0-1FFF

XMEM_A Holds located __xmem variables.

Linker segment type

DATA

Memory range

This segment can be placed anywhere in memory, except in ymem.

XMEM_I Holds initialized __xmem variables.

Linker segment type

DATA

Memory range

This segment can be placed anywhere in memory, except in ymem.

XMEM_ID Holds initializer data for initialized __xmem variables.

Linker segment type

CODE

Memory range

This segment can be placed anywhere in memory.

XMEM_N Holds __no_init__xmem variables.

Linker segment type

DATA

Memory range

This segment can be placed anywhere in memory, except in ymem.

XMEM_Z Holds zero-initialized __xmem variables.

Linker segment type

DATA

Memory range

This segment can be placed anywhere in memory, except in ymem.

YMEM_A Holds located __ymem variables.

Linker segment type

DATA

Memory range

This segment can be placed anywhere in memory, except in xmem.

YMEM_I Holds initialized __ymem variables.

Linker segment type

DATA

Memory range

This segment can be placed anywhere in memory, except in xmem.

YMEM_ID Holds initializer data for initialized __ymem variables.

Linker segment type

CODE

Memory range

This segment can be placed anywhere in memory.

YMEM_N Holds __no_init__ymem variables.

Linker segment type

DATA

Memory range

This segment can be placed anywhere in memory, except in xmem.

YMEM_Z Holds zero-initialized __ymem variables.

Linker segment type

DATA

Memory range

This segment can be placed anywhere in memory, except in xmem.

Compiler options

This chapter explains how to set the compiler options from the command line, and gives detailed reference information about each option.

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

Setting compiler options

To set compiler options from the command line, include them on the command line after the `iccdspic` command, either before or after the source filename. For example, when compiling the source `prog.c`, use the following command to generate an object file with debug information:

```
iccdspic prog --debug
```

Some options accept a filename, included after the option letter with a separating space. For example, to generate a listing to the file `list.lst`:

```
iccdspic prog -l list.lst
```

Some other options accept a string that is not a filename. The string is included after the option letter, but without a space. For example, to define a symbol:

```
iccdspic prog -DDEBUG=1
```

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

Note that a command line option has a *short* name and/or a *long* name:

- A short option name consists of one character, with or without parameters. You specify it with a single dash, for example `-e`.
- A long name consists of one or several words joined by underscores, and it may have parameters. You specify it with double dashes, for example `--char_is_signed`.

SPECIFYING PARAMETERS

When a parameter is needed for an option with a short name, it can be specified either immediately following the option or as the next command line argument.

For instance, an include file path of `\usr\include` can be specified either as:

```
-I\usr\include
```

or as

```
-I \usr\include
```

Note: `/` can be used instead of `\` as directory delimiter.

Additionally, output file options can take a parameter that is a directory name. The output file will then receive a default name and extension.

When a parameter is needed for an option with a long name, it can be specified either immediately after the equal sign (=) or as the next command line argument, for example:

```
--diag_suppress=Pe0001
```

or

```
--diag_suppress Pe0001
```

The option --`preprocess` is, however, an exception as the filename must be preceded by space. In the following example comments are included in the preprocessor output:

```
--preprocess=c prog
```

Options that accept multiple values may be repeated, and may also have comma-separated values (without space), for example:

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

The current directory is specified with a period (`.`), for example:

```
iccdspic prog -l .
```

A file specified by `'-'` is standard input or output, whichever is appropriate.

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

```
iccdspic prog -l ---r
```

SPECIFYING ENVIRONMENT VARIABLES

Compiler options can also be specified in the `QCCDSPIC` environment variable. The compiler automatically appends the value of this variable to every command line, so it provides a convenient method of specifying options that are required for every compilation.

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

Environment variable	Description
C_INCLUDE	Specifies directories to search for include files; for example: C_INCLUDE=c:\program files\iar systems\embedded workbench 3\dspic\inc;c:\headers
QCCDSPIC	Specifies command line options; for example: QCCDSPIC=-lA asm.lst -z9

Table 18: Environment variables

See the *dsPIC IAR Assembler Reference Guide* for information about the environment variables that can be used by the dsPIC IAR Assembler. See the *IAR Linker and Library Tools Reference Guide* for information about the environment variables that can be used by the IAR XLINK Linker™ and the IAR XLIB Librarian™.

ERROR RETURN CODES

The dsPIC IAR C/EC++ 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 --warnings_affect_exit_code was used.
2	There were non-fatal errors or fatal compilation errors making the compiler abort.
3	There were crashing errors.

Table 19: Error return codes

Options summary

The following table summarizes the compiler command line options:

Command line option	Description
`--char_is_signed`	'char' is 'signed char'
`--cpu={0|1}`	Processor variant
`-D`*`symbol`*`[=`*`value`*`]`	Defines preprocessor symbols
`--data_model=s|l`	Data model

Table 20: Compiler options summary

Command line option	Description
--debug	Generates debug information
--diag_error=*tag*,*tag*,...	Treats these as errors
--diag_remark=*tag*,*tag*,...	Treats these as remarks
--diag_suppress=*tag*,*tag*,...	Suppresses these diagnostics
--diag_warning=*tag*,*tag*,...	Treats these as warnings
-e	Enables language extensions
--ec++	Enables Embedded C ++ syntax
-f *filename*	Extends the command line
-I*path*	Includes file path
--IARStyleMessages	Generates error messages in the IAR standard format
-l[c|C|a|A][N][H] *filename*	Creates list file
--library_module	Makes library module
--migration_preprocessor_extensions	Extends the preprocessor
--module_name=*name*	Sets object module name
--no_code_motion	Disables code motion optimization
--no_cse	Disables common sub-expression elimination
--no_inline	Disables function inlining
--no_unroll	Disables loop unrolling
--no_warnings	Disables all warnings
-o *filename*	Sets object filename
--only_stdout	Uses standard output only
--preprocess=[c][n][l] *filename*	Preprocessor output to file
--public_equ *symbol*[=*value*]	Defines a global named assembler label
-r	Generates debug information
--remarks	Enables remarks
--require_prototypes	Verifies that prototypes are proper
-s[3|6|9]	Optimizes for speed
--silent	Sets silent operation
--strict_ansi	Enables strict ISO/ANSI

Table 20: Compiler options summary (Continued)

Command line option	Description
`-v{0|1}`	Processor variant
`--warnings_affect_exit_code`	Warnings affect exit code
`--warnings_are_errors`	Treats all warnings as errors
`-z[3|6|9]`	Optimizes for size
`--64bit_doubles`	Sets size of doubles to 64 bits instead of 32

Table 20: Compiler options summary (Continued)

Descriptions of options

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

`--char_is_signed`

`--char_is_signed`

By default the compiler interprets the `char` type as `unsigned char`. The `--char_is_signed` option causes the compiler to interpret the `char` type as `signed char` 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 since the library uses unsigned chars.

This option corresponds to the **Treat 'char' as 'signed char'** option in the **ICCDSPIC** category in the IAR Embedded Workbench.

`--cpu`

`--cpu={0|1}`
`-v`

Use this option to select the processor for which the code is to be generated. The following dsPIC cores are available:

Processor option	Supported dsPIC core
`-v0` or `--cpu=0`	`DSP instructions`
`-v1` or `--cpu=1`	`No DSP instructions`

Note that to specify the processor, you can use either the `--cpu` option or the `-v` option. For additional information, see *Processor variant*, page 9.

This option is related to the **Processor variant** option in the **General** category in the IAR Embedded Workbench.

`-D`

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

Use this option to define a preprocessor symbol with the name *`symbol`* and the value *`value`*. If no value is specified, `1` is used.

The option `-D` has the same effect as a `#define` statement at the top of the source file:

```
-Dsymbol
```

is equivalent to:

```
#define symbol
```

Example

You may want to arrange your source to produce either the test or production version of your program depending on whether the symbol `testver` was defined. To do this you would use include sections such as:

```
#ifdef  testver
  ...  ; additional code lines for test version only
#endif
```

Then, you would select the version required on the command line as follows:

Production version: `iccdspic prog`

Test version: `iccdspic prog -Dtestver`

This option can be used one or more times on the command line.

This option is related to the **Preprocessor** options in the **ICCDSPIC** category in the IAR Embedded Workbench.

`--data_model`

```
--data_model {s|l}
```

Use this option to select the data model for which the code is to be generated:

Data model	Default memory attribute	Default data pointer
`s` (small)	`__mem`	`__mem`
`l` (large) (default)	`__mem`	`__ptr`

Table 21: Available data models

If you do not include any of the data model options, the compiler uses the large data model as default.

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

Example

For example, use the following command to specify the large data model:

```
--data_model l
```

To set the equivalent option in the IAR Embedded Workbench, select **Large** on the **Project>Options>General>Target** option page.

`--debug, -r`

```
--debug
-r
```

Use the `--debug` or the `-r` option to make the compiler include information required by C-SPY™ and other symbolic debuggers in the object modules.

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

This option is related to the **Output** options in the **ICCDSPIC** category in the IAR Embedded Workbench.

`--diag_error`

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

Use this option to classify diagnostic messages as errors. An error indicates a violation of the C or Embedded C++ language rules, of such severity that object code will not be generated, and the exit code will not be 0.

Example

The following example classifies warning `Pe117` as an error:

```
--diag_error=Pe117
```

This option is related to the **Diagnostics** options in the **ICCDSPIC** category in the IAR Embedded Workbench.

`--diag_remark`

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

Use this option to classify diagnostic messages as remarks. A remark is the least severe type of diagnostic message and indicates a source code construct that may cause strange behavior in the generated code.

Example

The following example classifies the warning `Pe177` as a remark:

```
--diag_remark=Pe177
```

This option is related to the **Diagnostics** options in the **ICCDSPIC** category in the IAR Embedded Workbench.

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

Use this option to suppress diagnostic messages.

Example

The following example suppresses the warnings `Pe117` and `Pe177`:

```
--diag_suppress=Pe117,Pe177
```

This option is related to the **Diagnostics** options in the **ICCDSPIC** category in the IAR Embedded Workbench.

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

Use this option to classify diagnostic messages as warnings. A warning indicates an error or omission that is of concern, but which will not cause the compiler to stop before compilation is completed.

Example

The following example classifies the remark `Pe826` as a warning:

```
--diag_warning=Pe826
```

This option is related to the **Diagnostics** options in the **ICCDSPIC** category in the IAR Embedded Workbench.

`-e` `-e`

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

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

For additional information, see *IAR language extension overview, page 4.*

This option is related to the **Language** options in the **ICCDSPIC** category in the IAR Embedded Workbench.

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

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

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>ICCDSPIC>Language**. Note that in the IAR Embedded Workbench, EC++ is enabled by default.

-f `-f` *`filename`*

Reads command line options from the named file, with the default extension `xcl`.

By default the compiler accepts command parameters only from the command line itself and the `QCCDSPIC` environment variable. To make long command lines more manageable, and to avoid any operating system command line length limit, you use the `-f` option to specify a command file, from which the compiler reads command line items as if they had been entered at the position of the option.

In the command file, you format the items exactly as if they were on the command line itself, except that you may use multiple lines since the newline character acts just as a space or tab character.

Both C and C++ comments are allowed in the file. Double quotes behave like in DOS.

Example

For example, you could replace the command line:

```
iccdspic prog -r -Dtestver "-Dusername=John Smith"
-Duserid=463760
```

with

```
iccdspic prog -r -Dtestver -f userinfo
```

and the file `userinfo.xcl` containing:

```
"-Dusername=John Smith"
-Duserid=463760
```

-I `-IPath`

Use this option to specify paths for `#include` files. This option may be used more than once on a single command line.

Following is the full description of the compiler's `#include` file search procedure:

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

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

  it searches the following directories for the file to include:

  1. The directories specified with the `-I` option, in the order that they were specified.
  2. The directories specified using the `C_INCLUDE` environment variable, if any.

- When the compiler encounters the name of an `#include` file in double quotes such as:

```
#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. Example:

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

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

```
iccdspic ..\src.c -I..\dir\include
```

  Then the following directories are searched for the `io.h` file, in the following order:

`dir\h`	Current file.
`dir`	File including current file.
`dir\include`	As specified with the `-I` option.

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

Example

```
iccdspic prog -I\mylib1
```

Note: Both \ and / can be used as directory delimiters.

This option is related to the **Preprocessor** options in the **ICCDSPIC** category in the IAR Embedded Workbench.

--IARStyleMessages

`--IARStyleMessages`

Use this option to generate error messages in the IAR standard format. The default is off, which uses the Microchip format.

-l

`-l[c|C|a|A][N][H] filename`

By default the compiler does not generate a listing. Use this option to generate a listing to the file *filename* with the default extension `lst`.

The following modifiers are available:

Option modifier	Description
a	Assembler file
A (N is implied)	Assembler file with C or Embedded C++ source as comments
c	C or Embedded C++ list file
C (default)	C or Embedded C++ list file with assembler source as comments
N	No diagnostics in file
H	Includes source lines from header files in output. Without this option only source lines from the primary source file are included.

Table 22: Generating a compiler list file (-l)

Example

To generate a listing to the file `list.lst`, use:

```
iccdspic prog -l list
```

This option is related to the **List** options in the **ICCDSPIC** category in the IAR Embedded Workbench.

--library_module

`--library_module`

Use this option to make the compiler treat the object file as a library module rather than as 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.

This option is related to the **Output** options in the **ICCDSPIC** category in the IAR Embedded Workbench.

--migration_preprocessor_extensions

`--migration_preprocessor_extensions`

Migration preprocessor extensions extend the preprocessor in order to ease migration of code from earlier IAR compilers. The preprocessor extensions include:

- The availability of floating point in preprocessor expressions.
- The availability of basic type names and `sizeof` in preprocessor expressions.
- The availability of all symbol names (including typedefs and variables) in preprocessor expressions.

If you need to migrate code from an earlier IAR C/EC++ Compiler, you may want to enable these preprocessor extensions.

Note: If you use this option, not only will the compiler accept code that is not standard conformant, but it will also reject some code that *does* conform to standard.

Important! Do not depend on these extensions in newly written code. Support for them may be removed in future compiler versions.

--module_name

`--module_name=`*`name`*

Normally, the internal name of the object module is the name of the source file, without a directory name or extension. Use this option to specify an object module name.

To set the object module name explicitly, use the option `--module_name=`*`name`*, for example:

```
iccdspic prog --module_name=main
```

This option is particularly useful when several modules have the same filename, since the resulting duplicate module name would normally cause a linker error; for example, when the source file is a temporary file generated by a preprocessor.

Example

The following example—in which `%1` is an operating system variable containing the name of the source file—will give duplicate name errors from the linker:

```
preproc %1.c temp.c             ; preprocess source,
                                ; generating temp.c
iccdspic temp.c                 ; module name is
                                ; always 'temp'
```

To avoid this, use `--module_name=`*`name`* to retain the original name:

```
preproc %1.c temp.c                ; preprocess source,
                                   ; generating temp.c
iccdspic temp.c --module_name=%1  ; use original source
                                   ; name as module name
```

Note: In the above example, `preproc` is an external utility.

This option is related to the **Output** options in the **ICCDSPIC** category in the IAR Embedded Workbench.

--no_code_motion

`--no_code_motion`

Use this option to disable optimizations that move code. These optimizations, which are performed at optimization levels 6 and 9, normally reduce code size and execution time. The resulting code may however be difficult to debug.

Note: This option has no effect at optimization level 3.

This option is related to the **Code** options in the **ICCDSPIC** category in the IAR Embedded Workbench.

--no_cse

`--no_cse`

Use `--no_cse` to disable common sub-expression elimination.

On optimization levels 6 and 9, the compiler avoids calculating the same expresson more than once. This optimization normally reduces both code size and execution time. The resulting code may however be difficult to debug.

Note: This option has no effect at optimization level 3.

This option is related to the **Code** options in the **ICCDSPIC** category in the IAR Embedded Workbench.

--no_inline

`--no_inline`

Use `--no_inline` to disable function inlining.

Function inlining means that a simple function, whose definition is known at compile time, is integrated into the body of its caller to eliminate the overhead of the call.

This optimization, which is performed at optimization level 9, normally reduces the execution time, but increases the code size. The resulting code may also be difficult to debug. In certain cases, the code size will decrease when this option is used.

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

Note: This option has no effect at optimization levels 3 and 6.

This option is related to the **Code** options in the **ICCDSPIC** category in the IAR Embedded Workbench.

--no_unroll

`--no_unroll`

Use this option to disable loop unrolling.

The code body of a small loop, whose number of iterations can be determined at compile time, is duplicated to reduce the loop overhead.

For small loops, the overhead required to perform the looping can be large compared to 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, for example the instruction scheduler.

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. This option has no effect at optimization levels 3 and 6.

This option is related to the **Code** options in the **ICCDSPIC** category in the IAR Embedded Workbench.

--no_warnings

`--no_warnings`

By default the compiler issues standard warning messages. Use this option to disable all warning messages.

This option is related to the **Diagnostics** options in the **ICCDSPIC** category in the IAR Embedded Workbench.

-o

`-o` *`filename`*

Use the `-o` option to specify a name for the output file. The filename may include a pathname.

If no object code filename is specified, the compiler stores the object code in a file whose name consists of the source filename, excluding the path, plus the filename extension `r59`.

Example

To store the compiler output in a file called `obj.r59` in the `mypath` directory, you would use:

```
iccdspic prog -o \mypath\obj
```

Note: Both `\` and `/` can be used as directory delimiters.

This option is related to the **Output Directories** options in the **General** category in the IAR Embedded Workbench.

`--only_stdout` `--only_stdout`

Use this option to make the compiler use `stdout` also for messages that are normally directed to `stderr`.

`--preprocess` `--preprocess=[c][n][l] filename`

Use this option to direct preprocessor output to the named file, *`filename`*`.i`.

The filename consists of the filename itself, optionally preceded by a path name and optionally followed by an extension. If no extension is given, the extension `i` is used. In the syntax description above, note that a space is allowed in front of the filename.

The following table shows the mapping of the available preprocessor modifiers:

Command line option	Description
`--preprocess=c`	Preserve comments
`--preprocess=n`	Preprocess only
`--preprocess=l`	Generate `#line` directives

Table 23: Directing preprocessor output to file (--preprocess)

This option is related to the **Preprocessor** options in the **ICCDSPIC** category in the IAR Embedded Workbench.

`--public_equ` `--public_equ symbol[=value]`

This option is equivalent to defining a label in assembler language by using the `EQU` directive and exporting it using the `PUBLIC` directive.

-r, --debug

```
-r
--debug
```

Use this option to make the compiler include information required by the IAR C-SPY Debugger and other symbolic debuggers in the object modules.

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

This option is related to the **Output** options in the **ICCDSPIC** category in the IAR Embedded Workbench.

--remarks

```
--remarks
```

The least severe diagnostic messages are called remarks (see *Severity levels*, page 139). 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.

This option is related to the **Diagnostics** options in the **ICCDSPIC** category in the IAR Embedded Workbench.

--require_prototypes

```
--require_prototypes
```

This option forces the compiler to verify that all functions have proper prototypes. Using this option means that code containing any of the following will generate an error:

- A function call of a function with no declaration or with a Kernighan & Ritchie C declaration
- A function definition of a public function with no previous prototype declaration
- An indirect function call through a function pointer with a type that does not include a prototype.

-s

```
-s[3|6|9]
```

Use this option to make the compiler optimize the code for maximum execution speed.

If no optimization option is specified, the compiler will use the speed optimization `-s3` by default. If the `-s` option is used without specifying the optimization level, level 3 is used by default. Values other than 3, 6, or 9 are rounded down to the closest of those numbers, except 0-2, which are rounded up to 3.

Note: The `-s` and `-z` options cannot be used at the same time.

The following table shows how the optimization levels are mapped:

Option modifier	Description
3	Fully debuggable
6	Heavy optimization can make the program flow hard to follow during debug
9	Full optimization

Table 24: Specifying speed optimization (-s)

This option is related to the **Optimization** options in the **ICCDSPIC** category in the IAR Embedded Workbench.

--silent

`--silent`

By default the compiler issues introductory messages and a final statistics report. Use `--silent` to make the compiler operate without sending these messages to the standard output stream (normally the screen).

This option does not affect the display of error and warning messages.

--strict_ansi

`--strict_ansi`

By default the compiler accepts a superset of ISO/ANSI C (see the chapter *IAR C extensions*). Use `--strict_ansi` to ensure that the program conforms to the ISO/ANSI C standard.

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

This option is related to the **Language** options in the **ICCDSPIC** category in the IAR Embedded Workbench.

For a list of the predefined symbols, see the chapter *Predefined symbols*, page 117.

-v

```
-v{0|1}
--cpu={0|1}
```

Use this option to select the processor variants for which the code is to be generated. The following dsPIC cores are available:

Processor option	Supported dsPIC core
`-v0` or `--cpu=0`	`DSP instructions`
`-v1` or `--cpu=1`	`No DSP instructions`

Table 25: Mapping of processor options

See also *Processor variant*, page 9.

This option is related to the **CPU variant** option in the **General** category in the IAR Embedded Workbench.

--warnings_affect_exit_code

`--warnings_affect_exit_code`

By default the exit code is not affected by warnings, only errors produce a non-zero exit code. With this option, warnings will generate a non-zero exit code.

This option is related to the **Diagnostics** options in the **ICCDSPIC** category in the IAR Embedded Workbench.

--warnings_are_errors

`--warnings_are_errors`

Use this option to make the compiler treat all warnings as errors. If the compiler encounters an error, no object code is generated. Warnings that have been changed into remarks are not treated as errors.

Note: Any diagnostic messages that have been reclassified as warnings by the compiler option `--diag_warning` or the `#pragma diag_warning` directive will also be treated as errors when `--warnings_are_errors` is used.

For additional information, see *--diag_warning*, page 90, and *#pragma diag_warning*, page 112.

To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>ICCDSPIC>Diagnostics**.

-z

`-z[3|6|9]`

Use this option to make the compiler optimize the code for minimum size. If no optimization option is specified, `-z3` is used by default. Values other than 3, 6, or 9 are rounded up to the closest of those numbers, except 10 or above, which are rounded down to 9.

The following table shows how the optimization levels are mapped:

Option modifier	Description
3	Fully debuggable
6	Heavy optimization can make the program flow difficult to follow during debug
9	Full optimization

Table 26: Specifying size optimization (-z)

Note: The `-s` and `-z` options cannot be used at the same time.

This option is related to the **Optimization** options in the **ICCDSPIC** category in the IAR Embedded Workbench.

`--64bit_doubles`

`--64bit_doubles`

Use this option to force the compiler to use 64-bit doubles instead of 32-bit doubles which is the default.

This option is related to the **Target** options in the **General** category in the IAR Embedded Workbench.

Extended keywords

This chapter describes the extended keywords that support specific features of the dsPIC microcontroller, the general syntax rules for the keywords, and a detailed description of each keyword.

For information about the address ranges for the different memory areas, see the chapter *Segment reference*.

Summary of extended keywords

The following table summarizes the extended keywords that are available to the dsPIC IAR C/EC++ Compiler:

Extended keyword	Description	Type
`__constptr`	Const memory attribute	--
`__func`	Function pointer attribute	--
`__interrupt`	Supports interrupt functions	Special function type
`__intrinsic`	Reserved for compiler internal use only	--
`__mem`	Data memory attribute	--
`__monitor`	Supports atomic execution of a function	Function execution
`__no_init`	Supports non-volatile memory	Data storage
`__ptr`	Generic pointer	--
`__root`	Ensures that a function or variable is included in the object code even if unused	--
`__sfr`	Data memory attribute for variables placed in sfr memory	--
`__xmem`	Data memory attribute for variables placed in xmem memory	--
`__ymem`	Data memory attribute for variables placed in ymem memory	--

Table 27: Extended keywords summary

Using extended keywords

This section covers how extended keywords can be used when declaring and defining data and functions. The syntax rules for extended keywords are also described.

In addition to the rules presented here—to place the keyword directly in the code—the directives #pragma type_attribute and #pragma object_attribute can be used for specifying the keywords. Refer to the chapter *#pragma directives* for details about how to use the extended keywords together with #pragma directives.

The keywords and the @ operator are only available when language extensions are enabled in the dsPIC IAR C/EC++ Compiler.

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

Use the -e compiler option to enable language extensions. See *-e*, page 90 for additional information.

DATA STORAGE

The extended keywords that can be used for data can be divided into two groups that control the following:

- The memory type of objects and pointers (__constptr, __mem __sfr, __xmem, and __ymem).
- Other characteristics of objects (__root and __no_init).

See the chapter *Data storage* in *Part 2: Compiler reference* for more information about memory types.

Syntax

The keywords follow the same syntax as the type qualifiers const and volatile. The following declarations all place the variable i and j in xmem memory:

```
__xmem int i, j;
int __xmem i, j;
```

Notice that the keyword affects all the identifiers.

Pointers

A keyword that is followed by an asterisk (*), affects the type of the pointer being declared. A pointer to sfr memory is thus declared by:

```
char __sfr * p;
```

Notice that the location of the pointer variable p is not affected by the keyword. In the following example, however, the pointer variable p2 is placed in xmem memory. Like p, p2 points to a character in sfr memory.

```
__xmem char __sfr *p2;
```

Type definitions

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

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

and

```
__xmem char b;
char __xmem *bp;
```

FUNCTIONS

The extended keywords that can be used when functions are declared can be divided into two groups:

- Keywords that control the type of the functions. Keywords of this group must be specified both when the function is declared and when it is defined (`__interrupt` and `__monitor`).
- Keywords that only control the function object defined (`__root`).

Syntax

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

```
__interrupt void alpha(void);
```

The keywords that are *type* attributes must be specified both when they are defined and in the declaration. *Object* attributes only have to be specified when they are defined since they do not affect the way an object or function is used.

Descriptions of extended keywords

The following sections give detailed information about each extended keyword.

`__constptr`

The `__constptr` keyword is the data pointer for constants placed in code memory.

`__func`

`__func` is the function pointer keyword. It is the only function pointer keyword, and may be omitted.

`__interrupt` The `__interrupt` keyword specifies interrupt functions. The `#pragma vector` directive can be used for specifying the interrupt vector. An interrupt function must have a `void` return type and cannot have any parameters.

The following example declares an interrupt function with interrupt vector with offset `0x70` in the `INTVEC` segment:

```
#pragma vector=0x70
__interrupt void my_interrupt_handler(void);
```

An interrupt function cannot be called directly from a C program. It can only be executed as a response to an interrupt request.

It is possible to define an interrupt function without a vector, but then the compiler will not generate an entry in the interrupt vector table. For additional information, see *INTVEC*, page 76.

The range of the interrupt vectors depends on the device used. The `iochip.h` header file, which corresponds to the selected derivative, contains predefined names for the existing interrupt vectors.

`__intrinsic` The `__intrinsic` keyword is reserved for compiler internal use only.

`__mem` The `__mem` keyword is the data memory attribute for variables placed anywhere in the data memory.

`__monitor` The `__monitor` keyword causes interrupts to be disabled during execution of the function. This allows atomic operations to be performed, such as operations on semaphores that control access to resources by multiple processes. A function declared with the `__monitor` keyword is equivalent to any other function in all other respects.

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

For additional information, see the intrinsic functions *__disable_interrupt*, page 124, and *__enable_interrupt*, page 124.

Example

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 to simultaneously use resources that can only be used by one process at a time, for example a printer.

```
/* When the_lock is non-zero, someone owns the lock. */
static 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();
  }
}
```

`__no_init` The `__no_init` keyword is used for placing a variable in a non-volatile memory segment and for suppressing initialization at startup.

The __no_init keyword is placed in front of the type, for instance to place settings in non-volatile memory:

```
__no_init int settings[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 settings[10];
```

Note: The __no_init keyword cannot be used in typedefs.

__ptr

The __ptr keyword is a generic pointer that can point to both code and data memory.

__root

The __root attribute can be used on either a function or a variable to ensure that, when the module containing the function or variable is linked, the function or variable is also included, whether or not it is referenced by the rest of the program.

By default only the part of the runtime library calling main and any interrupt vectors are root. All other functions and variables are included in the linked output only if they are referenced by the rest of the program.

The __root keyword is placed in front of the type, for example to place settings in non-volatile memory:

```
__root int settings[10];
```

The #pragma object_attribute directive can also be used. The following declaration is equivalent to the previous one:

```
#pragma object_attribute=__root
int settings[10];
```

Note: The __root keyword cannot be used in typedefs.

__sfr

The __sfr keyword is the data memory attribute for variables placed in sfr memory.

__xmem

The __xmem keyword is the data memory attribute for variables placed in xmem memory.

__ymem

The __ymem keyword is the data memory attribute for variables placed in ymem memory.

#pragma directives

This chapter describes the #pragma directives of the dsPIC IAR C/EC++ Compiler.

The #pragma directives control the behavior of the compiler, for example, how it allocates memory, whether it allows extended keywords, and whether it outputs warning messages. The #pragma directives are preprocessed, which means that macros are substituted in a #pragma directive.

The #pragma directives are always enabled in the dsPIC IAR C/EC++ Compiler. They are consistent with the ISO/ANSI C and are very useful when you want to make sure that the source code is portable.

Summary of #pragma directives

The following table shows the `#pragma` directives of the compiler:

#pragma directive	Description
#pragma bitfields	Controls the order of bitfield members
#pragma constseg	Places constant variables in a named segment
#pragma dataseg	Places variables in a named segment
#pragma diag_default	Changes the severity level of diagnostic messages
#pragma diag_error	Changes the severity level of diagnostic messages
#pragma diag_remark	Changes the severity level of diagnostic messages
#pragma diag_suppress	Suppresses diagnostic messages
#pragma diag_warning	Changes the severity level of diagnostic messages
#pragma inline	Places function code inline
#pragma language	Controls the IAR language extensions
#pragma location	Specifies the absolute address of a variable
#pragma object_attribute	Changes the definition of a variable or a function
#pragma optimize	Specifies type and level of optimization
#pragma rtmodel	Inserts a runtime model attribute

Table 28: #pragma directives summary

#pragma directive	Description
#pragma type_attribute	Changes the declaration and definitions of a variable or a function
#pragma vector	Specifies the vector of an interrupt function

Table 28: #pragma directives summary (Continued)

Note: For portability reasons, the `#pragma` directives `alignment`, `codeseg`, `function`, `memory`, and `warnings`, are recognized and will give a diagnostic message. It is important to be aware of this if you need to port existing code that contains any of those `#pragma` directives.

Descriptions of #pragma directives

This section gives detailed information about each `#pragma` directive.

All `#pragma` directives should be entered like:

#pragma *pragmaname*=*pragmavalue*

or

#pragma *pragmaname* = *pragmavalue*

#pragma bitfields

`#pragma bitfields={reversed|default}`

The `#pragma bitfields` directive controls the order of bitfield members.

By default the dsPIC IAR C/EC++ Compiler places bitfield members from the least significant bit to the most significant bit in the container type. Use the `#pragma bitfields=reversed` directive to place the bitfield members from the most significant to the least significant bit. This setting remains active until you turn it off again with the `#pragma bitfields=default` directive.

#pragma constseg

The `#pragma constseg` directive places constant variables in a named segment. Use the following syntax:

```
#pragma constseg=MY_CONSTANTS
const int factorySettings[] = {42, 15, -128, 0};
#pragma constseg=default
```

The segment name must not be a predefined segment; see the chapter *Segment reference* for more information.

The memory in which the segment resides is optionally specified using the following syntax:

```
#pragma constseg=__sfr MyOtherSeg
```

All constants defined following this directive will be placed in the segment `MyOtherSeg` and accessed using sfr addressing.

#pragma dataseg

The `#pragma dataseg` directive places variables in a named segment. Use the following syntax:

```
#pragma dataseg=MY_SEGMENT
__no_init char myBuffer[1000];
#pragma dataseg=default
```

The segment name must not be a predefined segment, see the chapter *Segment reference* for more information. The variable `myBuffer` will not be initialized at startup and must thus not have any initializer.

The memory in which the segment resides is optionally specified using the following syntax:

```
#pragma dataseg=__sfr MyOtherSeg
```

All variables in `MyOtherSeg` will be accessed using sfr addressing.

#pragma diag_default

`#pragma diag_default=`*`tag`*`,`*`tag`*`,...`

Changes the severity level back to default or as defined on the command line for the diagnostic messages with the specified tags. See the chapter *Diagnostics* for more information about diagnostic messages.

Example

```
#pragma diag_default=Pe117
```

#pragma diag_error

`#pragma diag_error=`*`tag`*`,`*`tag`*`,...`

Changes the severity level to `error` for the specified diagnostics. See the chapter *Diagnostics* for more information about diagnostic messages.

Example

```
#pragma diag_error=Pe117
```

#pragma diag_remark

`#pragma diag_remark=`*tag*`,`*tag*`,...`

Changes the severity level to `remark` for the specified diagnostics. For example:

```
#pragma diag_remark=Pe177
```

See the chapter *Diagnostics* for more information about diagnostic messages.

#pragma diag_suppress

`#pragma diag_suppress=`*tag*`,`*tag*`,...`

Suppresses the diagnostic messages with the specified tags. For example:

```
#pragma diag_suppress=Pe117,Pe177
```

See the chapter *Diagnostics* for more information about diagnostic messages.

#pragma diag_warning

`#pragma diag_warning=`*tag*`,`*tag*`,...`

Changes the severity level to `warning` for the specified diagnostics. For example:

```
#pragma diag_warning=Pe826
```

See the chapter *Diagnostics* for more information about diagnostic messages.

#pragma inline

`#pragma inline[=forced]`

The `#pragma inline` directive advises the compiler that the function whose declaration follows immediately after the directive should be inlined—that is, expanded into the body of the calling function. Whether the inlining actually takes place is subject to the compiler's heuristics.

This is similar to the C++ keyword `inline`, but has the advantage of being available in C code.

Specifying `#pragma inline=forced` disables the compiler's heuristics and forces the inlining. If the inlining fails for some reason, for example if it cannot be used with the function type in question—like `printf`—an error message is emitted.

#pragma language

`#pragma language={extended|default}`

The `#pragma language` directive is used for turning on the IAR language extensions or for using the language settings specified on the command line:

`extended`	Turns on the IAR language extensions and turns off the `--strict_ansi` command line option.
`default`	Uses the settings specified on the command line.

`#pragma location`

```
#pragma location=address
```

The `#pragma location` directive specifices the location—the absolute address—of the variable whose declaration follows the `#pragma` directive. For example:

```
#pragma location=0x10FF00
char PORT1; /* PORT1D is located at address 0x10FF00 */
```

The directive can also take a string specifying the segment placement for either a variable or a function, for example:

```
#pragma location="foo"
```

For additional information and examples, see *Absolute location placement*, page 21 and *Segment placement*, page 21.

`#pragma object_attribute`

```
#pragma object_attribute=keyword
```

The `#pragma object_attribute` directive affects the declaration of the identifier that follows immediately after the directive.

The following keyword can be used with `#pragma object_attribute` for a variable:

`__no_init`	Places a variable in a non-volatile memory segment and suppresses initialization at startup.

The following keyword can be used with `#pragma object_attribute` for a function or variable:

`__root`	Ensures that a function or data object is included in the object code even if not referenced.

Example

In the following example, the variable `bar` is placed in the non-initialized segment:

```
#pragma object_attribute=__no_init
char bar;
```

Unlike the directive `#pragma type_attribute` that specifies the storing and accessing of a variable, it is not necessary to specify an object attribute in declarations. The following example declares `bar` without a `#pragma object_attribute`:

```
__no_init char bar;
```

`#pragma optimize`

`#pragma optimize=`*`token token token`*

where *`token`* is one or more of the following:

`s`	Optimizes for speed
`z`	Optimizes for size
`3|6|9`	Specifies level of optimization
`no_cse`	Turns off common sub-expression elimination
`no_inline`	Turns off function inlining
`no_unroll`	Turns off loop unrolling
`no_code_motion`	Turns off code motion.

The `#pragma optimize` directive is used for decreasing the optimization level or for turning off some specific optimizations. This `#pragma` directive only affects the function that follows immediately after the directive.

Notice 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.

Note: If you use the `#pragma optimize` directive to specify an optimization level that is higher than the optimization level you specify using a compiler option, the `#pragma` directive is ignored.

Example

```
#pragma optimize=s 9
int small_and_used_often()
{
    ...
}

#pragma optimize=z 9
int big_and_seldom_used()
{
    ...
}
```

`#pragma rtmodel`

`#pragma rtmodel("`*`key`*`","`*`value`*`")`

The `#pragma rtmodel` directive inserts the runtime model attribute *`key`* with the value *`value`*. It must be followed by a variable, since the pragma directive is associated with a variable. Keys beginning with __ are reserved by the compiler.

```
#pragma rtmodel("myattr","blue")
char is_blue=1;
```

The runtime model attribute is then passed to the linker. If the same key is found in another file, it must have the same value, otherwise it will not link. See *RTMODEL* in the *dsPIC IAR Assembler Referrence Guide* for a more detailed explanation.

`#pragma type_attribute` `#pragma type_attribute=`*`keyword`*

The `#pragma type_attribute` directive affects the declaration of the identifier, the next variable, or the next function, that follows immediately after the `#pragma` directive. It only affects the variable, not its type.

The following keywords can be used with the `#pragma type_attribute` directive for a variable:

`__constptr`	Places a variable in constptr memory, but only for const-declared variables
`__mem`	Places a variable in mem memory
`__sfr`	Places a variable in sfr memory
`__xmem`	Places a variable in xmem memory
`__ymem`	Places a variable in ymem memory

The following keywords can be used with `#pragma type_attribute` directive for a function:

`__interrupt`	Specifies interrupt functions. Use the `#pragma vector` directive to specify the interrupt vector; see page 116.
`__monitor`	Specifies a monitor function

Example

In the following example, `myBuffer` is placed in xmem memory, whereas the variable `i` is not affected by the `#pragma` directive.

```
#pragma type_attribute=__xmem
char inBuffer[10];
int i;
```

The following declarations, which use extended keywords, are equivalent. See the chapter *Extended keywords* for more details.

```
__xmem char inBuffer[10];
int i;
```

In the small memory model, the default pointer is __mem. In the following example, the pointer is located in sfr memory, pointing at __mem:

```
#pragma type_attribute=__sfr
int * pointer;
```

#pragma vector

`#pragma vector=vector`

The #pragma vector directive specifies the vector of a interrupt function whose declaration follows the #pragma directive.

Example

```
#pragma vector=0x70
__interrupt void my_handler(void);
```

Predefined symbols

This chapter gives reference information about the predefined preprocessor symbols that are supported in the dsPIC IAR C/EC++ Compiler. These symbols allow you to inspect the compile-time environment, for example the time and date of compilation.

Summary of predefined symbols

The following table summarizes the predefined symbols:

Predefined symbol	Description
`__cplusplus`	Determines whether the compiler runs in EC++ mode
`__CPU__`	Identifies the processor variant in use
`__DATA_MODEL__`	Identifies the memory model in use
`__DATE__`	Determines the date of compilation
`__DOUBLE_SIZE__`	Determines the size in bytes (4 or 8)
`__embedded_cplusplus`	Determines whether the compiler runs in EC++ mode
`__FILE__`	Identifies the name of the file being compiled
`__FLOAT_SIZE__`	Determines the size in bytes (4)
`__IAR_SYSTEMS_ICC__`	Identifies the IAR compiler platform
`__ICCDSPIC__`	Identifies the dsPIC IAR C/EC++ Compiler
`__LINE__`	Determines the current source line number
`__LONG_DOUBLE_SIZE__`	Determines the size in bytes (4 or 8)
`__STDC__`	Identifies ISO/ANSI Standard C
`__STDC_VERSION__`	Identifies the version of ISO/ANSI Standard C in use
`__TID__`	Identifies the target processor of the IAR compiler in use
`__TIME__`	Determines the time of compilation
`__VER__`	Identifies the version number of the IAR compiler in use

Table 29: Predefined symbols summary

Descriptions of predefined symbols

The following section gives reference information about each predefined symbol.

`__cplusplus`

This predefined symbol expands to the number 1 when the compiler runs in Embedded C++ mode. When the compiler runs in ANSI C mode, the symbol is undefined.

This symbol can be used with `#ifdef` to detect that the compiler accepts Embedded C++ code. It is particularly useful when creating header files that are to be shared by C and Embedded C++ code.

`__CPU__`

This predefined symbol identifies the processor variant. It expands to a number which corresponds to the processor option in use, `0` for processor option `-v0` or `1` for processor option `-v1`.

`__DATA_MODEL__`

This predefined symbol expands to a value reflecting the selected memory model according to the following table:

Value	Memory model
0	large
1	small

Table 30: Inspecting the data model using predefined symbols

`__DATE__`

Use this symbol to determine when the file was compiled. This symbol expands to the date of compilation which is returned in the form `Mmm dd yyyy`.

`__DOUBLE_SIZE__`

This predefined symbol sets the size in bytes to 4 or 8.

`__embedded_cplusplus`

This predefined symbol expands to the number 1 when the compiler runs in Embedded C++ mode. When the compiler runs in ANSI C mode, the symbol is undefined.

This symbol can be used with `#ifdef` to detect that the compiler accepts only the Embedded C++ subset of the C++ language.

`__FILE__`

Use this symbol to determine which file is currently being compiled. This symbol expands to the name of that file.

`__FLOAT_SIZE__`

This predefined symbol sets the size in bytes to 4.

`__IAR_SYSTEMS_ICC__`

This predefined symbol expands to a number that identifies the IAR compiler platform. The current identifier is 5. Notice that the number could be higher in a future version of the product.

This symbol can be tested with `#ifdef` to detect that the code was compiled by an IAR Compiler.

`__ICCDSPIC__`

This predefined symbol expands to the number 1 when the code is compiled with the dsPIC IAR C/EC++ Compiler.

`__LINE__`

This predefined symbol expands to the current line number of the file currently being compiled.

`__LONG_DOUBLE_SIZE__`

This predefined symbol sets the size in bytes to 4 or 8.

`__STDC__`

This predefined symbol expands to the number 1. This symbol can be tested with `#ifdef` to detect that the compiler in use adheres to ANSI C.

`__STDC_VERSION__`

ISO/ANSI Standard C and version identifier.

This predefined symbol expands to the number 199409L.

Note: This predefined symbol does not apply to the EC++ version of the product.

`__TID__`

Target identifier for the dsPIC IAR C/EC++ Compiler.

Expands to the target identifier containing the following parts:

- A number unique for each IAR compiler (i.e. unique for each target).
- The value of the `-v` option. For details, see *Processor variant*, page 9.
- An intrinsic flag. This flag is set for dsPIC because the compiler supports intrinsic functions.

The `__TID__` value is constructed as:

```
((i << 15) | (t << 8) | (c << 4))
```

You can extract the values as follows:

```
i = (__TID__ >> 15) & 0x01;  /* intrinsic flag */
t = (__TID__ >> 8) & 0x7F;   /* target identifier */
c = (__TID__ >> 4) & 0x0F;   /* cpu option */
```

To find the value of the target identifier for the current compiler, execute:

```
printf("%ld",(__TID__ >> 8) & 0x7F)
```

For the dsPIC microcontroller, the target identifier is `59`.

Note: The use of `__TID__` is not recommended. We recommend you use the symbols `__ICCDSPIC__`, `__DATA_MODEL__`, and `__CPU__` instead.

`__TIME__` Current time.

Expands to the time of compilation in the form `hh:mm:ss`.

`__VER__` Compiler version number.

Expands to an integer representing the version number of the compiler.

Example

The example below prints a message for version 3.34.

```
#if __VER__ == 334
#pragma message "Compiler version 3.34"
#endif
```

Intrinsic functions

This chapter gives reference information about the intrinsic functions.

The intrinsic functions provide direct access to low-level processor operations and can be very useful in, for example, time-critical routines. The intrinsic functions compile into in-line code, either as a single instruction or as a short sequence of instructions.

Intrinsic functions summary

There are two types of intrinsic functions: DSP-related intrinsic functions, and general intrinsic functions.

DSP-RELATED INTRINSIC FUNCTIONS

DSP-related intrinsic functions are accessed using macros located in the file `gsm.h` which in turn includes `indsp.h`. It also defines the signed integer types `int16_t`, `int32_t` and `int40_t`.

The following table summarizes the DSP-related intrinsic functions:

Intrinsic function	Description
`__int16_t abs_s (__int16_t var1)`	Saturated Q15 absolute value operation
`__int16_t add (__int16_t var1, __int16_t var2)`	Saturated Q15 addition
`__int16_t div_s (__int16_t var1, __int16_t var2)`	Saturated Q15 division
`__int16_t extract_h (__int32_t L_var1)`	Truncate Q31 -> Q15
`__int16_t extract_l (__int32_t L_var1)`	Get low word of Q31
`__int16_t mac_r (__int32_t L_var3, __int16_t var1, __int16_t var2)`	Saturated rounded Q15 multiply and accumulate
`__int16_t msu_r (__int32_t L_var3, __int16_t var1, __int16_t var2)`	Saturated rounded Q15 multiply and subtract

Table 31: DSP-related intrinsic functions summary

Intrinsic function	Description
`__int16_t mult (__int16_t var1, __int16_t var2)`	Saturated Q15*Q15->Q15 multiplication
`__int16_t mult_r (__int16_t var1, __int16_t var2)`	Saturated rounded Q15 multiplication
`__int16_t negate (__int16_t var1)`	Saturated Q15 negation
`__int16_t norm_l (__int32_t L_var1)`	Returns the bit-number of the first zero-bit
`__int16_t norm_s (__int16_t var1)`	Returns the bit-number of the first zero-bit
`__int16_t round (__int32_t L_var1)`	Round(Q31)->Q15
`__int16_t round_ub (__int32_t L_var1)`	Unbiased round(Q31)->Q15
`__int16_t shl (__int16_t var1, ____int16_t var2)`	Saturated Q15 shift left
`__int16_t shr (__int16_t var1, __int16_t var2)`	Saturated Q15 shift right
`__int16_t shr_r (__int16_t var1, __int16_t var2)`	Saturated rounded Q15 shift right
`__int16_t sub (__int16_t var1, __int16_t var2)`	Saturated Q15 subtraction
`__int16_t __xmem * add_br (__int16_t __xmem *base, __int16_t __xmem *ptr, __int16_t step)`	Add bitreverse. Base is base of xmem vector, ptr is the pointer to add to, and step is in bitreverse form and should represent a step of one, since it is also used to calculate size of the vector.
`__int32_t L_abs (__int32_t L_var1)`	Saturated Q31 absolute value operation
`__int32_t L_add (__int32_t L_var1, __int32_t L_var2)`	Saturated Q31 addition
`__int32_t L_deposit_h (__int16_t var1)`	Cast a Q15 value to a Q31
`__int32_t L_deposit_l (__int16_t var1)`	Create a Q31 value where the low word is the Q15 value 'var1' and the upper word is zero

Table 31: DSP-related intrinsic functions summary (Continued)

Intrinsic function	Description
`__int32_t L_mac (__int32_t L_var3, __int16_t var1, __int16_t var2)`	Saturated multiply and accumulate
`__int32_t L_msu (__int32_t L_var3, __int16_t var1, __int16_t var2)`	Saturated multiply and subtract
`__int32_t L_mult (__int16_t var1, __int16_t var2)`	Saturated Q15*Q15->Q31 multiplication
`__int32_t L_negate (__int32_t L_var1)`	Saturated Q31 negation
`__int32_t L_shl (__int32_t L_var1, __int16_t var2)`	Saturated Q31 shift left
`__int32_t L_shr (__int32_t L_var1, __int16_t var2)`	Saturated Q31 shift right
`__int32_t L_shr_r (__int32_t L_var1, __int16_t var2)`	Saturated and rounded Q31 shift right
`__int32_t L_sub (__int32_t L_var1, __int32_t L_var2)`	Saturated Q31 subtraction
`__int40_t LL_add (__int40_t LL_var1, __int40_t LL_var2)`	Saturated Q39 addition
`__int40_t LL_mac (__int40_t LL_var3, __int16_t var1, __int16_t var2)`	Saturated Q39 multiply and accumulate
`__int40_t LL_msu (__int40_t LL_var3, __int16_t var1, __int16_t var2)`	Saturated Q39 multiply and subtract
`__int40_t LL_negate (__int40_t LL_var1)`	Saturated Q39 negation
`__int40_t LL_sub (__int40_t LL_var1, __int40_t LL_var2)`	Saturated Q39 subtraction
`void __mem * add_mod(void __mem * base, __int16_t size, void __mem *ptr, __int16_t step)`	Modulo pointer addition. Base is start of vector, size is size of vector in bytes, ptr is current pointer, and step is the step.

Table 31: DSP-related intrinsic functions summary (Continued)

GENERAL INTRINSIC FUNCTIONS

The following table summarizes the general intrinsic functions:

Intrinsic function	Description
`__asm`	Assembles statements inline
`__clear_watchdog_timer`	Generates a CLRWDT instruction
`__disable_interrupt`	Disables interrupts
`__enable_interrupt`	Enables interrupts
`__no_operation`	Generates a NOP instruction
`__require`	Sets a constant literal
`__reset`	Generates a RESET instruction

Table 32: General intrinsic functions summary

Descriptions of intrinsic functions

To use intrinsic functions in an application, include the header file `intrinsics.h`.

Notice that the intrinsic function names start with double underscores, for example:

`__enable_interrupt`

asm

`void asm(const char *string);`

Assembles and inserts the supplied assembler statement inline. The statement can include instruction mnemonics, register mnemonics, constants, and/or a reference to a global variable. Optimizations depending on control-flow analysis, register contents tracking, etc will be disabled when using this function.

__clear_watchdog_timer

`void __clear_watchdog_timer(void);`

Inserts a `CLRWTD` instruction.

__disable_interrupt

`void __disable_interrupt(void);`

Disables interrupts.

__enable_interrupt

`void __enable_interrupt(void);`

Enables interrupts.

`__int16_t abs_s`

```
__int16_t abs_s (__int16_t var1)
```

Returns a saturated absolute value.

A saturated operation computes just like a normal operation, unless an overflow or underflow occurs. If this should happen, the result will be the highest or lowest possible representable value.

`__int16_t add`

```
__int16_t add (__int16_t var1, __int16_t var2)
```

Generates a saturated addition instruction.

A saturated operation computes just like a normal operation, unless an overflow or underflow occurs. If this should happen, the result will be the highest or lowest possible representable value.

`__int16_t div_s`

```
__int16_t div_s (__int16_t var1, __int16_t var2)
```

Generates a saturated division instruction.

A saturated operation computes just like a normal operation, unless an overflow or underflow occurs. If this should happen, the result will be the highest or lowest possible representable value.

`__int16_t extract_h`

```
__int16_t extract_h (__int32_t L_var1)
```

Extracts the high word of `L_var1`.

`__int16_t extract_l`

```
__int16_t extract_l (__int32_t L_var1)
```

Extracts the low word of `L_var1`.

`__int16_t mac_r`

```
__int16_t mac_r (__int32_t L_var1, __int16_t var2, __int16_t
        var3)
```

Generates a saturated and biased rounded multiply var2*var3 and accumulate instruction.

A saturated operation computes just like a normal operation, unless an overflow or underflow occurs. If this should happen, the result will be the highest or lowest possible representable value.

__int16_t msu_r

```
__int16_t msu_r (__int32_t L_var1, __int16_t var2, __int16_t
        var3)
```

Generates a saturated and biased rounded multiply var2*var3 and subtract instruction.

A saturated operation computes just like a normal operation, unless an overflow or underflow occurs. If this should happen, the result will be the highest or lowest possible representable value.

__int16_t mult

```
__int16_t mult (__int16_t var1, __int16_t var2)
```

Generates a saturated multiplication instruction.

A saturated operation computes just like a normal operation, unless an overflow or underflow occurs. If this should happen, the result will be the highest or lowest possible representable value.

__int16_t mult_r

```
__int16_t mult_r (__int16_t var1, __int16_t var2)
```

Generates a saturated and biased rounded multiplication instruction.

A saturated operation computes just like a normal operation, unless an overflow or underflow occurs. If this should happen, the result will be the highest or lowest possible representable value.

__int16_t negate

```
__int16_t negate (__int16_t var1)
```

Generates a saturated negation instruction.

A saturated operation computes just like a normal operation, unless an overflow or underflow occurs. If this should happen, the result will be the highest or lowest possible representable value.

__int16_t norm_l

```
__int16_t norm_l (__int32_t var1)
```

Returns the number of the first zero-bit from the left.

__int16_t norm_s

```
__int16_t norm_s (__int16_t var1)
```

Returns the number of the first zero-bit from the left.

__int16_t round

```
__int16_t round (__int32_t L_var1)
```

Returns a conventional (biased) rounded value.

__int16_t round_ub

```
__int16_t round_ub (__int32_t L_var1)
```

Returns a convergent (unbiased) rounded value.

__int16_t shl

```
__int16_t shl (__int16_t var1, __int16_t step)
```

Generates a saturated shift left instruction.

A saturated operation computes just like a normal operation, unless an overflow or underflow occurs. If this should happen, the result will be the highest or lowest possible representable value.

__int16_t shr

```
__int16_t shr (__int16_t var1, __int16_t step)
```

Generates a saturated signed shift right instruction.

A saturated operation computes just like a normal operation, unless an overflow or underflow occurs. If this should happen, the result will be the highest or lowest possible representable value.

__int16_t shr_r

```
__int16_t shr_r (__int16_t var1, __int16_t step)
```

Generates a saturated and biased rounded signed shift right instruction.

A saturated operation computes just like a normal operation, unless an overflow or underflow occurs. If this should happen, the result will be the highest or lowest possible representable value.

__int16_t sub

```
__int16_t sub (__int16_t var1, __int16_t var2)
```

Generates a saturated subtraction instruction.

A saturated operation computes just like a normal operation, unless an overflow or underflow occurs. If this should happen, the result will be the highest or lowest possible representable value.

__int16_t __xmem * add_br

```
__int16_t __xmem * add_br (__int16_t __xmem * base,
       __int16_t __xmem * ptr, __int16_t step )
```

Executes a bitreverse addition to `ptr`.

`base` points to the start of the vector, `ptr` is the current pointer, and `step` is 1 bit-reversed. For example, if the size of the vector is 256 words, and you step 1, the bit-reversed step value would be 0x1000000 (128).

__int32_t L_abs

```
__int32_t L_abs (__int32_t var1)
```

Returns a saturated absolute value.

A saturated operation computes just like a normal operation, unless an overflow or underflow occurs. If this should happen, the result will be the highest or lowest possible representable value.

__int32_t L_add

```
__int32_t L_add (__int32_t L_var1, __int32_t L_var2)
```

Generates a Q31 saturated addition instruction.

A saturated operation computes just like a normal operation, unless an overflow or underflow occurs. If this should happen, the result will be the highest or lowest possible representable value.

__int32_t L_deposit_h

```
__int32_t L_deposit_h (__int16_t var1)
```

Returns a Q31 value where the high word is var1 and the low word is zero.

__int32_t L_deposit_l

```
__int32_t L_deposit_l (__int16_t var1)
```

Returns a Q31 value where the low word is var1 and the high word is zero.

__int32_t L_mac

```
__int32_t L_mac (__int32_t L_var1, __int16_t var2, __int16_t
      var3)
```

Generates a saturated multiply var2*var3 and accumulate instruction.

A saturated operation computes just like a normal operation, unless an overflow or underflow occurs. If this should happen, the result will be the highest or lowest possible representable value.

__int32_t L_msu

```
__int32_t L_msu (__int32_t L_var1, __int16_t var2, __int16_t
      var3)
```

Generates a saturated multiply var2*var3 and subtract instruction.

A saturated operation computes just like a normal operation, unless an overflow or underflow occurs. If this should happen, the result will be the highest or lowest possible representable value.

`__int32_t L_mult`

`__int32_t L_mult (__int16_t var1, __int16_t var2)`

Generates a Q31 saturated multiplication instruction.

A saturated operation computes just like a normal operation, unless an overflow or underflow occurs. If this should happen, the result will be the highest or lowest possible representable value.

`__int32_t L_negate`

`__int32_t L_negate (__int32_t L_var1)`

Generates a Q31 saturated negation instruction.

A saturated operation computes just like a normal operation, unless an overflow or underflow occurs. If this should happen, the result will be the highest or lowest possible representable value.

`__int32_t L_shl`

`__int32_t L_shl (__int32_t L_var1, __int16_t step)`

Generates a Q31 saturated shift left instruction.

A saturated operation computes just like a normal operation, unless an overflow or underflow occurs. If this should happen, the result will be the highest or lowest possible representable value.

`__int32_t L_shr`

`__int32_t L_shr (__int32_t L_var1, __int16_t step)`

Generates a Q31 saturated shift right instruction.

A saturated operation computes just like a normal operation, unless an overflow or underflow occurs. If this should happen, the result will be the highest or lowest possible representable value.

`__int32_t L_shr_r`

`__int32_t L_shr_r (__int32_t L_var1, __int16_t step)`

Generates a Q31 biased rounded saturated shift right instruction.

A saturated operation computes just like a normal operation, unless an overflow or underflow occurs. If this should happen, the result will be the highest or lowest possible representable value.

__int32_t L_sub

```
__int32_t L_sub (__int32_t L_var1, __int32_t L_var2)
```

Generates a Q31 saturated subtraction instruction.

A saturated operation computes just like a normal operation, unless an overflow or underflow occurs. If this should happen, the result will be the highest or lowest possible representable value.

__int40_t LL_add

```
__int40_t LL_add (__int40_t LL_var1, __int40_t LL_var2)
```

Generates an 8.31 fractional saturated addition instruction.

A saturated operation computes just like a normal operation, unless an overflow or underflow occurs. If this should happen, the result will be the highest or lowest possible representable value.

__int40_t LL_mac

```
__int40_t LL_mac (__int40_t LL_var1, __int16_t var2,
       __int16_t var3)
```

Generates a saturated multiply var2*var3 and accumulate to an 8.31 fractional instruction.

A saturated operation computes just like a normal operation, unless an overflow or underflow occurs. If this should happen, the result will be the highest or lowest possible representable value.

__int40_t LL_msu

```
__int40_t LL_msu (__int40_t LL_var1, __int16_t var2,
       __int16_t var3)
```

Generates a saturated multiply var2*var3 and subtract to an 8.31 fractional instruction.

A saturated operation computes just like a normal operation, unless an overflow or underflow occurs. If this should happen, the result will be the highest or lowest possible representable value.

__int40_t LL_negate

```
__int40_t LL_negate (__int40_t LL_var1)
```

Generates an 8.31 saturated negate instruction.

A saturated operation computes just like a normal operation, unless an overflow or underflow occurs. If this should happen, the result will be the highest or lowest possible representable value.

`__int40_t LL_sub`

```
__int40_t LL_sub (__int40_t LL_var1, __int40_t LL_var2)
```

Generates an 8.31 saturated subtraction instruction.

A saturated operation computes just like a normal operation, unless an overflow or underflow occurs. If this should happen, the result will be the highest or lowest possible representable value.

`__mem * add_mod`

```
void __mem * add_mod (void __mem * base, __int16_t size,
       void __mem * ptr, __int16_t step)
```

Executes a module addition to `ptr`.

`base` points to the start of the vector, `size` is the size of the vector in bytes, `ptr` is the current pointer, and `step` is the step-value in bytes.

`__no_operation`

```
void __no_operation(void);
```

Generates a `NOP` instruction.

`__require`

```
void __require(void *);
```

Sets a constant literal as required.

One of the prominent features of the IAR XLINK Linker is its ability to strip away anything that is not needed. This is a very good feature since it reduces the resulting code size to a minimum. However, in some situations you may want to be able to explicitly include a piece of code or a variable even though it is not directly used.

The argument to `__require` could be a variable, a function name, or an exported assembler label. It must, however, be a constant literal. The label referred to will be treated as if it would be used at the location of the `__require` call.

Example

In the following example, the copyright message will be included in the generated binary file even though it is not directly used.

```
#include <intrdspic.h>
char copyright[] = "Copyright 2002 by XXXX";
void main(void)
{
  __require(copyright);
  [... the rest of the program ...]
}
```

`__reset` `void __reset(void);`

Inserts a `RESET` instruction.

Library functions

This chapter gives an introduction to the C and Embedded C++ library functions. It also lists the header files used for accessing library definitions.

For detailed information about the C library functions, see the online documentation supplied with the product.

The dsPIC IAR C/EC++ Compiler provides a complete set of library header files both for the IAR CLIB library and for the IAR DLIB library. The sections below summarize these header files.

IAR CLIB library

The dsPIC IAR C/EC++ Compiler package provides most of the important C library definitions that apply to PROM-based embedded systems. These are of three types:

- Standard C library definitions available for user programs. These are documented in this chapter.
- `CSTARTUP`, the single program module containing the start-up code. It is described in the *Runtime environment* chapter in this guide.
- Runtime support libraries; for example, low-level floating-point routines.
- Intrinsic functions, allowing low-level use of dsPIC features. See the chapter *Intrinsic functions* for more information.

LIBRARY OBJECT FILES

You must create an appropriate library object file for the chosen project settings. See the *Runtime environment* chapter for more information. The IAR XLINK Linker will include only those routines that are required—directly or indirectly—by your application.

Most of the library definitions can be used without modification, that is, directly from the library object files that are supplied with the product. There are some I/O-oriented routines (such as `putchar` and `getchar`) that you may need to customize for your target application.

HEADER FILES

The user program gains access to library definitions through header files, which it incorporates using the `#include` directive. To avoid wasting time at compilation, 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 this can cause the call to fail during execution, or generate error or warning messages at compile time or link time.

LIBRARY DEFINITIONS SUMMARY

This section lists the header files. Header files may additionally contain target-specific definitions.

Header file	Description
`assert.h`	Assertions.
`ctype.h`	Character handling.
`iccbutl.h`	Low-level routines.
`math.h`	Mathematics.
`setjmp.h`	Non-local jumps.
`stdarg.h`	Variable arguments.
`stdio.h`	Input/output.
`stdlib.h`	General utilities.
`string.h`	String handling.

Table 33: IAR C Library header files

The following table shows header files that do not contain any functions, but specify various definitions and data types:

Header file	Description
`errno.h`	Error return values.
`float.h`	Limits and sizes of floating-point types.
`limits.h`	Limits and sizes of integral types.
`stddef.h`	Common definitions including `size_t`, `NULL`, `ptrdiff_t`, and `offsetof`.

Table 34: Miscellaneous IAR C Library header files

IAR DLIB library

The dsPIC IAR C/EC++ Compiler package provides most of the important C and Embedded C++ library definitions that apply to PROM-based embedded systems. These are of the following types:

- Adherence to a free-standing implementation of the ISO 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 single program module containing the start-up code. It is described in the *Runtime environment* chapter in this guide.
- Runtime support libraries; for example, low-level floating-point routines.
- Intrinsic functions, allowing low-level use of dsPIC features. See the chapter *Intrinsic functions* for more information.

LIBRARY OBJECT FILES

You must select the appropriate library object file for your chosen project settings. See the chapter *Runtime environment* for more information. The linker will include only those routines that are required—directly or indirectly—by your application.

Most of the library definitions can be used without modification, that is, directly from the supplied library object files. There are some I/O-oriented routines (such as `__writechar` and `__readchar`) that you may need to customize for your application. For a description of the primitive I/O functions, see the *Runtime enviroment* chapter in this guide.

HEADER FILES

The user 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 this can cause the call to fail during execution, or generate error or warning messages at compile time or link time.

LIBRARY DEFINITIONS SUMMARY

This section lists the header files. Header files may additionally contain target-specific definitions; these are documented in the chapter *IAR C extensions*.

Standard C

The following table shows the traditional standard C library header files:

Header file	Usage
assert.h	Enforcing assertions when functions execute
ctype.h	Classifying characters
errno.h	Testing error codes reported by library functions

Table 35: Traditional standard C library header files

Header file	Usage
`float.h`	Testing floating-point type properties
`iso646.h`	Using Amendment 1—iso646.h standard header
`limits.h`	Testing integer type properties
`locale.h`	Adapting to different cultural conventions
`math.h`	Computing common mathematical functions
`setjmp.h`	Executing non-local goto statements
`signal.h`	Controlling various exceptional conditions
`stdarg.h`	Accessing a varying number of arguments
`stddef.h`	Defining several useful types and macros
`stdio.h`	Performing input and output
`stdlib.h`	Performing a variety of operations
`string.h`	Manipulating several kinds of strings
`time.h`	Converting between various time and date formats
`wchar.h`	Support for wide characters
`wctype.h`	Classifying wide characters

Table 35: Traditional standard C library header files (Continued)

Embedded C++

The following table shows the Embedded C++ library header files:

Header file	Usage
`complex`	Defining a class that supports complex arithmetic
`exception`	Defining several functions that control exception handling
`fstream`	Defining several I/O streams classes that manipulate external files
`iomanip`	Declaring several I/O streams manipulators that take an argument
`ios`	Defining the class that serves as the base for many I/O streams classes
`iosfwd`	Declaring several I/O streams classes before they are necessarily defined
`iostream`	Declaring the I/O streams objects that manipulate the standard streams
`istream`	Defining the class that performs extractions
`new`	Declaring several functions that allocate and free storage
`ostream`	Defining the class that performs insertions

Table 36: Embedded C++ library header files

Header file	Usage
sstream	Defining several I/O streams classes that manipulate string containers
stdexcept	Defining several classes useful for reporting exceptions
streambuf	Defining classes that buffer I/O streams operations
string	Defining a class that implements a string container
strstream	Defining several I/O streams classes that manipulate in-memory character sequences

Table 36: Embedded C++ library header files (Continued)

Using standard C libraries in EC++

The Embedded 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.

The following table shows the new header files:

Header file	Usage
cassert	Enforcing assertions when functions execute
cctype	Classifying characters
cerrno	Testing error codes reported by library functions
cfloat	Testing floating-point type properties
climits	Testing integer type properties
clocale	Adapting to different cultural conventions
cmath	Computing common mathematical functions
csetjmp	Executing non-local goto statements
csignal	Controlling various exceptional conditions
cstdarg	Accessing a varying number of arguments
cstddef	Defining several useful types and macros
cstdio	Performing input and output
cstdlib	Performing a variety of operations
cstring	Manipulating several kinds of strings
ctime	Converting between various time and date formats

Table 37: New standard C library header files

Compatibility with standard C++

In this implementation, the Embedded C++ library also includes a number of header files for compatibility with traditional C++ libraries:

Header file	Usage
`fstream.h`	Defining several I/O streams template classes that manipulate exteral files
`iomanip.h`	Declaring several I/O streams manipulators that take an argument
`iostream.h`	Declaring the I/O streams objects that manipulate the standard streams
`new.h`	Declaring several functions that allocate and free storage

Table 38: Traditional C++ library header files

Diagnostics

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* is a self-explanatory message, possibly several lines long.

Severity levels

The diagnostics are divided into different levels of severity:

Remark

A diagnostic 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 98.

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 96.

Error

A diagnostic that is produced when the compiler has found a construct which clearly violates the C or Embedded 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 85, for a description of the compiler options that are available for setting severity levels.

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

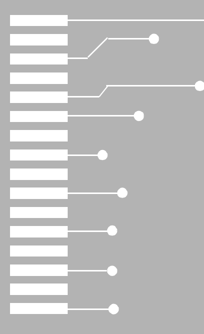
INTERNAL ERROR

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

```
Internal error: message
```

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

- 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.
- The version number of the compiler. To display it at sign-on, run the compiler, `iccdspic`, without parameters.



Part 3: Portability

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

- Implementation-defined behavior
- IAR C extensions.

Implementation-defined behavior

This chapter describes how IAR C 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: IAR C 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)

IAR C produces diagnostics 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; `message` is an explanatory message, possibly several lines.

ENVIRONMENT

Arguments to main (5.1.2.2.2.1)

In IAR C, the function called at program startup is called `main`. There is no prototype declared for `main`, and the only definition supported for `main` is:

```
int main(void)
```

To change this behavior, see *Customizing cstartup.s59*, page 54.

Interactive devices (5.1.2.3)

IAR C treats the streams `stdin` and `stdout` 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)

IAR C treats identifiers with external linkage 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. In IAR C, the source character set is the standard ASCII character set.

The execution character set is the set of legal characters that can appear in the execution environment. In IAR C, the execution character set is the standard ASCII character set.

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, i.e. using the same representation value 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 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 current and only locale supported in IAR C is the ‘C’ locale.

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 two's-complement form. The most-significant bit holds the sign; 1 for negative, 0 for positive and zero.

See *Data types*, page 67, for information about the ranges for the different integer types: `char`, `short`, `int`, `long`, and `long long`.

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, i.e. a large enough value will be converted into a negative value.

Signed bitwise operations (6.3)

Bitwise operations on signed integers work the same as bitwise operations on unsigned integers, i.e. 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 68, 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 70, for information about `size_t` in IAR C.

Conversion from/to pointers (6.3.4)

See *Casting*, page 70, for information about casting of data pointers and function pointers.

ptrdiff_t (6.3.6, 7.1.1)

See *ptrdiff_t*, page 70, for information about the `ptrdiff_t` in IAR C.

REGISTERS

Honoring the register keyword (6.5.1)

IAR C does not honor user requests for register variables. Instead it makes it own choices when optimizing.

STRUCTURES, UNIONS, ENUMERATIONS, AND BITFIELDS

Improper access to a union (6.3.2.3)

If a union get 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 *Data types*, page 67, for information about the alignment requirement for data objects in IAR C.

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 bitfield integer type chosen.

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)

IAR C does not limit the number of declarators. The number is limited only by the available memory.

STATEMENTS

Maximum number of case statements (6.6.4.2)

IAR C does not limit the number of case statements (case values) in a switch statement. 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 has 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 were enclosed in angle brackets.

Character sequences (6.8.2)

Preprocessor directives use the source character set, with the exception of escape sequences. Thus to specify a path for an include file, use only one backslash:

```
#include "mydirectory\myfile"
```

Within source code, two backslashes are necessary:

```
file = fopen("mydirectory\\myfile","rt");
```

Recognized #pragma directives (6.8.6)

The following `#pragma` directives are recognized in IAR C:

```
alignment
ARGSUSED
baseaddr
bitfields
can_instantiate
codeseg
constseg
dataseg
define_type_info
diag_default
diag_error
diag_remark
diag_suppress
diag_warning
do_not_instantiate
function
hdrstop
inline
instantiate
language
location
memory
message
none
```

```
no_pch
NOTREACHED
object_attribute
once
optimize
pack
__printf_args
__scanf_args
type_attribute
VARARGS
vector
warnings
```

For a description of the `#pragma` directives, see the chapter *#pragma directives*.

Default __DATE__ and __TIME__ (6.8.8)

The definitions for `__TIME__` and `__DATE__` are always available.

IAR 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)

IAR C does not support the signal part of the library.

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 out to the `stdout` stream immediately before a newline character are preserved. There is no way to read in the line through the stream `stdin` that was written out through the stream `stdout` in IAR C.

Null characters appended to data written to binary streams (7.9.2)

There are no binary streams implemented in IAR C.

Files (7.9.3)

There are no streams other than `stdin` and `stdout` in IAR C. This means that a file system is not implemented.

remove() (7.9.4.1)

There are no streams other than `stdin` and `stdout` in IAR C. This means that a file system is not implemented.

rename() (7.9.4.2)

There are no streams other than `stdin` and `stdout` in IAR C. 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 that 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 streams other than `stdin` and `stdout` in IAR C. This means that a file system is not implemented.

Message generated by perror() (7.9.10.4)

`perror()` is not supported in IAR C.

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, since this is an unsupported feature in IAR C.

Behavior of exit() (7.10.4.3)

The `exit()` function does not return in IAR C.

Environment (7.10.4.4)

An environment is not supported in IAR C.

system() (7.10.4.5)

The `system()` function is not supported in IAR C.

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 39: Message returned by strerror()—IAR CLIB library

The time zone (7.12.1)

The time zone function is not supported in IAR C.

clock() (7.12.2.1)

The `clock()` function is not supported in IAR C.

IAR DLIB LIBRARY FUNCTIONS

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)

IAR C does not support the signal part of the library.

Note: Interface functions exist but will not perform anything. Instead, they will result in an error.

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 out to the `stdout` stream immediately before a newline character are preserved. There is no way to read in the line through the stream `stdin` that was written out through the stream `stdout` in IAR C.

Null characters appended to data written to binary streams (7.9.2)

There are no binary streams implemented in IAR C.

Note: Interface functions exist but will not perform anything. Instead, they will result in an error.

Files (7.9.3)

There are no streams other than `stdin` and `stdout` in IAR C. This means that a file system is not implemented.

Note: Interface functions exist but will not perform anything. Instead, they will result in an error.

remove() (7.9.4.1)

There are no streams other than `stdin` and `stdout` in IAR C. This means that a file system is not implemented.

Note: Interface functions exist but will not perform anything. Instead, they will result in an error.

rename() (7.9.4.2)

There are no streams other than `stdin` and `stdout` in IAR C. This means that a file system is not implemented.

Note: Interface functions exist but will not perform anything. Instead, they will result in an error.

%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 that 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)

There are no streams other than `stdin` and `stdout` in IAR C. This means that a file system is not implemented.

Note: Interface functions exist but will not perform anything. Instead, they will result in an error.

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, since this is an unsupported feature in IAR C.

Behavior of exit() (7.10.4.3)

The `exit()` function does not return in IAR C.

Environment (7.10.4.4)

An environment is not supported in IAR C.

Note: Interface functions exist but will not perform anything. Instead, they will result in an error.

system() (7.10.4.5)

The `system()` function is not supported in IAR C.

Note: Interface functions exist but will not perform anything. Instead, they will result in an error.

Message returned by strerror() (7.11.6.2)

The messages returned by `strerror()` depending on the argument is:

Argument	Message
`EZERO`	`no error`
`EDOM`	`domain error`
`ERANGE`	`range error`
`EFPOS`	`file positioning error`
`EILSEQ`	`multi-byte encoding error`
`<0 || >99`	`unknown error`
all others	`error` *nnn*

Table 40: Message returned by strerror()—IAR DLIB library

The time zone (7.12.1)

Time is not supported in IAR C.

Note: Interface functions exist but will not perform anything. Instead, they will result in an error.

clock() (7.12.2.1)

Time is not supported in IAR C.

Note: Interface functions exist but will not perform anything. Instead, they will result in an error.

IAR C extensions

This chapter describes IAR extensions to the ISO standard for the C programming language. All extensions can also be used when compiling in Embedded C++ mode.

See the compiler options -e on page 90 and --strict_ansi on page 99 for information about enabling and disable language extensions from the command line.

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

Why should language extensions be used?

By using language extensions, you gain full control over the resources and features of the target microcontroller, and can thereby fine-tune your application.

If you want to use the source code with different compilers, note that language extensions may cause minor modifications before the code can be compiled. A compiler typically supports microcontroller-specific language extensions as well as vendor-specific ones.

Descriptions of language extensions

The language extensions can be categorized into different groups according to their functionality.

Memory, type, and object attributes

Entities such as variables and functions may be declared with memory, type, and object attributes. The syntax follows the syntax for qualifiers—such as `const`—but the semantics is different.

- A memory attribute controls the placement of the entity. There can be only one memory attribute.
- A type attribute controls other aspects of the object. There can be many different type attributes and they must be included when the object is declared.
- An object attribute only has to be specified at the definition but not at the declaration of an object.

See the *Extended keywords* chapter for a complete list of attributes.

Absolute placement

The operator `@` or the directive `#pragma location` can be used for specifying either the location of an absolute addressed variable or the segment placement of a variable or function. For example:

```
int x @ 0x1000;

void test(void) @ "MYOWNSEGMENT"
{
  ...
}
```

Inline assembler

Inline assembler can be used for inserting assembler instructions into the generated function. This is seldom needed since almost all can be expressed in C with the help of intrinsic functions.

The syntax for inline assembler is:

```
asm("NOP");
```

In strict ANSI mode the use of inline assembler is disabled.

C++ style comments

C++ style comments are accepted. A C++ style comment starts with the character sequence `//` and continues to the end of the line. For example:

```
// The length of the bar, in centimeters.
int length;
```

__ALIGNOF__

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

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

Assume that a processor can read 4 bytes of memory using one instruction but only when the memory read is placed on an address dividable 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. If is not true, only the first element of an array would be placed in accordance with the alignment requirements.

In the example below, the alignment of the structure is 4, under the assumption that `long` has alignment 4. Its size is 8, even though only 5 bytes are effectively used.

```
struct str {
  long a;
  char b;
};
```

In standard C, the size of an object can be accessed using the `sizeof` operator.

The `__ALIGNOF__` operator can be used to access the alignment of an object. It can take 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 IAR Systems compilers allow a similar feature for both structs and unions.

An anonymous structure type (i.e. one without a name) defines an unnamed object (and not a type) whose members are promoted to the surrounding scope. External anonymous structure types are allowed.

For example, the structure `str` below contains an anonymous union. The members of the union are accessed using the names `b` and `c`, for example `obj.b`.

Without anonymous structure types the union would have to be named—for example `u`—and the member elements accessed using the more clumsy syntax `obj.u.b`.

```
struct str {
  int a;
  union {
    int b;
    int c;
  }
};

struct str obj;
```

Bitfields and non-standard types

In standard C, a bitfield must be of type `int` or `unsigned int`. Using IAR extensions any integer type and enums may be used.

For example, in the following structure a `char` is used for holding three bits. The advantage is that the struct will be smaller.

```
struct str {
  char bitOne   : 1;
  char bitTwo   : 1;
  char bitThree : 1;
};
```

This matches G.5.8 in the appendix of the ISO standard, *ISO Portability Issues*.

Incomplete arrays at end of structs

The last element of a `struct` may be an incomplete array. This is useful since one chunk of memory can be allocated for the `struct` itself and for the array, regardless of the size of the array.

Note: The array may not be the only member of the `struct`. If that were the case, then the size of the array would be zero, which is not allowed in C.

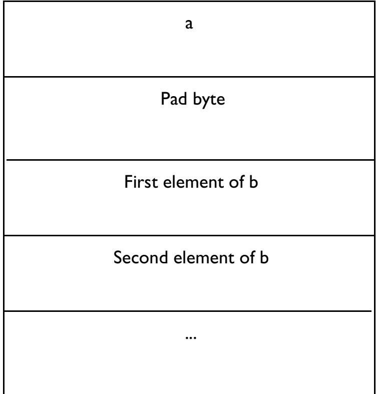
Example

```
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 above the alignment of `struct str` will be 2. (Assuming a processor where the alignment of `unsigned long` is 2.)

The memory layout of `struct str` is:



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 subscribed (if it is), and by the end of the compilation unit if it is not.

Empty translation units

A translation unit (source file) is allowed to be empty, i.e. it does not contain any declarations.

In strict ANSI mode a warning is issued if the compilation unit is empty.

Example

The following source file is only used in a debug build. (In a debug build the `NDEBUG` preprocessor flag is undefined.) Since the entire content of the file is surrounded by preprocessor tests, the translation unit will be empty when the application is compiled in release mode. Without this extension, this would be considered an error.

```
#ifndef NDEBUG

void PrintStatusToTerminal(void)
{
  /* Do something */
}

#endif
```

Comments at the end of preprocessor directives

This extension, which makes it legal to place text after preprocessor directives, is enabled unless strict ANSI mode is used. This language extension exists to support compilation of old legacy code; it is not recommended to write new code in this fashion.

Example

```
#ifdef FOO

  ... something ...

#endif FOO
```

Forward declaration of enums

It is possible to first declare the name of an `enum` and later resolve it by specifying the brace-enclosed list.

Extra comma at end of enum list

It is allowed to place an extra comma at the end of an `enum` list. In strict ANSI mode a warning is issued.

Note: C allows extra commas in similar situations, for example after the last element of the initializers to an array. The reason is that it is easy to get the commas wrong if parts of the list are moved using a normal cut-and-paste operation.

Example

```
enum {
   kOne,
   kTwo,   /* This is now allowed. */
};
```

Unsigned int enum constants

The `enum` constants may be given values that fit into the `unsigned int` range but not in the `int` range. In strict ANSI mode a warning is issued.

```
#include <limits.h>

enum {
  kFirst  = 0,
  KSecond = UINT_MAX
};
```

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 standard C, some operators allow such things, while others do not allow them.

A label preceding a "}"

In standard C, a label must be followed by at least one statement. Hence it is illegal to place the label at the end of a block. In the IAR Systems compiler a warning is issued.

To create a correct standard-compliant C program (so that you will not have to see the warning) you can place an empty statement after the label. An empty statement is a single `;` (semi-colon).

Example

```
void test(void)
{
  if (...) goto end;

   /* Do something */

  end:  /* Illegal at the end of block. */
}
```

Note: This also applies to the labels of switch statements. The following piece of code will generate the warning.

```
switch (x)
{
```

```
case 1:
  ...;
  break;

default:
}
```

A good way to convert this into a correct program is to place a `break;` statement after the `default:` label.

Empty declarations

An empty declaration (a semicolon by itself) is allowed but a remark is issued.

This is useful when preprocessor macros are used that could expand to nothing. Consider the following example. In a debug build the macros `DEBUG_ENTER` and `DEBUG_LEAVE` could be defined to something useful. In a release build, however, they could expand into nothing, leaving the `;` character in the code.

```
void test(void)
{
  DEBUG_ENTER();

  do_something();

  DEBUG_LEAVE();
}
```

Single value initialization

Standard C requires that all initializer expressions of static arrays, `structs`, and `unions` should be enclosed in braces.

Single-value initializers are allowed to appear without braces, but a warning is issued.

Example

In the IAR C Compiler, the following expression is allowed:

```
struct str
{
  int a;
} x = 10;
```

Casting pointers to integers

In an initializer, a pointer constant value may be cast to an integral type if the integral type is large enough to contain it.

In the example below we assume that pointers to __xmem and __sfr are 16 and 32 bits, respectively.

In the example below the first initialization is correct since it is possible to cast the 16-bit address to a 16-bit unsigned short variable. However, it is illegal to use the 32-bit address of b as initializer for a 16-bit value.

```
__xmem int a;
const int b=2;

unsigned short ap = (unsigned short)&a;  /* Correct */
unsigned short bp = (unsigned short)&b;  /* Error */
```

Casting integers to pointers and back in constant expressions

In constant integer expressions, it is allowed to cast an integer to a pointer and back.

Taking the address of a register variable

In standard C, it is illegal to take the address of a variable specified as a register variable.

The IAR 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"

long float is accepted as synonym for double.

Repeated typedefs

Redeclarations of typedef that occur in the same scope are allowed, but a warning is issued.

Mixing pointer types

Assignment and pointer difference is allowed between pointers to types that are interchangeable but not identical, for example, unsigned char * and char *. This includes pointers to integral types of the same size. A warning is issued.

Assignment of a string constant to a pointer to any kind of character is allowed, and no warning will be produced.

Non-top level const

Assignment of pointers is allowed in cases where the destination type has added type qualifiers that are not at the top level (for example `int **` to `int const **`). It is also allowed to compare and take the difference of such pointers.

Declarations in other scopes

External and static declarations in other scopes are visible. In the following example the variable `y` can be used at the end of the function, even though it should only be visible in the body of the `if` statement. A warning is issued.

```
int test(int x)
{
  if (x)
  {
    extern int y;
    y = 1;
  }

  return y;
}
```

External or static entities declared in other scopes are visible. A warning is issued.

Non-lvalue arrays

A non-`lvalue` array expression is converted to a pointer to the first element of the array when it is subscribed or similarly used.

A

-A (XLINK option) 58
address 25
addressing. *See* memory types
anonymous structures 23
applications
  building 3
  initializing 53
  terminating 53
architecture, dsPIC xiii
ARGFRAME (compiler function directive) 41
arrays 147
_ _asm (intrinsic function) 124
asm (inline assembler) 6
assembler directives
  CFI 56
  ENDMOD 55
  EQU 97
  EXTERN 55
  MODULE 55
  PUBLIC 55, 97
  REQUIRE 55
  RSEG 55
  RTMODEL 35
assembler labels
  ?C_EXIT 62
  ?C_GETCHAR 62
  ?C_PUTCHAR 62
  ?C_VIRTUAL_IO 62
assembler language interface 27
  calling assembler routines from C 36
  creating skeleton code 37
assembler list file 41
assembler modules 63
assembler, inline 6, 63
assert.h (library header file) 134–135
assumptions (programming experience) xiii
atomic operations, performing 106
auto variables 14
  saving stack space 64

B

Barr, Michael xv
bitfields
  in expressions 68
  in implementation-defined behavior 147
bitfields (#pragma directive) 68, 110
byte order, of dsPIC microcontroller 67

C

call chains 64
call frame information 56
call stack, displaying 56
callee-save registers, stored on stack 14
calling convention
  C 30
  Embedded C++ 40
calloc (standard library function) 15
cassert (library header file) 137
casting, of pointers and integers 70
cctype (library header file) 137
cerrno (library header file) 137
CFI (assembler directive) 56
cfloat (library header file) 137
char (data type), signed and unsigned 68, 87
characters, in implementation-defined behavior 144
--char_is_signed (compiler option) 87
_ _clear_watchdog_timer (intrinsic function) 124
CLIB library 133–134
  customizing 57
climits (library header file) 137
clocale (library header file) 137
_ _close (library function) 61
cmath (library header file) 137
code
  excluding when linking 55

placement of 73
portability of 44
startup 51
code execution, in dsPIC microcontroller 4
code models 4
code motion, disabling 95
CODE (segment) 51, 74
common sub-expression elimination, disabling 95
compiler environment variables 84
compiler error return codes 85
compiler listing, generating 93
compiler object file
including debug information 89, 98
specifying filename 96
compiler options
setting 83
specifying parameters 83
summary 85
typographic convention xvi
-D. 88
-e 90
-f 91
-I 92
-l 38, 93
-o 96
-r 89, 98
-s 98
-v 99
in predefined symbols 119
mapping of dsPIC cores 9
-z 100–101
--char_is_signed 87
--cpu 87
mapping of dsPIC cores 9
--data_model 88
--debug 89, 98
--diag_error 89
--diag_remark 89
--diag_suppress 90
--diag_warning 90
--ec++ 91
--IARStyleMessages 93
--library_module 93
--migration_preprocessor_extensions 94
--module_name 94
--no_code_motion 95
--no_cse 95
--no_inline 95
--no_unroll. 96
--no_warnings 96
--only_stdout 97
--preprocess 97
--public_equ 97
--remarks 98
--require_prototypes 98
--silent 99
--strict_ansi 99
--warnings_affect_exit_code 85, 100
--warnings_are_errors 100
--64bit_doubles 101
compiler version number 120
compiling, from the command line 3
complex numbers, supported in Embedded C++ 7
complex (library header file) 136
computer style, typographic convention xvi
consistency, module 35, 64
CONST (segment) 74
_ _constptr (extended keyword) 105
using in #pragma directives. 115
constseg (#pragma directive) 110
conventions, typographic xvi
copyright notice ii
cores
mapping of processor options 9
supported 9, 87, 99
_ _CPU_ _ (predefined symbol) 118–119
--cpu (compiler option) 87
mapping of dsPIC cores 9

CPU variant, specifying on command line 87, 99
csetjmp (library header file) 137
csignal (library header file) 137
CSTACK (segment) 49, 75
    See also stack
cstartup, customizing 54, 57
cstdarg (library header file) 137
cstddef (library header file) 137
cstdio (library header file) 137
cstdlib (library header file) 137
cstring (library header file) 137
ctime (library header file) 137
ctype.h (library header file) 134–135
C++
    features excluded from EC++ 7
    See also Embedded C++
C-SPY, low-level interface 62
?C_EXIT (assembler label) 62
?C_GETCHAR (assembler label) 62
C_INCLUDE (environment variable) 85, 92
?C_PUTCHAR (assembler label) 62
?C_VIRTUAL_IO (assembler label) 62

D

--data_model (compiler option) 88
data
    alignment of 67
    excluding when linking 55
    initialized 48
    located 51
    non-initialized 47
    non-zero initialized 47
    placement of 73
        specifying 9
    storage 4, 13
    zero-initialized 47
data memory, specifying 9
data models
    characteristics 9
data representation 67
data types
    floating point 68
    integers 67
    using efficiently 64
dataseg (#pragma directive) 111
_ _data_model (runtime model attribute) 36
_ _DATE_ _ (predefined symbol) 118
--debug (compiler option) 89, 98
debug information, including in object file 89, 98
declaration, of functions 30
declarators, in implementation-defined behavior 148
delete (keyword) 15
derivatives
    mapping of processor options 9
    supported 87, 99
diagnostic messages 139
    classifying as errors 89
    classifying as remarks 89
    classifying as warnings 90
    disabling warnings 96
    enabling remarks 98
    suppressing 90
diag_default (#pragma directive) 111
--diag_error (compiler option) 89
diag_error (#pragma directive) 111
--diag_remark (compiler option) 89
diag_remark (#pragma directive) 112
--diag_suppress (compiler option) 90
diag_suppress (#pragma directive) 112
--diag_warning (compiler option) 90
diag_warning #pragma directive) 112
DIFUNCT (segment) 51, 75
directives, #pragma 109
    overview 5
_ _disable_interrupt (intrinsic function) 124
disclaimer ii
DLIB library 134–138
    I/O functions 60

document conventions xv
documentation, library 133
double (data type). 68
_ _double_size (runtime model attribute) 36
dsPIC architecture xiii
  code execution. 4
  memory access 4
dsPIC cores
  mapping of processor options 9
  supported. 9, 87, 99
dsPIC derivatives
  terminology in this guide xv
dsPIC instruction set xiii
dynamic initialization. 57, 59–60
  in Embedded C++ 51
dynamic memory 15

E

--ec++ (compiler option) 91
_ _EI (intrinsic function) 124
Embedded C++
  calling convention 40
  differences from C++ 7
  dynamic initialization in 51
  enabling 91
  language extensions 6
  overview 6
Embedded C++ objects, placing in memory type 19
_ _enable_interrupt (intrinsic function) 124
endianness, of dsPIC microcontroller 67
ENDMOD (assembler directive) 55
enum (keyword) 68
enumerations, in implementation-defined behavior. 147
environment
  in implementation-defined behavior 144
  runtime 53
environment variables 84
  C_INCLUDE. 85, 92
  QCCDSPIC 85
EQU (assembler directive) 97
errno.h (library header file) 134–135
error messages 139
  classifying 89
error return codes 85
exception handling, missing from Embedded C++ 7
exception vectors 51
exception (library header file) 136
experience, programming. xiii
extended keywords. 103
  enabling. 90
  enum 68
  overriding default behaviors 10
  overview 5
  syntax 104
  _ _constptr. 105
    using in #pragma directives 115
  _ _func. 105
  _ _interrupt 25, 105–106, 108
    See also INTVEC (segment)
    using in #pragma directives 115–116
  _ _intrinsic. 106
  _ _mem 106
    using in #pragma directives 115
  _ _monitor 106
    using in #pragma directives 115
  _ _no_init 20, 107
    using in #pragma directives 113
  _ _ptr 108
  _ _root 108
    using in #pragma directives 113
  _ _sfr 108
    using in #pragma directives 115
  _ _xmem 108
    using in #pragma directives 115
  _ _ymem 108
    using in #pragma directives 115
EXTERN (assembler directive) 55

F

-f (compiler option) 91
fatal error messages 139
_ _FILE_ _ (predefined symbol) 118
file paths, specifying for #include files 92
filename, specifying for object file 96
float (floating-point type). 68
floating-point format 68
  implementation-defined behavior 146
  special cases 69
  32-bit 69
  4 bytes 69
  64-bit 69
  8 bytes 69
float.h (library header file) 134, 136
formats
  floating-point values 68
  standard IEEE (floating point) 68
_formatted_write (library function) 59
fragmentation, of heap memory 16
free (standard library function) 15
fstream (library header file) 136
fstream.h (library header file) 138
_ _func (extended keyword). 105
FUNCALL (compiler function directive) 41
function directives 41
function inlining, disabling 95
function parameters, type checking 64
function types, special 25
FUNCTION (compiler function directive) 41
functions
  calling 34
    using assembler instructions 34
  declaring 30
  executing
    using memory. 13
  inlining 63
  intrinsic 5, 63
  I/O 57
  overview 25
  placing in segments 26
  recursive 64
    storing data on stack 14–15
  return values from 32
  static 63

G

getchar (library function) 57
guidelines, reading xiii

H

header files 135
  assert.h. 134
  CLIB 133
  ctype.h 134
  DLIB 135
  errno.h 134
  float.h 134
  iccbutl.h. 134
  limits.h. 134
  math.h 134
  setjmp.h 134
  stdarg.h 134
  stddef.h 134
  stdio.h 134
  stdlib.h 134
  string.h. 134
heap 15
  size. 50
  specified in linker command file. 44
HEAP (segment) 50, 75
hidden parameters 31
hints
  optimization. 63
  programming. 63

I

-I (compiler option) . . . 92
--IARStyleMessages (compiler option) . . . 93
_ _IAR_SYSTEMS_ICC_ _ (predefined symbol) . . . 119
iccbutl.h (library header file) . . . 134
_ _ICCDSPIC_ _ (predefined symbol) . . . 119
ICODE (segment) . . . 51, 76
identifiers, in implementation-defined behavior . . . 144
IEEE format, floating-point values . . . 68
implementation-defined behavior . . . 143
indspic.h (header file) . . . 124
inheritance, in Embedded C++ . . . 6
initialization
  dynamic . . . 57, 59–60
  modifying cstartup . . . 57
inline assembler . . . 6, 63
  See also assembler language interface
inline (#pragma directive) . . . 112
input functions, in runtime library . . . 57
instruction set, dsPIC . . . xiii
int (data type) . . . 67
integers . . . 67
  casting . . . 70
  in implementation-defined behavior . . . 145
  ptrdiff_t . . . 70
  size_t . . . 70
internal error . . . 140
_ _interrupt (extended keyword) . . . 25, 105–106, 108
  using in #pragma directives . . . 115–116
interrupt functions
  in assembler language . . . 34
  in C language . . . 25
  placement in memory . . . 51
interrupt vector table . . . 25
  description . . . 34
interrupt vectors
  in assembler . . . 34
  specifying with #pragma directive . . . 116
interrupt (function type) . . . 25
interrupts
  disabling during function execution . . . 33
  INTVEC segment . . . 76
_ _intrinsic (extended keyword) . . . 106
intrinsic functions . . . 63
  overview . . . 5
  summary . . . 121
  _ _asm . . . 124
  _ _clear_watchdog_timer . . . 124
  _ _disable_interrupt . . . 124
  _ _EI . . . 124
  _ _enable_interrupt . . . 124
  _ _int16_t abs_s . . . 125
  _ _int16_t add . . . 125
  _ _int16_t div_s . . . 125
  _ _int16_t extract_h . . . 125
  _ _int16_t extract_l . . . 125
  _ _int16_t mac_r . . . 125
  _ _int16_t msu_r . . . 126
  _ _int16_t mult . . . 126
  _ _int16_t mult_r . . . 126
  _ _int16_t negate . . . 126
  _ _int16_t norm_l . . . 126
  _ _int16_t norm_s . . . 126
  _ _int16_t round . . . 126
  _ _int16_t round_ub . . . 127
  _ _int16_t shl . . . 127
  _ _int16_t shr . . . 127
  _ _int16_t shr_r . . . 127
  _ _int16_t sub . . . 127
  _ _int16_t __xmem * add_br . . . 127
  _ _int32_t L_abs . . . 128
  _ _int32_t L_add . . . 128
  _ _int32_t L_deposit_h . . . 128
  _ _int32_t L_deposit_l . . . 128
  _ _int32_t L_mac . . . 128
  _ _int32_t L_msu . . . 128
  _ _int32_t L_mult . . . 129

_ _int32_t L_negate. 129
_ _int32_t L_shl 129
_ _int32_t L_shr 129
_ _int32_t L_shr_r. 129
_ _int32_t L_sub 130
_ _int40_t LL_add. 130
_ _int40_t LL_mac 130
_ _int40_t LL_msu 130
_ _int40_t LL_negate 130
_ _int40_t LL_sub. 131
_ _mem * add_mod. 131
_ _no_operation. 131
_ _require 131
_ _reset 132
INTVEC (segment) 51, 76
_ _int16_t abs_s(intrinsic function) 125
_ _int16_t add (intrinsic function) 125
_ _int16_t div_s (intrinsic function). 125
_ _int16_t extract_h (intrinsic function) 125
_ _int16_t extract_l (intrinsic function) 125
_ _int16_t mac_r (intrinsic function) 125
_ _int16_t msu_r (intrinsic function) 126
_ _int16_t mult (intrinsic function) 126
_ _int16_t mult_r (intrinsic function). 126
_ _int16_t negate (intrinsic function). 126
_ _int16_t norm_l (intrinsic function) 126
_ _int16_t norm_s (intrinsic function) 126
_ _int16_t round (intrinsic function) 126
_ _int16_t round_ub (intrinsic function) 127
_ _int16_t shl (intrinsic function). 127
_ _int16_t shr (intrinsic function) 127
_ _int16_t shr_r (intrinsic function) 127
_ _int16_t sub (intrinsic function) 127
_ _int16_t __xmem * add_br (intrinsic function) 127
_ _int32_t L_abs (intrinsic function) 128
_ _int32_t L_add (intrinsic function). 128
_ _int32_t L_deposit_h (intrinsic function) 128
_ _int32_t L_deposit_l (intrinsic function) 128
_ _int32_t L_mac (intrinsic function) 128
_ _int32_t L_msu (intrinsic function) 128
_ _int32_t L_mult (intrinsic function) 129
_ _int32_t L_negate (intrinsic function) 129
_ _int32_t L_shl (intrinsic function) 129
_ _int32_t L_shr (intrinsic function) 129
_ _int32_t L_shr_r (intrinsic function). 129
_ _int32_t L_sub (intrinsic function) 130
_ _int40_t LL_add (intrinsic function). 130
_ _int40_t LL_mac (intrinsic function) 130
_ _int40_t LL_msu (intrinsic function) 130
_ _int40_t LL_negate (intrinsic function) 130
_ _int40_t LL_sub (intrinsic function). 131
iomanip (library header file) 136
iomanip.h (library header file) 138
ios (library header file). 136
iosfwd (library header file). 136
iostream (library header file) 136
iostream.h (library header file). 138
ISO/ANSI C
C++ features excluded from EC++ 7
specifying strict usage 99
iso646.h (library header file) 136
istream (library header file) 136
I/O functions 57

K

Kernighan, Brian W.. xv
keywords, extended 5

L

-l (compiler option) 38, 93
Labrosse, Jean J.. xv
language extensions
Embedded C++ 6
enabling. 90
overview 4
using anonymous structures and unions 23
language (#pragma directive). 112

large (memory model) 10
libraries. 3
compatibility 64
runtime 10
library documentation 133
library features, missing from Embedded C++ 7
library functions 133
CLIB 133–134
customizing 57
DLIB 134–138
I/O functions. 60
getchar. 57
printf 59
putchar. 57
remove. 61
rename. 61
scanf 60
sprintf 59
sscanf. 60
summary 134
_ _close 61
_ _lseek 61
_ _open 61
_ _read 61
_ _readchar 61
_ _write 61
_ _writechar. 61
library modules, creating 93
library object files 133, 135
--library_module (compiler option) 93
limits.h (library header file) 134, 136
_ _LINE_ _ (predefined symbol) 119
linker command files 44
customizing 45
ready-made 44
linking, from the command line 4
listing, generating. 93
literature, recommended. xv
Little Endian. 67
locale.h (library header file). 136
located data 51
location (#pragma directive) 113
example 21
LOCFRAME (compiler function directive). 41
long long (data type) 68
long (data type) 68
_ _long_double_size (runtime model attribute) 36
loop unrolling, disabling 96
low-level processor operations. 5, 121
_ _low_level_init, customizing 54
_ _lseek (library function) 61

M

malloc (standard library function) 15
Mann, Bernhard xv
math.h (library header file) 134, 136
_medium_write (library function) 59
_ _mem (extended keyword) 106
using in #pragma directives. 115
_ _mem * add_mod (intrinsic function). 131
memory
access methods 4, 16
allocating in Embedded C++. 15
dynamic. 15
heap 15
non-initialized 20
RAM, saving 64
releasing in Embedded C++ 15
stack. 13
saving 64
static 15
used by executing functions 13
used by global or static variables 15
memory management, type-safe 6
memory models
default 10
memory types. 16
Embedded C++ 19

placing variables in 19
summary 17
_ _MEMORY_MODEL_ _ (predefined symbol) 118
MEM_A (segment) 76
MEM_I (segment) 76
MEM_ID (segment) 77
MEM_N (segment) 77
MEM_Z (segment) 77
migration
from earlier IAR compilers 94
--migration_preprocessor_extensions (compiler option) . . 94
module consistency 35, 64
module name, specifying 94
MODULE (assembler directive) 55
modules, assembler 55
module-local variables 63
--module_name (compiler option) 94
_ _monitor (extended keyword) 106
using in #pragma directives 115
monitor functions 26, 33, 106
multiple inheritance, missing from Embedded C++ 7

N

namespaces, missing from Embedded C++ 7
new cast syntax, missing from Embedded C++ 7
new (keyword) 15
new (library header file) 136
new.h (library header file) 138
non-initialized memory 20
non-scalar parameters 64
--no_code_motion (compiler option) 95
--no_cse (compiler option) 95
_ _no_init (extended keyword) 20, 107
using in #pragma directives 113
--no_inline (compiler option) 95
_ _no_operation (intrinsic function) 131
--no_unroll (compiler option) 96
--no_warnings (compiler option) 96
NULL 134

O

-o (compiler option) 96
object files
library 133, 135
specifying filename 96
object module name, specifying 94
object_attribute (#pragma directive) 20, 113
offsetof 134
--only_stdout (compiler option) 97
_ _open (library function) 61
optimization 63
code motion, disabling 95
common sub-expression elimination, disabling 95
function inlining, disabling 95
hints 63
loop unrolling, disabling 96
size, specifying 100–101
speed, specifying 98
techniques 4
optimize (#pragma directive) 114
options summary, compiler 85
Oram, Andy xv
ostream (library header file) 136
output functions, in runtime library 57
output, preprocessor 97

P

parameters
function 31
hidden 31
non-scalar 64
register 31
specifying 83
stack 31–32
typographic convention xvi
permanent registers 32
placeholder segments 44
placement of code and data 73

pointers . . . 17, 147
casting . . . 70
size of . . . 70
using instead of large non-scalar parameters . . . 64
polymorphism, in Embedded C++ . . . 6
porting, of code . . . 44
containing #pragma directives . . . 110
#pragma directives . . . 109
predefined symbols
overview . . . 5
_ _cplusplus . . . 118
_ _CPU_ _ . . . 118–119
_ _DATE_ _ . . . 118
_ _embedded_cplusplus . . . 118
_ _FILE_ _ . . . 118
_ _IAR_SYSTEMS_ICC_ _ . . . 119
_ _ICCDSPIC_ _ . . . 119
_ _LINE_ _ . . . 119
_ _MEMORY_MODEL_ _ . . . 118
_ _STDC_ _ . . . 119
_ _STDC_VERSION_ _ . . . 119
_ _TID_ _ . . . 119
_ _TIME_ _ . . . 120
_ _VER_ _ . . . 120
--preprocess (compiler option) . . . 97
preprocessing directives, in implementation-defined behavior . . . 148
preprocessor
extending . . . 94
output, directing to file . . . 97
preprocessor symbols, defining . . . 88
prerequisites (programming experience) . . . xiii
printf (library function) . . . 59
processor operations, low-level . . . 5, 121
processor options
mapping of derivatives . . . 9
processor variant, specifying on command line . . . 87, 99
programming experience, required . . . xiii
programming hints . . . 63
project options, setting . . . 9
_ _ptr (extended keyword) . . . 108
ptrdiff_t . . . 134
ptrdiff_t (integer type) . . . 70
PUBLIC (assembler directive) . . . 55, 97
--public_equ (compiler option) . . . 97
putchar (library function) . . . 57

Q

QCCDSPIC (environment variable) . . . 85
qualifiers, in implementation-defined behavior . . . 148

R

-r (compiler option) . . . 89, 98
RAM memory
saving . . . 64
RCODE (segment) . . . 77
_ _read (library function) . . . 61
read formatter, selecting . . . 60
_ _readchar (library function) . . . 61
reading guidelines . . . xiii
reading, recommended . . . xv
realloc (standard library function) . . . 15
recursive functions . . . 64
storing data on stack . . . 14–15
reference information, typographic convention . . . xvi
register parameters . . . 31
registered trademarks . . . ii
registers . . . 147
callee-save, stored on stack . . . 14
erasing using assembler-level routine . . . 30
permanent . . . 32
scratch . . . 32
remark (diagnostic message) . . . 139
classifying . . . 89
enabling . . . 98
--remarks (compiler option) . . . 98
remove (library function) . . . 61
rename (library function) . . . 61

_ _require (intrinsic function) 131
REQUIRE (assembler directive) 55
--require_prototypes (compiler option) 98
_ _reset (intrinsic function) 132
return values, from functions 32
Ritchie, Dennis M. xv
_ _root (extended keyword) 108
  using in #pragma directives. 113
routines, time-critical 5, 121
RSEG (assembler directive) 55
RTMODEL (assembler directive) 35
rtmodel (#pragma directive). 114
_ _rt_version (runtime model attribute) 36
runtime environment 53
runtime libraries 6, 10
  summary 11
runtime model attributes 35
  predefined 36
  _ _data_model 36
  _ _double_size. 36
  _ _long_double_size 36
  _ _rt_version 36
runtime type information, missing from Embedded C++ . . 7

S

-s (compiler option) 98
saddr memory. 10
scanf (library function). 60
scratch registers 32
search procedure, #include files. 92
segment types, in XLINK 43
segments. 43, 73
  CODE 51, 74
  code 51
  CONST 74
  CSTACK 49, 75
  DIFUNCT 51, 75
  dynamic data 50
  HEAP 50, 75
  ICODE. 51, 76
  INTVEC 51, 76
  mem 48
  MEM_A 76
  MEM_I 76
  MEM_ID 77
  MEM_N 77
  MEM_Z. 77
  non-static 49
  placeholder 44
  placement in memory, example. 45
  RCODE 77
  sfr. 48
  SFR_A. 78
  SFR_I 78
  SFR_ID 78
  SFR_N. 78
  SFR_Z 79
  static 46
  summary 73
  XMEM_A 79
  XMEM_I. 79
  XMEM_ID 79
  XMEM_N 80
  XMEM_Z 80
  YMEM_A 80
  YMEM_I. 80
  YMEM_ID 81
  YMEM_N 81
  YMEM_Z 81
semaphores, operations on 106
setjmp.h (library header file) 134, 136
severity level, of diagnostic messages 139
  specifying 140
_ _sfr (extended keyword) 108
  using in #pragma directives. 115
SFR_A (segment). 78
SFR_I (segment) 78
SFR_ID (segment) 78

SFR_N (segment) . . . 78
SFR_Z (segment) . . . 79
short addressing . . . 10
short (data type) . . . 67
signal.h (library header file) . . . 136
signed char (data type) . . . 67–68
  specifying . . . 87
--silent (compiler option) . . . 99
silent operation, specifying . . . 99
size optimization, specifying . . . 100–101
size_t (integer type) . . . 70, 134
skeleton code, creating for assembler language interface . 37
small (data model) . . . 10
_small_write (library function) . . . 59
special function registers . . . 22
special function types, overview . . . 5
speed optimization, specifying . . . 98
sprintf (library function) . . . 59
sscanf (library function) . . . 60
sstream (library header file) . . . 137
stack . . . 13, 49
  advantages and problems using . . . 14
  contents of . . . 14
  internal data . . . 75
  saving space . . . 64
  size . . . 50
  specified in linker command file . . . 44
stack parameters . . . 31–32
stack pointer . . . 14
standard error . . . 97
standard output, specifying . . . 97
Standard Template Library (STL), missing from Embedded C++ . . . 7
startup code . . . 51
  See also cstartup
statements, in implementation-defined behavior . . . 148
static functions . . . 63
static memory . . . 15
stdarg.h (library header file) . . . 134, 136
_ _STDC_ _ (predefined symbol) . . . 119
_ _STDC_VERSION_ _ (predefined symbol) . . . 119
stddef.h (library header file) . . . 134, 136
stderr . . . 61, 97
stdexcept (library header file) . . . 137
stdin . . . 61
stdio.h (library header file) . . . 134, 136
stdlib.h (library header file) . . . 134, 136
stdout . . . 61, 97
streambuf (library header file) . . . 137
streams, supported in Embedded C++ . . . 7
--strict_ansi (compiler option) . . . 99
string (library header file) . . . 137
strings, supported in Embedded C++ . . . 7
string.h (library header file) . . . 134, 136
Stroustrup, Bjarne . . . xv
strstream (library header file) . . . 137
structures . . . 70
  anonymous . . . 23
  in implementation-defined behavior . . . 147
  placing in memory type . . . 19
symbols
  predefined
    overview of . . . 5
  preprocessor, defining . . . 88
syntax, extended keywords . . . 104

T

target identifier (predefined symbol) . . . 119
templates, missing from Embedded C++ . . . 7
terminology, of this guide . . . xv
this (pointer) . . . 40
_ _TID_ _ (predefined symbol) . . . 119
_ _TIME_ _ (predefined symbol) . . . 120
time-critical routines . . . 5, 121
time.h (library header file) . . . 136
tips, programming . . . 63
trademarks . . . ii
translation, in implementation-defined behavior . . . 143
type checking, of function parameters . . . 64

type-safe memory management 6
type_attribute (#pragma directive) 115
typographic conventions xvi

U

uninitialized variables 20
unions 23, 70
in implementation-defined behavior 147
unsigned char (data type) 67–68
changing to signed char 87
unsigned int (data type) 67–68
unsigned long long (data type) 68
unsigned long (data type) 68
unsigned short (data type) 67

V

-v (compiler option) 99, 119
mapping of dsPIC cores 9
variables
as pointers 18
auto 13–14, 64
defined inside a function 13
global, placement in memory 15
local. *See* auto variables
located in memory 21
module-local 63
placing at absolute addresses 21
placing in named segments 21
static, placement in memory 15
uninitialized 20
vector (#pragma directive) 25, 116
_ _VER_ _ (predefined symbol) 120
version, of compiler 120

W

warnings 139
classifying 90
disabling 96
exit code 100
--warnings_affect_exit_code (compiler option) 85
--warnings_are_errors (compiler option) 100
wchar.h (library header file) 136
wchar_t (data type) 67
wctype.h (library header file) 136
_ _write (library function) 61
write formatter, selecting 59
_ _writechar (library function) 61

X

X memory 16
XLINK options
-A 58
_ _xmem (extended keyword) 108
using in #pragma directives 115
XMEM_A (segment) 79
XMEM_I (segment) 79
XMEM_ID (segment) 79
XMEM_N (segment) 80
XMEM_Z (segment) 80

Y

Y memory 16
_ _ymem (extended keyword) 108
using in #pragma directives 115
YMEM_A (segment) 80
YMEM_I (segment) 80
YMEM_ID (segment) 81
YMEM_N (segment) 81
YMEM_Z (segment) 81

Z

-z (compiler option) 100–101

Symbols

#include file paths, specifying 92
#pragma directives
bitfields 68, 110
constseg 110
dataseg 111
diag_default 111
diag_error 111
diag_remark 112
diag_suppress 112
diag_warning 112
inline 112
language 112
location 113
example 26
object_attribute 20, 113
optimize 114
overriding default behaviors 10
overview 5
rtmodel 114
syntax 110
type_attribute 115
vector 25, 116
-A (XLINK option) 58
-D (compiler option) 88
-e (compiler option) 90
-f (compiler option) 91
-I (compiler option) 92
-l (compiler option) 38, 93
-o (compiler option) 96
-r (compiler option) 89, 98
-s (compiler option) 98
-v (compiler option) 99, 119
mapping of dsPIC cores 9
-z (compiler option) 100–101
--char_is_signed (compiler option) 87
--cpu (compiler option) 87
mapping of dsPIC cores 9
--data_model (compiler option) 88
--debug (compiler option) 89, 98
--diag_error (compiler option) 89
--diag_remark (compiler option) 89
--diag_suppress (compiler option) 90
--diag_warning (compiler option) 90
--ec++ (compiler option) 91
--IARStyleMessages (compiler option) 93
--library_module (compiler option) 93
--migration_preprocessor_extensions (compiler option) . . 94
--module_name (compiler option) 94
--no_code_motion (compiler option) 95
--no_cse (compiler option) 95
--no_inline (compiler option) 95
--no_unroll (compiler option) 96
--no_warnings (compiler option) 96
--only_stdout (compiler option) 97
--preprocess (compiler option) 97
--remarks (compiler option) 98
--require_prototypes (compiler option) 98
--silent (compiler option) 99
--strict_ansi (compiler option) 99
--warnings_affect_exit_code (compiler option) 85, 100
--64bit_doubles (compiler option) 101
?C_EXIT (assembler label) 62
?C_GETCHAR (assembler label) 62
?C_PUTCHAR (assembler label) 62
?C_VIRTUAL_IO (assembler label) 62
_ _asm (intrinsic function) 124
_ _clear_watchdog_timer (intrinsic function) 124
_ _close (library function) 61
_ _constptr (extended keyword) 105
using in #pragma directives 115
_ _cplusplus (predefined symbol) 118
_ _CPU_ _ (predefined symbol) 118–119
_ _data_model (runtime model attribute) 36
_ _DATE_ _ (predefined symbol) 118
_ _disable_interrupt (intrinsic function) 124
_ _double_size (runtime model attribute) 36

_ _EI (intrinsic function) 124
_ _embedded_cplusplus (predefined symbol) 118
_ _enable_interrupt (intrinsic function) 124
_ _FILE_ _ (predefined symbol) 118
_ _func (extended keyword). 105
_ _IAR_SYSTEMS_ICC_ _ (predefined symbol). 119
_ _ICCDSPIC_ _ (predefined symbol) 119
_ _interrupt (extended keyword) 25, 105–106, 108
using in #pragma directives. 115–116
_ _intrinsic (extended keyword). 106
_ _int16_t abs_s (intrinsic function) 125
_ _int16_t add (intrinsic function) 125
_ _int16_t div_s (intrinsic function). 125
_ _int16_t extract_h (intrinsic function) 125
_ _int16_t extract_l (intrinsic function) 125
_ _int16_t mac_r (intrinsic function) 125
_ _int16_t msu_r (intrinsic function) 126
_ _int16_t mult (intrinsic function) 126
_ _int16_t mult_r (intrinsic function). 126
_ _int16_t negate (intrinsic function). 126
_ _int16_t norm_l (intrinsic function) 126
_ _int16_t norm_s (intrinsic function) 126
_ _int16_t round (intrinsic function) 126
_ _int16_t round_ub (intrinsic function) 127
_ _int16_t shl (intrinsic function). 127
_ _int16_t shr (intrinsic function) 127
_ _int16_t shr_r (intrinsic function) 127
_ _int16_t sub (intrinsic function) 127
_ _int16_t __xmem * add_br (intrinsic function) 127
_ _int32_t L_abs (intrinsic function) 128
_ _int32_t L_add (intrinsic function). 128
_ _int32_t L_deposit_h (intrinsic function) 128
_ _int32_t L_deposit_l (intrinsic function) 128
_ _int32_t L_mac (intrinsic function) 128
_ _int32_t L_msu (intrinsic function) 128
_ _int32_t L_mult (intrinsic function) 129
_ _int32_t L_negate (intrinsic function) 129
_ _int32_t L_shl (intrinsic function) 129
_ _int32_t L_shr (intrinsic function) 129
_ _int32_t L_shr_r (intrinsic function). 129
_ _int32_t L_sub (intrinsic function) 130
_ _int40_t LL_add (intrinsic function). 130
_ _int40_t LL_mac (intrinsic function) 130
_ _int40_t LL_msu (intrinsic function) 130
_ _int40_t LL_negate (intrinsic function) 130
_ _int40_t LL_sub (intrinsic function). 131
_ _LINE_ _ (predefined symbol) 119
_ _long_double_size (runtime model attribute) 36
_ _low_level_init, customizing 54
_ _lseek (library function) 61
_ _mem (extended keyword) 106
using in #pragma directives. 115
_ _mem * add_mod (intrinsic function). 131
_ _MEMORY_MODEL_ _ (predefined symbol) 118
_ _monitor (extended keyword) 106
using in #pragma directives. 115
_ _no_init (extended keyword) 20, 107
using in #pragma directives. 113
_ _no_operation (intrinsic function) 131
_ _open (library function) 61
_ _ptr (extended keyword) 108
_ _read (library function) 61
_ _readchar (library function) 61
_ _require (intrinsic function) 131
_ _reset (intrinsic function) 132
_ _root (extended keyword) 108
using in #pragma directives. 113
_ _rt_version (runtime model attribute) 36
_ _sfr (extended keyword) 108
using in #pragma directives. 115
_ _STDC_ _ (predefined symbol) 119
_ _STDC_VERSION_ _ (predefined symbol). 119
_ _TID_ _ (predefined symbol) 119
_ _TIME_ _ (predefined symbol) 120
_ _VER_ _ (predefined symbol) 120
_ _write (library function) 61
_ _writechar (library function). 61
_ _xmem (extended keyword) 108

using in #pragma directives. 115
_ _ymem (extended keyword) 108
using in #pragma directives. 115
_formatted_write (library function) 59
_medium_write (library function) 59
_small_write (library function) 59

Numerics

4-byte (floating-point format) 69
--64bit_doubles (compiler option) 101
8-byte (floating-point format) 69