

ARM® IAR C/C++ Compiler

Reference Guide

for Advanced RISC Machines Ltd's

ARM Cores

COPYRIGHT NOTICE

© Copyright 1999–2004 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.

ARM and Thumb are registered trademarks of Advanced RISC Machines Ltd.

Microsoft and Windows are registered trademarks of Microsoft Corporation. Intel and Pentium are registered trademarks and XScale a trademark of Intel Corporation.

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

EDITION NOTICE

Tenth edition: February 2004

Part number: CARM-10

This guide applies to version 4.x of the ARM IAR Embedded Workbench™.

Brief contents

Tables	xiii
Preface	xv
Part 1. Using the compiler	1
Getting started	3
Data storage	11
Functions	15
Placing code and data	23
The DLIB runtime environment	37
Assembler language interface	67
Using C++	83
Efficient coding for embedded applications	87
Part 2. Compiler reference	101
Data representation	103
Segment reference	113
Compiler options	121
Extended keywords	147
Pragma directives	153
Predefined symbols	165
Intrinsic functions	171
Library functions	177
Implementation-defined behavior	183

IAR language extensions	195
Diagnostics	207
Index	209

Contents

Tables	xiii
Preface	xv
Who should read this guide	xv
How to use this guide	xv
What this guide contains	xvi
Other documentation	xvii
Further reading	xvii
Document conventions	xviii
Typographic conventions	xviii
 Part I. Using the compiler	1
Getting started	3
IAR language overview	3
Building applications—an overview	4
Compiling	4
Linking	4
Basic settings for project configuration	5
Processor variant	5
CPU mode	6
Interworking	6
VFP and floating-point arithmetic	7
Byte order	7
Optimization for speed and size	7
Runtime environment	8
Special support for embedded systems	9
Extended keywords	9
Predefined symbols	10
Special function types	10
Header files for I/O	10
Accessing low-level features	10

Data storage	11
Introduction	11
The stack and auto variables	11
Dynamic memory on the heap	13
Functions	15
ARM and Thumb code	15
Keywords for functions	15
Execution in RAM	16
Interrupt functions	17
Interrupts and fast interrupts	17
Nested interrupts	18
Software interrupts	19
Installing interrupt functions	20
Interrupt operations	20
Monitor functions	21
C++ and special function types	22
Function directives	22
Placing code and data	23
Segments and memory	23
What is a segment?	23
Placing segments in memory	24
Customizing the linker command file	25
Data segments	27
Static memory segments	27
The stack	29
The heap	30
Located data	31
Code segments	31
Startup code	31
Normal code	32
Exception vectors	32

C++ dynamic initialization	33
Efficient usage of segments and memory	33
Controlling data and function placement	33
Creating user-defined segments	35
The linked result of code and data placement	35
The DLIB runtime environment	37
Introduction to the runtime environment	37
Runtime environment functionality	37
Library selection	38
Situations that require library building	39
Library configurations	39
Using a prebuilt library	40
Customizing a prebuilt library without rebuilding	42
Modifying a library by setting library options	43
Choosing printf formatter	43
Choosing scanf formatter	44
Overriding library modules	45
Building and using a customized library	47
Setting up a library project	47
Modifying the library functionality	47
Using a customized library	48
System startup and termination	49
System startup	50
System termination	50
Customizing system initialization	51
__low_level_init	51
Modifying the cstartup file	51
Standard streams for input and output	52
Implementing low-level character input and output	52
Configuration symbols for printf and scanf	53
Customizing formatting capabilities	54

File input and output	54
Locale	55
Locale support in prebuilt libraries	56
Customizing the locale support	56
Switching locales at runtime	57
Environment interaction	57
Signal and raise	58
Time	59
Strtod	59
Assert	59
C-SPY Debugger runtime interface	60
Low-level debugger runtime interface	60
The debugger terminal I/O window	61
Checking module consistency	61
Runtime model attributes	61
Using runtime model attributes	62
Predefined runtime attributes	62
User-defined runtime model attributes	63
Implementation of cstartup	63
Modules and segment parts	64
Added C functionality	65
Assembler language interface	67
Mixing C and assembler	67
Intrinsic functions	67
Mixing C and assembler modules	68
Inline assembler	69
Calling assembler routines from C	70
Creating skeleton code	70
Compiling the code	71
Calling assembler routines from C++	72
Calling convention	73
Function declarations	73
C and C++ linkage	73

Preserved versus scratch registers	74
Function entrance	75
Function exit	77
Return address handling	77
Examples	78
Calling functions	79
Call frame information	81
Using C++	83
Overview	83
Standard Embedded C++	83
Extended Embedded C++	84
Enabling C++ support	84
Feature descriptions	85
Classes	85
Functions	85
Templates	85
Variants of casts	86
Mutable	86
Namespace	86
The STD namespace	86
Efficient coding for embedded applications	87
Taking advantage of the compilation system	87
Controlling compiler optimizations	88
Selecting data types and placing data in memory	92
Using efficient data types	92
Rearranging elements in a structure	93
Anonymous structs and unions	94
Writing efficient code	95
Saving stack space and RAM memory	96
Function prototypes	96
Integer types and bit negation	97
Protecting simultaneously accessed variables	98
Accessing special function registers	98

Non-initialized variables	99
Part 2. Compiler reference	101
Data representation	103
Alignment	103
Byte order	103
Basic data types	104
Integer types	104
Floating-point types	105
Pointer types	107
Code pointers	107
Data pointers	107
Casting	107
Structure types	108
Alignment	108
General layout	108
Packed structure types	109
Type and object attributes	110
Type attributes	110
Object attributes	110
Declaring objects in C source files	111
Data types in C++	111
Segment reference	113
Summary of segments	113
Descriptions of segments	114
Compiler options	121
Setting command line options	121
Specifying parameters	122
Specifying environment variables	123
Error return codes	123

Options summary	123
Descriptions of options	126
Extended keywords	147
Using extended keywords	147
Summary of extended keywords	147
Descriptions of extended keywords	148
Pragma directives	153
Summary of pragma directives	153
Descriptions of pragma directives	154
Predefined symbols	165
Summary of predefined symbols	165
Descriptions of predefined symbols	166
Intrinsic functions	171
Intrinsic functions summary	171
Descriptions of intrinsic functions	172
Library functions	177
Introduction	177
Header files	177
Library object files	177
Reentrancy	178
IAR DLIB Library	178
C header files	179
C++ header files	179
Library functions as intrinsic functions	182
Implementation-defined behavior	183
Descriptions of implementation-defined behavior	183
Translation	183
Environment	184
Identifiers	184
Characters	184

Integers	186
Floating point	186
Arrays and pointers	187
Registers	187
Structures, unions, enumerations, and bitfields	187
Qualifiers	188
Declarators	188
Statements	188
Preprocessing directives	188
IAR DLIB Library functions	190
IAR language extensions	195
Why should language extensions be used?	195
Descriptions of language extensions	195
Diagnostics	207
Message format	207
Severity levels	207
Setting the severity level	208
Internal error	208
Index	209

Tables

1: Typographic conventions used in this guide	xviii
2: Command line options for specifying library and dependency files	8
3: XLINK segment memory types	24
4: Memory layout of a target system (example)	25
5: Segment name suffixes	28
6: Exception stacks	30
7: Segment groups	32
8: Library configurations	39
9: Prebuilt libraries	40
10: Customizable items	42
11: Formatters for printf	44
12: Formatters for scanf	45
13: Descriptions of printf configuration symbols	53
14: Descriptions of scanf configuration symbols	54
15: Low-level I/O files	55
16: Functions with special meanings when linked with debug info	60
17: Example of runtime model attributes	62
18: Predefined runtime model attributes	63
19: Registers used for passing parameters	75
20: VFP registers used for passing parameters in non-interworking mode	76
21: Registers used for returning values	77
22: VFP registers used for returning values in non-interworking mode	77
23: Call frame information resources defined in a names block	81
24: Compiler optimization levels	88
25: Integer types	104
26: Floating-point types	105
27: Segment summary	113
28: Environment variables	123
29: Error return codes	123
30: Compiler options summary	123
31: Generating a list of dependencies (--dependencies)	128

32: Generating a compiler list file (-l)	135
33: Directing preprocessor output to file (--preprocess)	141
34: Specifying speed optimization (-s)	143
35: Specifying size optimization (-z)	145
36: Extended keywords summary	147
37: Pragma directives summary	153
38: Predefined symbols summary	165
39: Predefined symbols for inspecting the CPU mode	167
40: Values for specifying different CPU cores in __TID__	168
41: Intrinsic functions summary	171
42: Traditional standard C header files—DLIB	179
43: Embedded C++ header files	180
44: Additional Embedded C++ header files—DLIB	180
45: Standard template library header files	180
46: New standard C header files—DLIB	181
47: Message returned by strerror()—IAR DLIB library	193

Preface

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

Who should read this guide

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

- The architecture and instruction set of the ARM core. Refer to the documentation from Advanced RISC Machines Ltd for information about the ARM core
- The C or C++ programming language
- Application development for embedded systems
- The operating system of your host machine.

How to use this guide

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

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

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

What this guide contains

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

Part 1. Using the compiler

- *Getting started* gives the information you need to get started using the ARM IAR C/C++ Compiler for efficiently developing your application.
- *Data storage* describes how data can be stored in memory.
- *Functions* describes the different ways code can be generated, and introduces the concept of function type attributes, such as interrupt functions.
- *Placing code and data* describes the concept of segments, introduces the linker command file, and describes how code and data are placed in memory.
- *The DLIB runtime environment* gives an overview of the runtime libraries and how they can be customized. The chapter also describes system initialization and introduces the `cstartup` file.
- *Assembler language interface* contains information required when parts of an application are written in assembler language. This includes the calling convention.
- *Using C++* gives an overview of the two levels of C++ support: The industry-standard EC++ and IAR Extended EC++.
- *Efficient coding for embedded applications* gives hints about how to write code that compiles to efficient code for an embedded application.

Part 2. Compiler reference

- *Data representation* describes the available data types, pointers, and structure types. It also describes the concepts of attributes.
- *Segment reference* gives reference information about the compiler's use of segments.
- *Compiler options* explains how to set the compiler options, gives a summary of the options, and contains detailed reference information for each compiler option.
- *Extended keywords* gives reference information about each of the ARM-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 be used for accessing ARM-specific low-level features.
- *Library functions* gives an introduction to the C or C++ library functions, and summarizes the header files.
- *Implementation-defined behavior* describes how the ARM IAR C/C++ Compiler handles the implementation-defined areas of the C language standard.
- *IAR language extensions* describes the IAR extensions to the ISO/ANSI standard for the C programming language.
- *Diagnostics* describes how the compiler's diagnostic system works.

Other documentation

The complete set of IAR Systems development tools for the ARM core 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 *ARM® IAR Embedded Workbench™ IDE User Guide*
- Programming for the ARM IAR Assembler, refer to the *ARM® IAR Assembler Reference Guide*
- Using the IAR XLINK Linker™, the IAR XAR Library Builder™, and the IAR XLIB Librarian™, refer to the *IAR Linker and Library Tools Reference Guide*
- Using the IAR DLIB Library, refer to the online help system, available from the ARM IAR Embedded Workbench IDE **Help** menu
- *ARM® IAR Embedded Workbench Migration Guide* gives hints for porting application code and projects to this product version.

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:

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

We recommend that you visit the following websites:

- The Advanced RISC Machines Ltd website, **www.arm.com**, contains information and news about the ARM core.
- The IAR website, **www.iar.com**, holds application notes and other product information.

- Finally, the Embedded C++ Technical Committee website, www.caravan.net/ec2plus, contains information about the Embedded C++ standard.

Document conventions

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

TYPOGRAPHIC CONVENTIONS

This guide uses the following typographic conventions:




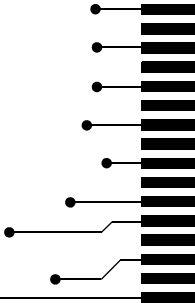
Style	Used for
computer	Text that you enter or that appears on the screen.
<i>parameter</i>	A label representing the actual value you should enter as part of a command.
[option]	An optional part of a command.
{a b c}	Alternatives in a command.
bold	Names of menus, menu commands, buttons, and dialog boxes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

Table 1: Typographic conventions used in this guide

Part I. Using the compiler

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

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





Getting started

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

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

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

IAR language overview

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

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

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

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

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

Building applications—an overview

A typical application is built from a number of source files and libraries. The source files can be written in C, C++, or assembler language, and can be compiled into object files by the ARM IAR C/C++ Compiler or the ARM IAR Assembler.

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

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



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

COMPILING

In the command line interface, the following line compiles the source file `myfile.c` into the object file `myfile.r79` using the default settings:

```
iccarm myfile.c
```

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

LINKING

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

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

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

```
xlink myfile.r79 myfile2.r79 -s __program_start -f lnkarm.xcl  
dl4tpann18n.r79 -o aout.a79 -Felf/dwarf
```

In this example, `myfile.r79` and `myfile2.r79` are object files, `lnkarm.xcl` is the linker command file, and `dl4tpann18n.r79` is the runtime library. The option `-s` specifies the label where the application starts. The option `-o` specifies the name of the output file, and the option `-F` must be used for specifying output files.

The IAR XLINK Linker produces output after your specifications. Choose the output format that suits your purpose. You might want to load the output to a debugger—which means that you need output with debug information. Alternatively, you might want to load the output to a PROM programmer—in which case you need output without debug information, such as Intel-hex or Motorola S-records.

Basic settings for project configuration

This section gives an overview of the basic settings for the project setup that are needed to make the compiler generate optimal code for the ARM device you are using. You can specify the options either from the command line interface or in the IAR Embedded Workbench IDE. For details about how to set options, see *Setting command line options*, page 121, and the *ARM® IAR Embedded Workbench™ IDE User Guide*.

The basic settings available for the ARM core are:

- Processor variant
- CPU mode
- Interworking
- Floating-point arithmetic
- Byte order
- Optimization settings
- Runtime library.

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

PROCESSOR VARIANT

The ARM IAR C/C++ Compiler supports several different ARM cores and derivatives based on these cores. All supported cores support Thumb instructions and 64-bit multiply instructions. The object code that the compiler generates is not binary compatible. Therefore it is crucial to specify a processor option to the compiler. The default core is ARM7TDMI.



See the *ARM® IAR Embedded Workbench™ IDE User Guide* for information about setting the **Processor variant** option in the IAR Embedded Workbench.



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

The following cores and processor macrocells are recognized:

- ARM7TDMI
- ARM7TDMI-S
- ARM710T
- ARM720T
- ARM740T
- ARM9TDMI
- ARM920T
- ARM922T
- ARM940T
- ARM9E
- ARM9E-S
- ARM926EJ-S
- ARM946E-S
- ARM966E-S
- ARM10E
- ARM1020E
- ARM1022E
- ARM1026EJ-S
- XScale

CPU MODE

The ARM IAR C/C++ Compiler supports two CPU modes: ARM and Thumb.

All functions and function pointers will compile in the mode that you specify, except those explicitly declared `__arm` or `__thumb`.



See the *ARM® IAR Embedded Workbench™ IDE User Guide* for information about setting the **Processor variant** or **Chip** option in the IAR Embedded Workbench.



Use the `--cpu_mode` option to specify the CPU mode for your project; see `--cpu_mode`, page 126, for syntax information.

INTERWORKING

When code is compiled with the `--interwork` option, ARM and Thumb code can be freely mixed. Interworking library functions can be called from both ARM and Thumb code. The interworking libraries are slightly larger than non-interworking libraries.



See the *ARM® IAR Embedded Workbench™ IDE User Guide* for information about setting the **Generate interwork code** option in the IAR Embedded Workbench.



Use the `--interwork` option to specify interworking capabilities for your project; see `--interwork`, page 134, for syntax information.

VFP AND FLOATING-POINT ARITHMETIC

If you are using an ARM core that contains a Vector Floating Point (VFP) coprocessor, you can use the `--fpu` option to generate code that carries out floating-point operations utilizing the coprocessor, instead of using the software floating-point library routines.



See the *ARM® IAR Embedded Workbench™ IDE User Guide* for information about setting the **FPU** option in the IAR Embedded Workbench.



Use the `--fpu` option to specify interworking capabilities for your project; see `--fpu`, page 133, for syntax information.

BYTE ORDER

The ARM IAR C/EC++ Compiler supports the big-endian and little-endian byte order. All user and library modules in your application must use the same byte order.



See the *ARM® IAR Embedded Workbench™ IDE User Guide* for information about setting the **Endian mode** option in the IAR Embedded Workbench.



Use the `--endian` option to specify the byte order for your project; see `--endian`, page 132, for syntax information.

OPTIMIZATION FOR SPEED AND SIZE

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

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

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

For details about compiler optimizations, see *Controlling compiler optimizations*, page 88. For more information about efficient coding techniques, see the chapter *Efficient coding for embedded applications*.

RUNTIME ENVIRONMENT

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

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

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

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

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



Choosing a runtime library in the IAR Embedded Workbench

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

Note that for the DLIB library there are two different configurations—Normal and Full—which include different levels of support for locale, file descriptors, multibytes, et cetera. See *Library configurations*, page 39, for more information.

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



Choosing a runtime library from the command line

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

Command line	Description
<code>-I\arm\inc</code>	Specifies the include paths
<code>libraryfile.r79</code>	Specifies the library object file

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

Command line	Description
<code>-D_DLIB_CONFIG_FILE=</code> <code>C:\...\configfile.h</code>	Specifies the library configuration file

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

For a list of all prebuilt library object files for the IAR DLIB Library, see Table 9, *Prebuilt libraries*, page 40. The table also shows how the object files correspond to the dependent project options, and the corresponding configuration files. Make sure to use the object file that matches your other project options.

Setting library and runtime environment options

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

- The formatters used by the functions `printf`, `scanf`, and their variants, see *Modifying a library by setting library options*, page 52.
- The size of the stack and the heap, see *The stack*, page 38, and *The heap*, page 38, respectively.

Special support for embedded systems

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

EXTENDED KEYWORDS

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



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

The command line option `-e` makes the extended keywords available, and reserves them so that they cannot be used as variable names. See, *-e*, page 131 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 ARM IAR C/C++ Compiler. They are consistent with ISO/ANSI C, and are very useful when you want to make sure that the source code is portable.

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

PREDEFINED SYMBOLS

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

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

SPECIAL FUNCTION TYPES

The special hardware features of the ARM core are supported by the compiler's special function types: software interrupts, interrupts, fast interrupts, and monitor. With these functions, you can write a complete application without having to write any of them in assembler language.

For detailed information, see the chapter *Functions*.

HEADER FILES FOR I/O

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

For an example, see *Accessing special function registers*, page 98.

ACCESSING LOW-LEVEL FEATURES

For hardware-related parts of your application, accessing low-level features is essential. The ARM IAR C/C++ Compiler supports several ways of doing this: intrinsic functions, mixing C and assembler modules, and inline assembler. For information about the different methods, see *Mixing C and assembler*, page 67.

Data storage

This chapter gives a brief introduction to the memory layout of the ARM core and the fundamental ways data can be stored in memory: on the stack, in static (global) memory, or in heap memory.

Introduction

The ARM core can address 4 Gbytes of continuous memory, ranging from 0x00000000 to 0xFFFFFFFF. Different types of physical memory can be placed in the memory range. A typical application will have both read-only memory (ROM) and read/write memory (RAM). In addition, some parts of the memory range contain processor control registers and peripheral units.

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

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

The stack and auto variables

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

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

The stack can contain:

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

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

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

Advantages

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

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

Potential problems

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

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

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

Dynamic memory on the heap

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

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

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

Potential problems

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

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

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

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

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

Functions

This chapter contains information about functions. First, you get an overview of ARM and Thumb code generation. Execution in RAM and the special function types interrupt, software interrupt, fast interrupts, and monitor are described, including how to declare C++ member functions by using special function types. Then, it is shown how to place functions into named segments. The last section describes the function directives.

ARM and Thumb code

The ARM IAR C/C++ Compiler can generate code for either the 32-bit ARM or the 16-bit Thumb instruction set. Use the `--cpu_mode` option to specify which instruction set should be used for your project. For individual functions, it is possible to override the project setting by using the extended keywords `__arm` and `__thumb`. You can freely mix ARM and thumb code in the same application, as long as the `--interwork` option is used.

There are no code or segment size limitations. When performing function calls, the compiler always attempts to generate the most efficient assembler language instruction or instruction sequence available. As a result, 4 Gbytes of continuous memory in the range `0x0-0xFFFFFFFF` can be used for placing code.

The size of all code pointers is 4 bytes. There are restrictions to implicit and explicit casts from code pointers to data pointers or integer types or vice versa. For further information about the restrictions, see *Pointer types*, page 107.

In the chapter *Assembler language interface*, the generated code is studied in more detail in the description of calling C functions from assembler language and vice versa.

Keywords for functions

The ARM IAR C/C++ Compiler provides a set of extended keywords that can be used when functions are declared. The keywords 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: `__arm`, `__fiq`, `__interwork`, `__irq`, `__monitor`, `__swi`, and `__thumb`.
- Keywords that only control the defined function: `__root` and `__ramfunc`.

For reference information about these keywords, see the chapter *Extended keywords* in *Part 2. Compiler reference*.

Syntax

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

```
__irq __arm 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 because they do not affect the way an object or function is used.

Execution in RAM

The `__ramfunc` keyword makes a function execute in RAM. The function is copied from ROM to RAM by `cstartup`, see *System startup and termination*, page 49.

The keyword is specified before the return type:

```
__ramfunc void foo(void);
```

If a function declared `__ramfunc` tries to access ROM, the compiler will issue a warning.

If the whole memory area used for code and constants is disabled—for example, when the whole flash memory is being erased—only functions and data stored in RAM may be used. Interrupts must be disabled unless the interrupt vector and the interrupt service routines are also stored in RAM.

String literals and other constants can be avoided by using initialized variables. For example, the following lines:

```
const int myc[] = { 10, 20 };    // myc initializer in
                                // DATA_C (ROM)
msg("Hello");                  // String literal in
                                // DATA_C (ROM)

may be rewritten to:
static int myc[] = { 10, 20 };   // Initialized by cstartup
static char hello[] = "Hello";  // Initialized by cstartup
msg(hello);                     // hello stored in DATA_I
                                // (RAM)
```

If the option `--segment` is used for renaming segments, the new segments for storing `__ramfunc` functions must be declared in the linker command file. The compiler option `--segment code=MYSEG` is used in this example.

All functions without attributes are linked in segment MYSEG_T. Functions declared `__ramfunc` are linked in segment MYSEG_I, with initializers in segment MYSEG_ID. The following segment options need to be added in the linker command file:

```
-Z (CODE) MYSEG=ROMSTART-ROMEND      // Ordinary functions
-Z (DATA) MYSEG_I=RAMSTART-RAMEND     // Functions stored
                                      // in RAM.
-Z (CONST) MYSEG_ID=ROMSTART-ROMEND  // Initializer for
                                      // MYSEG_I.
-QMYSEG_I=MYSEG_ID                   // Instruct XLINK to place
                                      // all data content of
                                      // MYSEG_I in MYSEG_ID
```

The contents of MYSEG_ID is copied to MYSEG_I by the function `__segment_init`, called by `cstartup`; for details, see `Segment_init.c` and `Segment_init.h` in the `arm\src\lib` directory.

Interrupt functions

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

In general, when an interrupt occurs in the code, the processor simply stops executing the code it runs and starts executing an interrupt routine instead. The processor state prior to the interrupt is stored so that it can be restored at the end of the interrupt routine. This enables the execution of the original code to continue.

The ARM IAR C/C++ Compiler supports interrupts, software interrupts, and fast interrupts. This allows an application to take full advantage of these powerful ARM features without forcing you to implement anything in assembler language.

All interrupt functions must be compiled in ARM mode; if you are using Thumb mode, use the `__arm` extended keyword or the `#pragma type_attribute=__arm` directive to alter the default behavior.

INTERRUPTS AND FAST INTERRUPTS

The interrupt and fast interrupt functions are easy to handle as they do not accept parameters or have a return value.

- An interrupt function is declared by use of the `__irq` extended keyword or the `#pragma type_attribute=__irq` directive. For syntax information, see `__irq`, page 149, and `#pragma type_attribute`, page 162, respectively.
- A fast interrupt function is declared by use of the `__fiq` extended keyword or the `#pragma type_attribute=__fiq` directive. For syntax information, see `__fiq`, page 149, and `#pragma type_attribute`, page 162, respectively.

Note: An interrupt function (`irq`) and a fast interrupt function (`fiq`) must have a return type of `void` and cannot have any parameters. A software interrupt function (`swi`) may have parameters and return values. By default, only four registers, `R0–R3`, can be used for parameters and only the registers `R0–R1` can be used for return values.

NESTED INTERRUPTS

Interrupts are automatically disabled by the ARM core prior to entering an interrupt handler. If an interrupt handler re-enables interrupts, calls functions, and another interrupt occurs, then the return address of the interrupted function—stored in `LR`—is overwritten when the second IRQ is taken. In addition, the contents of `SPSR` will be destroyed when the second interrupt occurs. The `__irq` keyword itself does not save and restore `LR` and `SPSR`. To make an interrupt handler perform the necessary steps needed when handling nested interrupts, the keyword `__nested` must be used in addition to `__irq`. The function prolog—function entrance sequence—that the compiler generates for nested interrupt handlers will switch from IRQ mode to system mode. Make sure that both the IRQ stack and system stack is set up. If you use the default `cstartup.s79` file, both stacks are correctly set up.

Compiler-generated interrupt handlers that allow nested interrupts are supported for IRQ interrupts only. The FIQ interrupts are designed to be serviced quickly, which in most cases mean that the overhead of nested interrupts would be too high.

This example shows how to use nested interrupts with the ARM vectored interrupt controller (VIC):

```
__irq __nested __arm void interrupt_handler(void) {
    void (*interrupt_task)();
    unsigned int vector;

    vector = VICVectAddr;           // Get interrupt vector.
    interrupt_task = (void(*)())vector;

    VICVectAddr = 0;                // Acknowledge interrupt in VIC.

    __enable_interrupt();           // Allow other IRQ interrupts
                                    // to be serviced from this
                                    // point.

    (*interrupt_task)();            // Execute the task associated
                                    // with this interrupt.
}
```

SOFTWARE INTERRUPTS

Software interrupt functions are slightly more complex, in the way that they accept arguments and have return values. The `__swi` keyword also expects a software interrupt number which is specified with the `#pragma swi_number=number` directive. A `__swi` function can for example be declared in the following way:

```
#pragma swi_number=0x23
__swi __arm int swi_function(int a, int b);
```

The `swi_number` is used as an argument to the generated assembler `SWI` instruction, and can be used to select one software interrupt function in a system containing several such functions.

For additional information, see `__swi`, page 151, and `#pragma swi_number`, page 162, respectively.

Software interrupt handler

The software interrupt handler is an assembler language-written function that calls the correct software interrupt function depending on the specified `swi_number`. The software interrupt handler must be written in assembler because it must have access to the link register, `LR`. IAR provides a software interrupt handler in the IAR C libraries that can be used, but you can also write your own if needed.

The software interrupt handler extracts the software interrupt number from the assembler `SWI` instruction located in `SP-4` in ARM mode and `SP-2` in Thumb mode. The decoded software interrupt number is then used to find the corresponding C language software interrupt function, which is called. If the interrupt first encountered should not be handled, the IAR software interrupt handler continues to the label `__next_swi_in_chain`, which can be used to chain several software interrupt handlers.

If third party libraries using software interrupts are used, their software interrupt handler must be used together with your software interrupt handler. The label `__next_swi_in_chain` refers to the software interrupt handler that will be called if the software interrupt request cannot be satisfied by the IAR software interrupt handler.

The IAR software interrupt handler's entry point `__iar_swi_handler` and the public label `__next_swi_in_chain` are declared in the header file `\inc\arm_interrupt.h`. This file should be included in your program if the IAR software interrupt handler is used.

The source code of the IAR software interrupt handler is available in the file `\src\lib\swi_handler.s79`. If you need to write your own software interrupt handler, this file can be used as a template.

INSTALLING INTERRUPT FUNCTIONS

All interrupt functions and software interrupt handlers must be installed into the vector table. This can be done directly in C or assembler language. The following C function, `install_handler`, inserts a branch instruction to the function `function` at the vector address `vector` and returns the old contents of the vector.

```
unsigned int
install_handler(unsigned int *vector, unsigned int function)
{
    unsigned int vec, old_vec;

    vec = ((function - (unsigned int)vector - 8) >> 2);

    old_vec = *vector;
    vec |= 0xea000000; /* add opcode for B instruction */
    *vector = vec;

    old_vec &= ~0xea000000;
    old_vec = ( old_vec << 2 ) + (unsigned int)vector + 8;

    return(old_vec);
}
```

Note that the branch instruction must be sufficient to reach the interrupt function.

The chapter *Assembler language interface* contains more information about the runtime environment used by interrupt routines. See the ARM core documentation for more information about the interrupt vector table.

INTERRUPT OPERATIONS

An interrupt function is called when an external event occurs. Normally it is called immediately while another function is executing. When the interrupt function has finished executing, it returns to the original function. It is imperative that the environment of the interrupted function is restored; this includes the value of processor registers and the processor status register.

When an interrupt occurs, the following actions are performed:

- The operating mode is changed corresponding to the particular exception
- The address of the instruction following the exception entry instruction is saved in R14 of the new mode
- The old value of the CPSR is saved in the SPSR of the new mode
- Interrupt requests are disabled by setting bit 7 of the CPSR and, if the exception is a fast interrupt, further fast interrupts are disabled by setting bit 6 of the CPSR
- The PC is forced to begin executing at the relevant vector address.

For example, if an interrupt for vector 0x18 occurs, the processor will start to execute code at address 0x18. The memory area that is used as start location for interrupts is called the interrupt vector table. The content of the interrupt vector is normally a branch instruction jumping to the interrupt routine.

Note: If the interrupt function enables interrupts, the special processor registers needed to return from the interrupt routine must be assumed to be destroyed. For this reason they must be stored by the interrupt routine to be restored before it returns. This is handled automatically if the `__nested` keyword is used.

MONITOR FUNCTIONS

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

Note: If a monitor function is called while the processor is executing in user mode, the function will not be able to change the interrupt settings.

To define a monitor function, the `__monitor` keyword can be used.

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

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

```

For additional information, see `__monitor`, page 149.

C++ AND SPECIAL FUNCTION TYPES

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

Function directives

The function directives `FUNCTION`, `ARGFRAME`, `LOCFRAME`, and `FUNCALL` are generated by the ARM IAR C/C++ 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 with 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 ARM IAR C/C++ Compiler does not use static overlay, as it has no use for it.

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

Placing code and data

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

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

Segments and memory

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

WHAT IS A SEGMENT?

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

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

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

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

Segment memory type

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

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

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

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

Table 3: XLINK segment memory types

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

Placing segments in memory

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

In particular, the linker command file specifies:

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

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.

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

CUSTOMIZING THE LINKER COMMAND FILE

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

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

Range	Type
0x000000–0x00003F	ROM or RAM
0x008000–0x0FFFFFFF	ROM or other non-volatile memory
0x100000–0x7FFFFFFF	RAM or other read/write memory

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

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

Note: In the default linker command file `lnkarm.xcl`, the start and end addresses for ROM and RAM segments are defined using the `-D` directive:

```
-DROMSTART=08000
-DROMEND=FFFFF
-DDRAMSTART=100000
-DDRAMEND=7FFFFFFF
```

The contents of the linker command file

The `arm\config` directory contains ready-made linker command files. In addition, you can find ready-made linker command files adapted to various ARM evaluation boards in `arm\src\examples` sub-directories. The files contain the information required by the linker, and is ready to be used. If, for example, your application uses additional external RAM, you need to add details about the external RAM memory area.

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

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

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

- The CPU used:
`-carm`

This specifies your target core.

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

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

Using the `-Z` command for sequential placement

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

The following illustrates how to use the `-Z` command to place the segment `MYSEGMENTA` followed by the segment `MYSEGMENTB` in `CONST` memory (that is, ROM) in the memory range `0x008000-0xFFFFF`.

```
-Z (CONST) MYSEGMENTA, MYSEGMENTB=008000-0FFFFF
```

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

```
-Z (CONST) MYSEGMENTA=008000-0FFFFF
-Z (CODE) MYCODE
```

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

```
-Z (CONST) MYSMALLSEGMENT=008000-000FFF
-Z (CONST) MYLARGESEGMENT=008000-0FFFFF
```

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

Using the `-P` command for packed placement

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

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

```
-P (DATA) MYDATA=100000-101FFF, 110000-111FFF
```

If your application has an additional RAM area in the memory range `0x10F000-0x10F7FF`, you just add that to the original definition:

```
-P (DATA) MYDATA=100000-101FFF, 10F000-10F7FF, 110000-111FFF
```

Data segments

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

STATIC MEMORY SEGMENTS

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

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

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

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

Segment naming

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

Some of the declared data is placed in non-volatile memory, for example ROM, and some of the data is placed in RAM. For this reason, it is also important to know the XLINK segment memory type of each segment. For more details about segment memory types, see *Segment memory type*, page 24.

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

Categories of declared data	Segment memory type	Suffix
Absolute addressed located constants	CONST	AC
Absolute addressed located data declared <code>__no_init</code>	DATA	AN
Constants	CONST	C
Non-zero initialized data	DATA	I
Initializers for the above	CONST	ID
Non-initialized data	DATA	N
Zero-initialized data	DATA	Z

Table 5: Segment name suffixes

For a summary of all supported segments, see *Summary of segments*, page 113.

Examples

Assume the following examples:

<code>int j;</code> <code>int i = 0;</code>	The variables that are to be initialized to zero when the system starts will be placed in the segment <code>DATA_Z</code> .
<code>__no_init int j;</code>	The non-initialized variables will be placed in the segment <code>DATA_N</code> .
<code>int j = 4;</code>	The non-zero initialized variables will be placed in the segment <code>DATA_I</code> .

Initialized data

When an application is started, the `cstartup` module initializes static and global variables in two steps:

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

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

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

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

```
DATA_I           0x100000-0x1000FF and 0x100200-0x1002FF
DATA_ID          0x020000-0x0201FF
```

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:

```
DATA_I           0x100000-0x1000FF and 0x100200-0x1002FF
DATA_ID          0x020000-0x0200FF and 0x020200-0x0202FF
```

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

Data segments for static memory in the default linker command file

The default linker command file contains the following directives to place the static data segments:

```
// Various constants and initializers
-Z (CONST) INITTAB, DATA_ID, DATA_C=ROMSTART-ROMEND

//Data segments
-Z (DATA) DATA_I, DATA_Z, DATA_N=RAMSTART-RAMEND
```

THE STACK

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

The data segment used for holding the stack is called *CSTACK*. The *cstartup* module initializes the stack pointer to the end of the stack segment.

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

```
-D_CSTACK_SIZE=2000
```

Note that the size is written hexadecimally without the *0x* notation.

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

```
-Z (DATA) CSTACK+_CSTACK_SIZE=RAMSTART-RAMEND
```

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.

Exception stacks

The ARM architecture supports five exception modes which are entered when different exceptions occur. Each exception mode has its own stack to avoid corrupting the System/User mode stack. The table shows proposed stack names for the various exception stacks, but any name can be used.

Processor mode	Proposed stack segment name	Description
Supervisor	<code>SVC_STACK</code>	Operating system stack.
IRQ	<code>IRQ_STACK</code>	Stack for general-purpose (IRQ) interrupt handlers .
FIQ	<code>FIQ_STACK</code>	Stack for high-speed (FIQ) interrupt handlers .
Undefined	<code>UND_STACK</code>	Stack for undefined instruction interrupts. Supports software emulation of hardware coprocessors and instruction set extensions.
Abort	<code>ABT_STACK</code>	Stack for instruction fetch and data access memory abort interrupt handlers.

Table 6: Exception stacks

For each processor mode where a stack is needed, a separate stack pointer must be initialized in your startup code, and segment placement should be done in the linker command file. The IRQ stack is the only exception stack which is preconfigured in the supplied `cstartup.s`⁷⁹ and `lnkarm.xcl` files, but other exception stacks can easily be added.

THE HEAP

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

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

The size and placement of this segment is defined in the linker command file, much in the same way as the size and placement of the stack. This example is taken from the default linker command file:

```
-D_HEAP_SIZE=8000
and
-Z (DATA) HEAP+_HEAP_SIZE=RAMSTART-RAMEND
```



Heap size and standard I/O

If you have excluded `FILE` descriptors from the DLIB runtime environment, like in the Normal configuration, there are no input and output buffers at all. Otherwise, like in the Full configuration, be aware that the size of the input and output buffers is set to 512 bytes in the `stdio` library header file. If the heap is too small, I/O will not be buffered, which is considerably slower than when I/O is buffered. If you execute the application using the simulator driver of the IAR C-SPY Debugger, you are not likely to notice the speed penalty, but it is quite noticeable when the application runs on an ARM core. If you use the standard I/O library, you should set the heap size to a value which accommodates the needs of the standard I/O buffer, for example 1 Kbyte.

LOCATED DATA

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

Code segments

This section contains descriptions of the segments used for storing code, and the runtime environment support of the special function types interrupt, software interrupt, and fast interrupt. For a complete list of all segments, see *Summary of segments*, page 113.

STARTUP CODE

The segment `ICODE` contains code used during system setup (`cstartup`), runtime initialization (`cmain`), and system termination (`cexit`). The system setup code is called from the exception vector 0 (the reset vector). In addition, the segments must be placed into one continuous memory space, which means the `-P` segment directive cannot be used.

In the default linker command file, the following line states that this segment can be placed anywhere in the 0x08000–0xFFFFF memory range:

```
-Z (CODE) ICODE=08000-FFFFF
```

NORMAL CODE

The name of the actual segments are *NAME_SUFFIX*. For example, the segment `CODE_I` holds code which executes in RAM and which was initialized by `CODE_ID`. Code for normal functions is placed in the `CODE` segment.

For each of the segment groups, the following segments are available:

Contents	Type	Suffix
Code	Read	No suffix
Code executing in RAM	Read/(Write)	I
Initializer for A_I	Read	ID

Table 7: Segment groups

To simplify this, the linker command file uses the XLINK option `-Q` to specify automatic setup for copy initialization of segments, also known as scatter loading. This will cause the linker to generate a new initializer segment into which it will place all data content of the code segment. Everything else, such as symbols and debugging information, will still be associated with the code segment. Code in the application must at runtime copy the contents of the initializer segment in ROM to the code segment in RAM. This is very similar to what compilers do for initialized variables.

```
// __ramfunc code copied to and executed from RAM
-Z (DATA) CODE_I=RAMSTART-RAMEND

-QCODE_I=CODE_ID
```

EXCEPTION VECTORS

The exception vectors are placed in the segment `INTVEC` which is normally located at address 0. The linker directive would then look like this:

```
-Z (CODE) INTVEC=00-3F
```

If the exception vectors contain a branch to the exception handlers, the exception handlers must be located within reach of the branch instruction. If instead a `load pc` instruction is located at the exception vector, there are no placement restrictions.

The above is also valid for the `__iar_swi_handler` function, a software interrupt handler that is provided with the product. This function is included in the C libraries and is located in the `SWITAB` segment. Its source code is available in the `arm\src\lib` directory, in case you need to modify this function.

C++ dynamic initialization

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

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

For example:

```
-Z (CODE) DIFUNCT=08000-FFFF
```

For additional information, see *DIFUNCT*, page 115.

Efficient usage of segments and memory

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

CONTROLLING DATA AND FUNCTION PLACEMENT

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

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

Note: Placing variables and functions into named segments can also be done using the `--segment` option. For details of this option, see *--segment*, page 143.

Data placement at an absolute location

To place a variable at an absolute address, the argument to the operator `@` and the `#pragma location` directive should be a literal number, representing the actual address. The absolute location must fulfil the alignment requirement for the variable that should be located.

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 @ 0x1011;    /* Error, misaligned. */

```

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

Data placement into named segments

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

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

Example

```

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

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

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

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

```

Function placement into named segments

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

Example

```

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

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

```

Declaring located variables extern

Using IAR extensions in C, read-only SFRs can be declared like this:

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

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

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

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

CREATING USER-DEFINED SEGMENTS

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

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

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

THE LINKED RESULT OF CODE AND DATA PLACEMENT

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

Segment too long errors and range errors

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

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

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

Linker map file

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

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



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

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

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

The DLIB runtime environment

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

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

The chapter then describes how to configure functionality like locale and file I/O, and how to get C-SPY runtime support.

Finally, the chapter describes how you prevent incompatible modules from being linked together.

Introduction to the runtime environment

The demands of an embedded application on the runtime environment depend both on the application itself and on the target hardware. The IAR DLIB runtime environment can be used as is together with the IAR C-SPY Debugger. However, to be able to run the application on hardware, you must adapt the runtime environment.

This section gives an overview of:

- The runtime environment and its components
- Library selection.

RUNTIME ENVIRONMENT FUNCTIONALITY

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

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

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

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

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

If you build your application project with the XLINK options **With runtime control modules** (`-r`) or **With I/O emulation modules** (`-rt`), certain functions in the library, such as functions for I/O and exit, will be replaced by functions that communicate with the IAR C-SPY Debugger. For further information, see *C-SPY Debugger runtime interface*, page 60.

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

LIBRARY SELECTION

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

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

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

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

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

SITUATIONS THAT REQUIRE LIBRARY BUILDING

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

You must build your own library when:

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

For information about how to build a customized library, see *Building and using a customized library*, page 47.

LIBRARY CONFIGURATIONS

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

The following DLIB library configurations are available:

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

Table 8: Library configurations

In addition to these configurations, you can define your own configurations, which means that you must modify the configuration file. Note that the library configuration file describes how a library was built and thus cannot be changed unless you rebuild the library. For further information, see *Building and using a customized library*, page 47.

The prebuilt libraries are based on the default configurations, see Table 9, *Prebuilt libraries*, page 40. There is also a ready-made library project template that you can use if you want to rebuild the runtime library.

Using a prebuilt library

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

- Architecture
- CPU mode
- Interworking
- VFP
- Byte order
- Stack alignment—all prebuilt libraries are built using stack alignment 8, which is also compatible with stack alignment 4
- Library configuration—Normal or Full.

For the ARM IAR C/C++ Compiler, this means there is a prebuilt runtime library for each combination of these options. The following table shows the mapping of the library file, architecture, CPU mode, interworking, VFP, byte order, and library configurations:

Library	Architecture	CPU mode	Interworking	VFP	Byte order	Library configuration
d14tpainb8f.r79	v4T	ARM	Yes	No	Big	Full
d14tpainb8n.r79	v4T	ARM	Yes	No	Big	Normal
d14tpainl8f.r79	v4T	ARM	Yes	No	Little	Full
d14tpainl8n.r79	v4T	ARM	Yes	No	Little	Normal
d14tpannb8f.r79	v4T	ARM	No	No	Big	Full
d14tpannb8n.r79	v4T	ARM	No	No	Big	Normal
d14tpannl8f.r79	v4T	ARM	No	No	Little	Full
d14tpannl8n.r79	v4T	ARM	No	No	Little	Normal
d14tptinb8f.r79	v4T	Thumb	Yes	No	Big	Full
d14tptinb8n.r79	v4T	Thumb	Yes	No	Big	Normal
d14tptinl8f.r79	v4T	Thumb	Yes	No	Little	Full
d14tptinl8n.r79	v4T	Thumb	Yes	No	Little	Normal
d14tptnnb8f.r79	v4T	Thumb	No	No	Big	Full
d14tptnnb8n.r79	v4T	Thumb	No	No	Big	Normal
d14tptnnl8f.r79	v4T	Thumb	No	No	Little	Full
d14tptnnl8n.r79	v4T	Thumb	No	No	Little	Normal
d15tpainb8f.r79	v5T	ARM	Yes	No	Big	Full
d15tpainb8n.r79	v5T	ARM	Yes	No	Big	Normal

Table 9: Prebuilt libraries

Library	Architecture	CPU mode	Interworking	VFP	Byte order	Library configuration
dl5tpainl8f.r79	v5T	ARM	Yes	No	Little	Full
dl5tpainl8n.r79	v5T	ARM	Yes	No	Little	Normal
dl5tpaivb8f.r79	v5T	ARM	Yes	Yes	Big	Full
dl5tpaivb8n.r79	v5T	ARM	Yes	Yes	Big	Normal
dl5tpaivl8f.r79	v5T	ARM	Yes	Yes	Little	Full
dl5tpaivl8n.r79	v5T	ARM	Yes	Yes	Little	Normal
dl5tpannb8f.r79	v5T	ARM	No	No	Big	Full
dl5tpannb8n.r79	v5T	ARM	No	No	Big	Normal
dl5tpannl8f.r79	v5T	ARM	No	No	Little	Full
dl5tpannl8n.r79	v5T	ARM	No	No	Little	Normal
dl5tpanvb8f.r79	v5T	ARM	No	Yes	Big	Full
dl5tpanvb8n.r79	v5T	ARM	No	Yes	Big	Normal
dl5tpanvl8f.r79	v5T	ARM	No	Yes	Little	Full
dl5tpanvl8n.r79	v5T	ARM	No	Yes	Little	Normal
dl5tptinb8f.r79	v5T	Thumb	Yes	No	Big	Full
dl5tptinb8n.r79	v5T	Thumb	Yes	No	Big	Normal
dl5tptinl8f.r79	v5T	Thumb	Yes	No	Little	Full
dl5tptinl8n.r79	v5T	Thumb	Yes	No	Little	Normal
dl5tptnnb8f.r79	v5T	Thumb	No	No	Big	Full
dl5tptnnb8n.r79	v5T	Thumb	No	No	Big	Normal
dl5tptnnl8f.r79	v5T	Thumb	No	No	Little	Full
dl5tptnnl8n.r79	v5T	Thumb	No	No	Little	Normal

Table 9: Prebuilt libraries (Continued)

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

```
<type><architecture>p<cpu_mode><interworking><fpv><endian><stack_
align><library_configuration>.r79
```

where

- `<type>` is `dl` for the IAR DLIB Library
- `<architecture>` is the name of the architecture. It can be one of `4t` or `5t`, for ARM architecture `v4T` or `v5T`, respectively
- `<cpu_mode>` is one of `t`, or `a`, for Thumb and ARM, respectively
- `<interworking>` is `i` if the library was compiled with the interworking option, otherwise it is `n`

- `<fpu>` is `v` if the library was compiled with the VFPv1 floating-point support, otherwise it is `n` for software floating-point support
- `<endian>` is one of `l`, or `b`, for little-endian and big-endian, respectively
- `<stack_align>` reflects the stack alignment used in the library
- `<library_configuraton>` is one of `n` or `f` for normal and full, respectively.

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



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



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

- Specify which library object file to use on the XLINK command line, for instance:
`dl4tpainl8n.r79`
- Specify the include paths for the compiler and assembler:
`-I arm\inc`
- Specify the library configuration file for the compiler:
`-D_DLIB_CONFIG_FILE=C:\...\dl4tpainl8n.h`

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

CUSTOMIZING A PREBUILT LIBRARY WITHOUT REBUILDING

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

- Selecting formatters used by `printf` and `scanf` by setting the appropriate options
- Overriding library modules with your own customized versions.

The following items can be customized:

Items that can be customized	Described in
Formatters for <code>printf</code> and <code>scanf</code>	page 43
Startup and termination code	page 49
Low-level input and output	page 52
File input and output	page 54
Low-level environment functions	page 57
Low-level signal functions	page 58
Low-level time functions	page 59
Size of heaps, stacks, and segments	page 29

Table 10: Customizable items

For a description about how to override library modules, see *Overriding library modules*, page 45.

Modifying a library by setting library options

Setting library options is the easiest way to modify the runtime environment. There are options for:

- Overriding default formatters for the `printf` and `scanf` functions
- Setting the sizes of the heap and stack, see page 33.

To override the default formatter for all the `printf` related functions, except for `wprintf` variants, and all the `scanf`-related functions, except for `wscanf` variants, you simply set the appropriate library options. This section describes the different options available.

Note: If you rebuild the library, it is possible to optimize these functions even further, see *Configuration symbols for `printf` and `scanf`*, page 53.

CHOOSING PRINTF FORMATTER

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

The following table summarizes the capabilities of the different formatters:

Formatting capabilities	<code>_PrintfFull</code> (default)	<code>_PrintfLarge</code>	<code>_PrintfSmall</code>	<code>_PrintfTiny</code>
Multibyte support	*	*	*	No
Conversion specifiers <code>a</code> , and <code>A</code>	Yes	No	No	No
Conversion specifiers <code>e</code> , <code>E</code> , <code>f</code> , <code>F</code> , <code>g</code> , and <code>G</code>	Yes	Yes	No	No
Conversion specifier <code>n</code>	Yes	Yes	No	No
Format flag <code>space</code> , <code>+</code> , <code>-</code> , <code>#</code> , and <code>0</code>	Yes	Yes	Yes	No
Length modifiers <code>h</code> , <code>l</code> , <code>L</code> , <code>s</code> , <code>t</code> , and <code>Z</code>	Yes	Yes	Yes	No
Field width and precision, including <code>*</code>	Yes	Yes	Yes	No
<code>long long</code> support	Yes	Yes	No	No
Approximate relative size	100%	85%	20%	10%

Table 11: Formatters for `printf`

* Depends on which library configuration is used.

For information about how to fine-tune the formatting capabilities even further, see *Configuration symbols for `printf` and `scanf`*, page 53.



Specifying the print formatter in the IAR Embedded Workbench

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



Specifying `printf` formatter from the command line

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

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

CHOOSING SCANF FORMATTER

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

The following table summarizes the capabilities of the different formatters:

Formatting capabilities	<code>_ScanfFull</code> (default)	<code>_ScanfLarge</code>	<code>_ScanfSmall</code>
Multibyte support	*	*	*
Floating-point support	Yes	No	No
Conversion specifier <code>n</code>	Yes	No	No
Scan set <code>[</code> and <code>]</code>	Yes	Yes	No
Assignment suppressing <code>*</code>	Yes	Yes	No
<code>long long</code> support	Yes	No	No
Approximate relative size	100%	35%	30%

Table 12: Formatters for `scanf`

* Depends on which library configuration is used.

For information about how to fine-tune the formatting capabilities even further, see *Configuration symbols for `printf` and `scanf`*, page 53.



Specifying `scanf` formatter in the IAR Embedded Workbench

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



Specifying `scanf` formatter from the command line

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

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

Overriding library modules

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

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



Overriding library modules using the IAR Embedded Workbench

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

- 1 Copy the appropriate *library_module.c* file to your project directory.
- 2 Make the required additions to the file (or create your own routine, using the default file as a model), and make sure to save it under the same name.
- 3 Add the customized file to your project.
- 4 Rebuild your project.



Overriding library modules from the command line

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

- 1 Copy the appropriate *library_module.c* to your project directory.
- 2 Make the required additions to the file (or create your own routine, using the default file as a model), and make sure to save it under the same name.
- 3 Compile the modified file using the same options as for the rest of the project:

```
iccarm library_module
```

This creates a replacement object module file named *library_module.r79*.

- 4 Add *library_module.r79* to the XLINK command line, either directly or by using an extended linker command file, for example:

```
xlink library_module dl4tpainl8n.r79
```

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

Run XLINK to rebuild your application.

This will use your version of *library_module.r79*, instead of the one in the library. For information about the XLINK options, see the *IAR Linker and Library Tools Reference Guide*.

Building and using a customized library

In some situations, see *Situations that require library building*, page 39, it is necessary to rebuild the library. In those cases you need to:

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

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

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

SETTING UP A LIBRARY PROJECT

The IAR Embedded Workbench provides a library project template which can be used for customizing the runtime environment configuration. This library template has full library configuration, see Table 8, *Library configurations*, page 39.



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

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

MODIFYING THE LIBRARY FUNCTIONALITY

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

The library functionality is determined by a set of *configuration symbols*. The default values of these symbols are defined in the `Dlib_defaults.h` file. This read-only file describes the configuration possibilities. In addition, your library has its own library configuration file `dlArmCustom.h`, which sets up that specific library with full library configuration. For more information, see Table 10, *Customizable items*, page 42.

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

Modifying the library configuration file

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

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

USING A CUSTOMIZED LIBRARY

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



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

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

System startup and termination

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

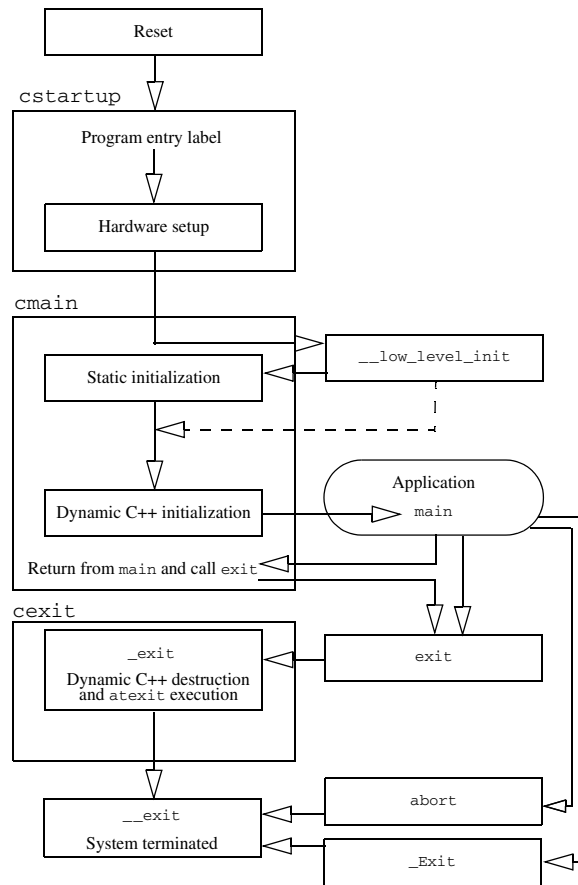


Figure 1: Startup and exit sequences

The code for handling startup and termination is located in the source files `cstartup.s79`, `cmain.s79`, `cexit.s79`, and `low_level_init.c` or `low_level_init.s79`, located in the `arm\src\lib` directory.

SYSTEM STARTUP

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

- When the cpu is reset it will jump to the program entry label in the `cstartup` module.
- Exception stack pointers are initialized to the end of each corresponding segment
- The stack pointer is initialized to the end of the `CSTACK` segment
- The preferred mode is set to ARM or Thumb
- The function `__low_level_init` is called, giving the application a chance to perform early initializations
- Static variables are initialized; this includes clearing zero-initialized memory and copying the ROM image of the RAM memory of the rest of the initialized variables depending on the return value of `__low_level_init`
- Static C++ objects are constructed
- The `main` function is called, which starts the application.

SYSTEM TERMINATION

An application can terminate normally in two different ways:

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

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

The default `exit` function is written in 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 C++ destructors for static and global variables, and functions registered with the standard C function `atexit`
- Close all open files
- Call `__exit`
- When `__exit` is reached, stop the system.

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

C-SPY interface to system termination

If your project is linked with the XLINK options **With runtime control modules** or **With I/O emulation modules**, the normal `__exit` and `abort` functions are replaced with special ones. C-SPY will then recognize when those functions are called and can take appropriate actions to simulate program termination. For more information, see *C-SPY Debugger runtime interface*, page 60.

Customizing system initialization

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

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

The code for handling system startup is located in the source files `cstartup.s79` and `low_level_init.c`, located in the `arm\src` directory. If you intend to rebuild the library, the source files are available in the template library project, see *Building and using a customized library*, page 47.

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

__LOW_LEVEL_INIT

Two skeleton low-level initialization files are supplied with the product—a C source file, `low_level_init.c` and an alternative assembler source file, `low_level_init.s79`. The latter is part of the prebuilt runtime environment. The only limitation using the C source version is that static initialized variables cannot be used within the file, as variable initialization has not been performed at this point.

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

MODIFYING THE CSTARTUP FILE

As noted earlier, you should not modify the `cstartup.s79` file if a customized version of `__low_level_init` is enough for your needs. However, if you do need to modify the `cstartup.s79` file, we recommend that you follow the general procedure for creating a modified copy of the file and adding it to your project, see *Overriding library modules*, page 45.

Standard streams for input and output

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

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

IMPLEMENTING LOW-LEVEL CHARACTER INPUT AND OUTPUT

To implement low-level functionality of the `stdin` and `stdout` streams, you must write the functions `__read` and `__write`, respectively. You can find template source code for these functions in the `arm/src` directory.

If you intend to rebuild the library, the source files are available in the template library project, see *Building and using a customized library*, page 47. Note that customizing the low-level routines for input and output does not require you to rebuild the library.

Note: If you write your own variants of `__read` or `__write`, special considerations for the C-SPY runtime interface are needed, see *C-SPY Debugger runtime interface*, page 60.

Example of using `__write` and `__read`

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

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

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

For information about the @operator, see *Controlling data and function placement*, page 33.

Configuration symbols for printf and scanf

When you set up your application project, you typically need to consider what `printf` and `scanf` formatting capabilities your application requires, see *Modifying a library by setting library options*, page 43.

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

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

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

Printf configuration symbols	Includes support for
<code>_DLIB_PRINTF_MULTIBYTE</code>	Multibytes
<code>_DLIB_PRINTF_LONG_LONG</code>	Long long (ll qualifier)

Table 13: Descriptions of printf configuration symbols

Printf configuration symbols	Includes support for
_DLIB_PRINTF_SPECIFIER_FLOAT	Floats
_DLIB_PRINTF_SPECIFIER_A	Hexadecimal floats
_DLIB_PRINTF_SPECIFIER_N	Output count (%n)
_DLIB_PRINTF_QUALIFIERS	Qualifiers h, l, L, v, t, and z
_DLIB_PRINTF_FLAGS	Flags -, +, #, and 0
_DLIB_PRINTF_WIDTH_AND_PRECISION	Width and precision
_DLIB_PRINTF_CHAR_BY_CHAR	Output char by char or buffered

Table 13: Descriptions of printf configuration symbols (Continued)

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

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

Table 14: Descriptions of scanf configuration symbols

CUSTOMIZING FORMATTING CAPABILITIES

To customize the formatting capabilities, you need to set up a library project, see *Building and using a customized library*, page 47. Define the configuration symbols according to your application requirements.

File input and output

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

Note that file I/O capability in the library is only supported by libraries with full library configuration, see *Library configurations*, page 39. In other words, file I/O is supported when the configuration symbol `__DLIB_FILE_DESCRIPTOR` is enabled. If not enabled, functions taking a *FILE* * argument cannot be used.

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

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

Table 15: Low-level I/O files

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

Note: If you link your library with the XLINK option **With I/O emulation modules**, C-SPY variants of the low-level I/O functions will be linked for interaction with C-SPY. For more information, see *C-SPY Debugger runtime interface*, page 60.

Locale

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

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

The DLIB library can be used in two major modes:

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

LOCALE SUPPORT IN PREBUILT LIBRARIES

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

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

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

CUSTOMIZING THE LOCALE SUPPORT

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

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

Locale configuration symbols

The configuration symbol `_DLIB_FULL_LOCALE_SUPPORT`, which is defined in the library configuration file, determines whether a library has support for a locale interface or not. The locale configuration symbols `_LOCALE_USE_XX_YY` and `_ENCODING_USE_ZZ` define all the supported locales.

If you want to customize the locale support, you simply define the locale configuration symbols required by your application. For more information, see *Building and using a customized library*, page 47.

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

Building a library without support for locale interface

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

Building a library with support for locale interface

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

SWITCHING LOCALES AT RUNTIME

The standard library function `setlocale` is used for switching locales when the application is running.

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

`xx_YY`

or

`xx_YY.encoding`

`xx` specifies the language code, `YY` specifies a region qualifier, and `encoding` specifies the multibyte encoding that should be used.

The `xx_YY` part matches the `_LOCALE_USE_XX_YY` preprocessor symbols that can be specified in the library configuration file.

Example

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

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

Environment interaction

Your application can interact with the environment using the functions `getenv` and `system`.

Note: The `putenv` function is not required by the standard.

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

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

```
key=value\0
```

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

For example:

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

If you need a more sophisticated environment variable handling, you should implement your own `getenv`, and possibly `putenv` function. This does not require that you rebuild the library. You can find source templates in the files `getenv.c` and `environ.c` in the `arm\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 45.

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

If you decide to rebuild the library, you can find source templates in the library project template. For further information, see *Building and using a customized library*, page 47.

Note: If you link your application with the XLINK option **With I/O emulation modules**, the functions `getenv` and `system` will be replaced by C-SPY variants. For further information, see *C-SPY Debugger runtime interface*, page 60.

Signal and raise

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

This does not require that you rebuild the library. You can find source templates in the files `Signal.c` and `Raise.c` in the `arm\src` directory. For information about overriding default library modules, see *Overriding library modules*, page 45.

If you decide to rebuild the library, you can find source templates in the library project template. For further information, see *Building and using a customized library*, page 47.

Time

To make the `time` and `date` functions work, you must implement the three functions `clock`, `time`, and `__getzone`.

This does not require that you rebuild the library. You can find source templates in the files `Clock.c` and `Time.c`, and `Getzone.c` in the `arm\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 45.

If you decide to rebuild the library, you can find source templates in the library project template. For further information, see *Building and using a customized library*, page 47.

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

Note: If you link your application with the XLINK option **With I/O emulation modules**, the functions `clock` and `time` will be replaced by C-SPY variants that return the host clock and time respectively. For further information, see *C-SPY Debugger runtime interface*, page 60.

Strtod

The function `strtod` does not accept hexadecimal floating-point strings in libraries with the normal library configuration. To make a library do so, you need to rebuild the library, see *Building and using a customized library*, page 47. Enable the configuration symbol `_DLIB_STRTOD_HEX_FLOAT` in the library configuration file.

Assert

If you have linked your application with the XLINK options **With runtime control modules** or **With I/O emulation modules**, C-SPY will be notified about failed asserts. If this is not the behavior you require, you must add the source file `xReportAssert.c` to your application project. Alternatively, you can rebuild the library. The `__ReportAssert` function generates the assert notification. You can find template code in the `arm\src\lib` directory. For further information, see *Building and using a customized library*, page 47.

C-SPY Debugger runtime interface

To include support for the C-SPY debugger runtime interface, you must link your application with the XLINK options **With runtime control modules** or **With I/O emulation modules**. In this case, C-SPY variants of the following library functions will be linked to the application:

Function	Description
abort	C-SPY notifies that the application has called abort *
__exit	C-SPY notifies that the end of the application has been reached *
__read	stdin, stdout, and stderr will be directed to the Terminal I/O window; all other files will read the associated host file
__write	stdin, stdout, and stderr will be directed to the Terminal I/O window, all other files will write to the associated host file
__open	Opens a file on the host computer
__close	Closes the associated host file on the host computer
__seek	Seeks in the associated host file on the host computer
remove	Writes a message to the Debug Log window and returns -1
rename	Writes a message to the Debug Log window and returns -1
time	Returns the time on the host computer
clock	Returns the clock on the host computer
getenv	Writes a message to the Debug Log window and returns 0
system	Writes a message to the Debug Log window and returns -1
_ReportAssert	Handles failed asserts *

Table 16: Functions with special meanings when linked with debug info

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

LOW-LEVEL DEBUGGER RUNTIME INTERFACE

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

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

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

THE DEBUGGER TERMINAL I/O WINDOW

When the functions `__read` or `__write` are called to perform I/O operations on the streams `stdin`, `stdout`, or `stderr`, data will be sent to or read from the C-SPY Terminal I/O window.

To make the Terminal I/O window available, the application must be linked using the XLINK option **With I/O emulation modules**. See the *ARM® IAR Embedded Workbench™ IDE User Guide* for more information about the Terminal I/O window.

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

Checking module consistency

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

When developing an application, it is important to ensure that incompatible modules are not used together. For example, in the ARM IAR C/C++ Compiler, it is possible to specify the byte order. If you write a routine that only works in little endian mode, it is possible to check that the routine is not used in an application built using big endian.

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

RUNTIME MODEL ATTRIBUTES

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

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

Example

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

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

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

Table 17: Example of runtime model attributes

USING RUNTIME MODEL ATTRIBUTES

Runtime model attributes can be specified in your C/C++ source code to ensure module consistency with other object files by using the `pragma rtmodel` directive. For detailed syntax information, see *#pragma rtmodel*, page 161.

Runtime model attributes can also be specified in your assembler source code by using the `RTMODEL` assembler directive. For detailed syntax information, see the *ARM® IAR Assembler Reference Guide*.

Example

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

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

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

PREDEFINED RUNTIME ATTRIBUTES

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

Runtime model attribute	Value	Description
__cpu_mode	__pcs__arm	Specifies the default cpu mode; see
	__pcs__thumb	--cpu_mode, page 126, for details
	__pcs__interwork	
__endian	little or big	Specifies the byte order
__rt_version	n	Specifies the compiler runtime model version This runtime key is always present in all modules generated by the ARM IAR C/C++ Compiler. If a major change in the runtime characteristics occurs, the value of this key changes
StackAlign4	USED	Specifies the stack alignment in bytes. Only one of the two attributes is set at a time.
StackAlign8		

Table 18: Predefined runtime model attributes

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

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

USER-DEFINED RUNTIME MODEL ATTRIBUTES

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

For instance, if you have a UART that can run in two modes, you can specify a runtime model attribute, for example uart. For each mode, specify a value, for example mode1 and mode2. You should declare this in each module that assumes that the UART is in a particular mode.

Implementation of cstartup

This section presents some general techniques used in the cstartup.s79 file, including background information that might be useful if you need to modify it. The cstartup.s79 file itself is well commented and is not described in detail in this guide.

Note: Do not modify the cstartup.s79 file unless required by your application. Your first option should always be to use a customized version of __low_level_init for initialization code.

For information about assembler source files, see the *ARM® IAR Assembler Reference Guide*.

MODULES AND SEGMENT PARTS

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

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

When XLINK builds an application, it starts with a small number of modules that have either been declared using the `__root` keyword or have the program entry label. 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.s79` file has been designed to use the mechanism described in the previous paragraph, 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 included in your application. Should no variables of a certain type exist, the code is simply discarded.

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

The segment parts defined in the `cstartup.s79` file are guaranteed to be placed immediately after each other. There are two reasons for this. First, the alignment requirement of the segment parts is every four bytes. The size of all arm-mode assembler instructions are four bytes. Second, XLINK will not change the order of the segment parts or modules.

This lets the `cstartup.s79` 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 code simply falls through to the next piece of code not discarded by the linker. The following example shows this technique:

```
MODULE    doSomething

RSEG     MYSEG:CODE:NOROOT(2)    // 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 MOV instruction below is included.

        RSEG    MYSEG:CODE:NOROOT(2)    // 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(2)    // Third segment part.
        PUBLIC  ?end_of_test

?end_of_test:
        MOV     PC,LR    // This is included if ?do_something above
                       // is included.

        ENDMOD

```

Added C functionality

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

The following include files are available:

<code>stdint.h</code>	Integer characteristics
<code>stdbool.h</code>	Bool type; the option Allow IAR extensions (-e) must be used

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

The following include files have added functionality:

- `math.h`
- `stdio.h`

- `stdlib.h`

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

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

<code>vscanf,</code> <code>vfscanf,</code> <code>vsscanf,</code> <code>vsnprintf</code>	Variants that have a <code>va_list</code> as argument.
<code>snprintf</code>	Same as <code>sprintf</code> , but writes to a size-limited array.
<code>__write_array</code>	Corresponds to <code>fwrite</code> on <code>stdout</code> .
<code>__ungetchar</code>	Corresponds to <code>ungetc</code> on <code>stdout</code> .
<code>__gets</code>	Corresponds to <code>fgets</code> on <code>stdin</code> .

In `stdlib.h`, the following functions have been added:

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

Assembler language interface

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

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

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

Mixing C and assembler

The ARM IAR C/C++ Compiler provides several ways for mixing C or C++ and assembler: inline assembler, modules written entirely in assembler, and the C alternative—intrinsic functions. It might be tempting to use simple inline assembler. However, you should carefully choose which method to use.

INTRINSIC FUNCTIONS

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

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

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

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

MIXING C AND ASSEMBLER MODULES

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

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

The first issue is discussed in the section *Calling assembler routines from C*, page 70. The following three are covered in the section *Calling convention*, page 73.

The answer to the final question is that the call stack can be displayed when you run assembler code in the debugger. However, the debugger requires information about the *call frame*, which must be supplied as annotations in the assembler source file. For more information, see *Call frame information*, page 81.

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

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

There will be some overhead in the form of a function call and return instruction sequence, and the compiler will regard some registers as scratch registers. In many cases, the overhead of the function call and return instruction sequence is compensated by the work of the optimizer.

The recommended method for mixing C or C++ and assembler modules is described in *Calling assembler routines from C*, page 70, and *Calling assembler routines from C++*, page 72, respectively.

INLINE ASSEMBLER

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

```
bool flag;

void foo()
{
    while (!flag)
    {
        asm("MOV flag,PIND");
    }
}
```

In this example, the assignment of `flag` is not noticed by the compiler, which means it is not recommended that the surrounding code relies on the inline assembler statement

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

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

- The compiler's various optimizations will disregard any effects of the inline sequences, which will not be optimized at all
- The directives `CODE16` and `CODE32` will cause errors; several other directives cannot be used at all
- Alignment cannot be controlled; this means, for example, that `DC32` directives may be misaligned
- Auto variables cannot be accessed
- Labels cannot be declared.

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

Calling assembler routines from C

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

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

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

or

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

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

CREATING SKELETON CODE

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

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

```
long gLong;
double gDouble;

long func(long arg1, double arg2)
{
    long locLong = arg1;
    gLong = arg1;
    gDouble = arg2;
    return locLong;
}

int main()
{
    long locLong = gLong;
    gLong = func(locLong, gDouble);
    return 0;
}
```


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

COMPILING THE CODE



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



Use the following options to compile the skeleton code:

```
iccarm skeleton -lA .
```

The `-lA` option creates an assembler language output file including C or C++ source lines as assembler comments. The `.` (period) specifies that the assembler file should be named in the same way as the C or C++ module (`skeleton`), but with the filename extension `s79`. Also remember to specify the code model, and optimization level you are using.

The result is the assembler source output file `skeleton.s79`.

Note: The `-lA` option creates a list file containing call frame information (CFI) directives, which can be useful if you intend to study these directives and how they are used. If you only want to study the calling convention, you can exclude the CFI directives from the list file by using the option `-lB` instead. Note that CFI information must be included in the source code to make the C-SPY Call Stack window work.

The output file

The output file contains the following important information:

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

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

Calling assembler routines from C++

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

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

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

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

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

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

```
class X;

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

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

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

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

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

Calling convention

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

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

This section describes the calling convention used by the ARM IAR C/C++ Compiler. The following issues are described:

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

Unless otherwise noted, the calling convention used by the ARM IAR C/C++ Compiler adheres to the Advanced RISC Machines Ltd ARM/Thumb Procedure Call Standard (ATPCS); see the ARM web site.

FUNCTION DECLARATIONS

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

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

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

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

C AND C++ LINKAGE

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

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

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

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

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

PRESERVED VERSUS SCRATCH REGISTERS

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

Scratch registers

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

Any of the registers R0 to R3, and R12, can be used as a scratch register by the function.

Preserved registers

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

The registers R4 through to R11 are preserved registers. They are preserved by the called function.

Special registers

Some registers have special handling that you must consider:

- The stack pointer register, R13/SP, must at all times point to or below the last element on the stack. In the eventuality of an interrupt, everything below the point the stack pointer points to, will be destroyed.
- The register R15/PC is dedicated for the Program Counter.
- The link register, R14/LR, holds the return address at the entrance of the function.

FUNCTION ENTRANCE

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

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 larger than 32 bits, the memory location where the structure is to be stored is passed as an extra parameter. Notice that it is always treated as the *first parameter*.
- If the function is a non-static C++ member function, then the `this` pointer is passed as the first parameter (but placed after the return structure pointer, if there is one).

Register parameters

The registers available for passing parameters are R0–R3:

Parameters	Passed in registers
Scalar and floating-point values no larger than 32 bits, and single-precision (32-bits) floating-point values	Passed using the first free register: R0–R3
Structure values larger than 32 bits	Passed using the first free register: R0–R3. Any remaining parameters of this type are passed on the stack.
long long and double-precision (64-bit) values	Passed in first available register pair: R0 : R1, R1 : R2, or R2 : R3, or divided between the last parameter register and the stack.

Table 19: Registers used for passing parameters

The assignment of registers to parameters is a straightforward process. Traversing the parameters from left to right, the first parameter is assigned to the available register or registers. Should there be no more available registers, the parameter is passed on the stack in reverse order.

When functions that have parameters smaller than 32 bits are called, the values are sign or zero extended to ensure that the unused bits have consistent values. Whether the values will be sign or zero extended depends on their type—signed or unsigned.

If floating-point operations are performed by a Vector Floating Point (VFP) coprocessor, and the compiler is used in non-interworking mode, the following changes are made to the parameter passing convention.

Parameters	Passed in registers
Single-precision floating-point values	Passed using the first free register: S0–S15
Double-precision floating-point values	Passed using the first free register: D0–D7

Table 20: VFP registers used for passing parameters in non-interworking mode

For information about how to enable support in the compiler for the VFP coprocessor, see *VFP and floating-point arithmetic*, page 7.

Stack layout

Stack parameters are stored in memory, starting at the location pointed to by the stack pointer. Below the stack pointer (towards low memory) there is free space that the called function can use. The first stack parameter is stored at the location pointed to by the stack pointer. The next one is stored at the next location on the stack that is divisible by four, etc. It is the responsibility of the caller to clean the stack after the called function has returned.

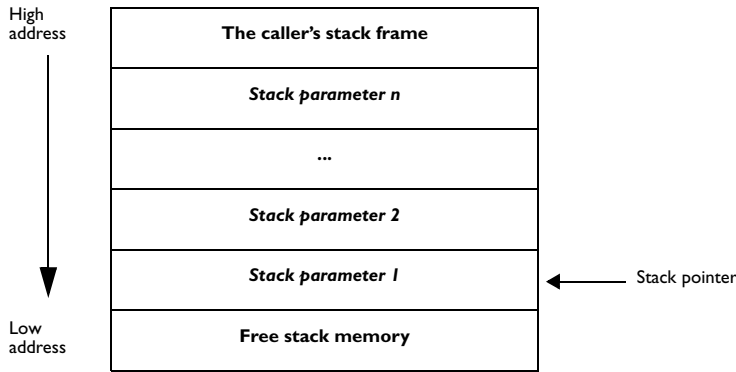


Figure 2: Storing stack parameters in memory

According to the Advanced RISC Machines Ltd Arm/Thumb Procedure Call Standard (ATPCS; see the ARM web site), the stack should be aligned to 8 at function entry. With the ARM IAR C/C++ Compiler, the stack is by default aligned to 4, but it is possible to align the stack to 8. For information about how to change the stack alignment, see *--stack_align*, page 144.

FUNCTION EXIT

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

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

Registers used for returning values

The registers available for returning values are R0 and R0 : R1.

Return values	Passed in register/register pair
Scalar and structure return values no larger than 32 bits, and single-precision (32-bit) floating-point return values	R0
The memory address of a structure return value larger than 32 bits	R0
long long and double-precision (64-bit) return values	R0 : R1

Table 21: Registers used for returning values

If the returned value is smaller than 32 bits, the value is sign or zero extended to 32 bits.

If floating-point operations are performed by a Vector Floating Point (VFP) coprocessor, and the compiler is used in non-interworking mode, the following changes are made to the parameter passing convention.

Parameters	Passed in register/register pair
Single-precision floating-point values	S0
Double-precision floating-point values	D0

Table 22: VFP registers used for returning values in non-interworking mode

For information about how to enable support in the compiler for the VFP coprocessor, see *VFP and floating-point arithmetic*, page 7.

Stack layout

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

RETURN ADDRESS HANDLING

A function written in assembler language should, when finished, return to the caller by jumping to the address pointed to by the register LR.

At function entry, non-scratch registers and the `LR` register can be pushed with one instruction. At function exit, all these registers can be popped with one instruction. The return address can be popped directly to `PC`.

The following example shows what this can look like in ARM mode:

```
STMDB      SP!, {R4-R9, LR}      // function entry
.
.
.
LDMIA      SP!, {R4-R9, PC}      // function exit
```

In Thumb mode, the `PUSH` and `POP` instructions will be used instead of `STMDB` and `LDMIA`.

EXAMPLES

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

Example 1

Assume that we have the following function declaration:

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

This function takes one parameter in the register `R0`, and the return value is passed back to its caller in the register `R0`.

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    R0, R0, #+0x1
    MOV    PC, LR
```

Example 2

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

```
struct a_struct { int a,b,c,d,e; };
int a_function(struct a_struct x, int y);
```

The values of the structure members `a`, `b`, `c`, and `d` are passed in registers `R0-R3`. The last structure member `e` and the integer parameter `y` are passed on the stack. The calling function must reserve eight bytes on the top of the stack and copy the contents of the two stack parameters to that location. The return value is passed back to its caller in the register `R0`.

Example 3

The function below will return a `struct`.

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

It is the responsibility of the calling function to allocate a memory location for the return value and pass a pointer to it as a hidden first parameter. The pointer to the location where the return value should be stored is passed in `R0`. The caller assumes that `R0` remains untouched. The parameter `x` is passed in `R1`.

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

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

In this case, the return value is a scalar, so there is no hidden parameter. The parameter `x` is passed in `R0`, and the return value is returned in `R0`.

Calling functions

There are two fundamentally different ways to call functions—directly or via a function pointer. This section presents the assembler instructions that can be used for calling and returning from functions.

The compiler automatically generates the most efficient assembler instructions to call functions. The normal function calling instruction is the branch-and-link instruction:

```
BL label
```

The location that the called function should return to (that is, the location immediately after this instruction) is stored in the link register, `LR`.

With the `BL` instruction, the destination label may not be further away than 32 Mbytes in ARM mode and 4 Mbytes in Thumb mode. Longer jumps can be made with the `BX` instruction. This instruction also changes the CPU mode between Thumb and ARM states, if this is required by the called function.

```
BX reg
```

For some function calls, it is uncertain whether the branch-and-link instruction can reach the destination address, for example if the calling and called functions are declared in different modules. In such situations, or if the CPU mode changes between Thumb and ARM, the compiler generates a `_BLF` assembler pseudo instruction. This instruction has two arguments, where the first is a label to the called function, and the second is a label to a module local relay function. This relay function will be used when the called function cannot be reached directly by a simple branch-and-link instruction.

The `_BLF` pseudo instruction will expand to an ordinary `BL function` instruction if this is sufficient—otherwise a `BL function_relay` will be generated.

During segment placement, XLINK will minimize the number of relay functions by letting references in a module use relay functions in adjacent modules whenever possible. Any unused relay functions are simply discarded.

The following example shows a simple call to the function `func` in ARM mode:

```
_BLF    func, func??rA
```

The call will also generate a relay function `func??rA` that may be used for calling the function `func`.

```
CODE32
func??rA
    LDR    R12, ??RelayData_0
    MOV    PC, R12

DATA
??RelayData_0:
    DC32    func
```

Here the address to the function `func` is loaded into the register `R12` which is then used for calling the function `func`. The relay function loads somewhat differently in Thumb mode but the functionality is the same.

When a function should return control to the caller, the `MOV PC, LR` instruction will be used if no registers have been pushed at function entry and the `STMDB SP!, {Reg-Reg, LR}` instruction (in ARM mode) if registers have been saved at function entry.

When a function call is made via a function pointer in non-interworking ARM mode, the following code will be generated:

```
LDR    Reg1, function_pointer ; Location of function pointer
LDR    Reg2, [Reg1, #0]       ; Load function address
MOV    LR, PC                 ; Save return address
MOV    PC, Reg2               ; Make function call
```

The address is stored in a register and is then used for calling the function. Calls via a function pointer reach the whole 32-bit address space.

Call frame information

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

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

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

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

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

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

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

Resource	Description
CFA R13	The call frames of the stack
R0–R12	Processor general-purpose 32-bit registers
R13	Stack pointer, SP
R14	Link register, LR
S0–S31	Vector Floating Point (VFP) 32-bit coprocessor registers
CPSR	Current program status register
SPSR	Saved program status register
?RET	The return address

Table 23: Call frame information resources defined in a names block

See the *ARM® IAR Assembler Reference Guide* for more information.

Using C++

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

Overview

STANDARD EMBEDDED C++

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

The following C++ features are supported:

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

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

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

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

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

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

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

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

EXTENDED EMBEDDED C++

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

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

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

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

ENABLING C++ SUPPORT



In the ARM IAR C/C++ Compiler, the default language is C. To be able to compile files written in Embedded C++, you must use the `--ec++` compiler option. See `--ec++`, page 131.

To take advantage of *Extended* Embedded C++ features in your source code, you must use the `--eec++` compiler option. See `--eec++`, page 131.



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

Feature descriptions

CLASSES

A class type, `class`, and `struct` in C++ can have static data and function members, and dynamic data and function members. The same rules apply to the static members as for statically linked symbols outside of a class, in other words, they can have any IAR type, memory, or object attributes. *Dynamic* members cannot have any IAR attributes, however.

Example

```
class A {
    public:
        static int i @ 60;          // Located at address 60.
        static void f();
};
```

FUNCTIONS

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

Example

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

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

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

TEMPLATES

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

The standard template library

The STL (standard template library) delivered with the product is tailored for Extended EC++, as described in *Extended Embedded C++*, page 84.

STL and the IAR C-SPY Debugger

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

VARIANTS OF CASTS

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

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

MUTABLE

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

NAMESPACE

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

THE STD NAMESPACE

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

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


Efficient coding for embedded applications

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

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

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

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

Taking advantage of the compilation system

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

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

CONTROLLING COMPILER OPTIMIZATIONS

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

The following table describes the optimization levels:

Optimization level	Description
None	(Best debug support) Variables live through their entire scope
Low	Limited live-range for variables
Medium	Live-dead analysis and optimization Dead code elimination Redundant label elimination Redundant branch elimination Code hoisting Peephole optimization Some register content analysis and optimization Static clustering Common subexpression elimination
High	(Maximum optimization) Instruction scheduling Cross jumping Advanced register content analysis and optimization Loop unrolling Function inlining Code motion Type-based alias analysis

Table 24: Compiler optimization levels

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

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

Both compiler options and pragma directives are available for specifying the preferred type and level of optimization. The chapter *Compiler options* contains reference information about the command line options used for specifying optimization type and level. Refer to the *ARM® IAR Embedded Workbench™ IDE User Guide* for information about the compiler options available in the IAR Embedded Workbench. Refer to *#pragma optimize*, page 158, for information about the pragma directives that can be used for specifying optimization type and level.

Fine-tuning enabled transformations

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

- Common sub-expression elimination
- Loop unrolling
- Function inlining
- Code motion
- Type-based alias analysis
- Static clustering
- Instruction scheduling

Common subexpression elimination

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

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

Loop unrolling

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

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

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

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

Function inlining

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

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

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

Code motion

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

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

Type-based alias analysis

A C/C++ application that conforms to the ISO/ANSI standard accesses an object only by an lvalue that has one of the following types:

- a `const` or `volatile` qualified version of the declared type of the object
- a type that is the signed or unsigned type corresponding to a `const` or `volatile` qualified version of the declared type of the object
- an aggregate or union type that includes one of the aforementioned types among its members
- a character type.

By default, the compiler is free to assume that objects are only accessed through the declared type or through `unsigned char`. However, at optimization level **High** the type-based alias analysis optimization is used, which means that the optimizer will assume that the program is standards compliant and the rules above will be used for determining what objects may be affected when a pointer indirection is used in an assignment.

Consider the following example:

```
short s;
unsigned short us;
long l;
```

```

unsigned long ul;
float f;

unsigned short *usptra;
char *cptra;

struct A
{
    short s;
    float f;
} a;

void test(float *fptra, long *lptra)
{
    /* May affect: */
    *lptra = 0;      /* l, ul */
    *fptra = 1.0;    /* f, a */
    *usptra = 4711;  /* s, us, a */
    *cptra = 17;     /* s, us, l, ul, f, usptra, cptra, a */
}

```

Because an object should only be accessed as its declared type (or a qualified version of its declared type, or a signed/unsigned type corresponding to its declared type) it is also assumed that the object that `fptra` points to will not be affected by an assignment to the object that `lptra` points to.

This may cause unexpected behavior for some non-conforming programs. The following contrived example illustrates one of the benefits of type-based alias analysis and what can happen when a non-conforming program breaks the rules above.

```

short f(short *sptra, long *lptra)
{
    short x = *sptra;
    *lptra = 0;
    return *sptra + x;
}

```

Because the `*lptra = 0` assignment cannot affect the object that `sptra` points to, the optimizer will assume that `*sptra` in the return statement has the same value as variable `x` was assigned at the beginning of the function. Hence, it is possible to eliminate a memory access by returning `x << 1` instead of `*sptra + x`.

```

short fail()
{
    union
    {
        short s[2];
        long l;
    } u;
}

```

```

    u.s[0] = 4711;

    return f(&u.s[0], &u.d);
}

```

When the function fails to pass the address of the same object as both a pointer to `short` and as a pointer to `long` for the function `f`, the result will most likely not be what was expected.

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

Static clustering

When static clustering is enabled, static and global variables are arranged so that variables that are accessed in the same function are stored close to each other. This makes it possible for the compiler to use the same base pointer for several accesses. Alignment gaps between variables can also be eliminated.

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

Instruction scheduling

The ARM IAR C/C++ Compiler features an instruction scheduler to increase the performance of the generated code. To achieve that goal, the scheduler rearranges the instructions to minimize the number of pipeline stalls emanating from resource conflicts within the microprocessor.

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

Selecting data types and placing data in memory

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

USING EFFICIENT DATA TYPES

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

- Use `int` or `long` instead of `char` or `short` whenever possible, to avoid sign extension or zero extension. In particular, loop indexes should always be `int` or `long` to minimize code generation. Also, in Thumb mode, accesses through the stack pointer (`SP`) is restricted to 32-bit data types, which further emphasizes the benefits of using one of these data types.
- Use unsigned data types, unless your application really requires signed values.
- Try to avoid 64-bit data types, such as `double` and `long long`.

- Bitfields and packed structures generate large and slow code and should be avoided in time-critical applications.
- Using floating-point types on a microprocessor without a math coprocessor is very inefficient, both in terms of code size and execution speed.
- Declaring a pointer to `const` data tells the calling function that the data pointed to will not change, which opens for better optimizations.

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

Floating-point types

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

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

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

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

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

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

REARRANGING ELEMENTS IN A STRUCTURE

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

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

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

For information about alignment requirements, see *Alignment*, page 103.

There are two ways to solve the problem:

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

For further details about the `#pragma pack` directive, see *#pragma pack*, page 159.

ANONYMOUS STRUCTS AND UNIONS

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

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



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



Use the `-e` compiler option to enable language extensions. See *-e*, page 131, for additional information.

Example

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

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



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

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

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

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

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

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

Writing efficient code

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

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

- The use of local variables is preferred over static or global variables. The reason is that the optimizer must assume, for example, that called functions may modify non-local variables.

- Avoid taking the address of local variables using the `&` operator. There are two main reasons why this is inefficient. First, the variable must be placed in memory, and thus cannot be placed in a processor register. This results in larger and slower code. Second, the optimizer can no longer assume that the local variable is unaffected over function calls.
- Module-local variables—variables that are declared static—are preferred over global variables. Also avoid taking the address of frequently accessed static variables.
- The compiler is capable of inlining functions. This means that instead of calling a function, the compiler inserts the content of the function at the location where the function was called. The result is a faster, but often larger, application. Also, inlining may enable further optimizations. The compiler often inlines small functions declared static. The use of the `#pragma inline` directive and the C++ keyword `inline` gives the application developer fine-grained control, and it is the preferred method compared to the traditional way of using preprocessor macros. Too much inlining can decrease performance due to the limited number of ARM registers. This feature can be disabled using the `--no_inline` command line option; see *--no_inline*, page 138.
- Avoid using inline assembler. Instead, try writing the code in C or C++, use intrinsic functions, or write a separate module in assembler language. For more details, see *Mixing C and assembler*, page 67.

SAVING STACK SPACE AND RAM MEMORY

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

- If stack space is limited, avoid long call chains and recursive functions.
- Declare variables with a short life span as auto variables. When the life spans for these variables end, the previously occupied memory can then 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, such as structures, to functions; in order to save stack space, you should instead pass them as pointers or, in EC++, as references.

FUNCTION PROTOTYPES

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

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

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

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

Prototyped style

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

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

Kernighan & Ritchie style

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

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

INTEGER TYPES AND BIT NEGATION

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

In some cases there may be warnings (for example, constant conditional or pointless comparison), in others just a different result than what is expected. Under certain circumstances the compiler may warn only at higher optimizations, for example, if the compiler relies on optimizations to identify some instances of constant conditionals.

In the following example an 8-bit character, a 32-bit integer, and two's complement is assumed:

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

Here, the test is always false. On the right hand side, `0x80` is `0x00000080`, and `~0x00000080` becomes `0xFFFFF7F`. On the left hand side, `c1` is an 8-bit unsigned character, and, thus, cannot be larger than 255. It also cannot be negative, thus the integral promoted value can never have the top 24 bits set.

PROTECTING SIMULTANEOUSLY ACCESSED VARIABLES

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

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

A sequence that access a variable must also not be interrupted, this can be done using the `__monitor` keyword in interruptible code. This must be done for both write *and* read sequences, otherwise you might end up reading a partially updated variable.

This is true, for all variables of all sizes. Accessing a byte-sized variable can be an atomic operation, but this is not guaranteed and you should not rely on it unless you continuously study the compiler output. It is safer to ensure that the sequence is an atomic operation using the `__monitor` keyword.



Protecting the EEPROM write mechanism

A typical example of when it can be necessary to use the `__monitor` keyword is when protecting the EEPROM write mechanism, which can be used from two threads (for example, main code and interrupts). Servicing an interrupt during an EEPROM write sequence can in many cases corrupt the written data.

ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for a number of ARM derivatives are included in the ARM IAR C/C++ Compiler delivery. The header files are named `iochip.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

SFRs with bitfields are declared in the header file. The following example is from `ioks32c5000a.h`:

```
/* system configuration register */
typedef struct {
    __REG32 se      :1; /* stall enable, must be 0 */
    __REG32 ce      :1; /* cache enable */
    __REG32 we      :1;
    __REG32 cm      :2; /* cache mode */
    __REG32 isbp    :10; /* internal SRAM base pointer */
    __REG32 srbbp   :10; /* special register bank base pointer */
    __REG32 ce      :6; /* cache enable */
} __syscfg_bits;

__IO_REG32_BIT(__SYSCFG, 0x03FF0000, __READ_WRITE, __syscfg_bits);
```

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

```
// whole register access
__SYSCFG = 0x12345678;

// Bitfield accesses
__SYSCFG_bit.we = 1;
__SYSCFG_bit.cm = 3;
```

You can also use the header files as templates when you create new header files for other ARM derivatives.

NON-INITIALIZED VARIABLES

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

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

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

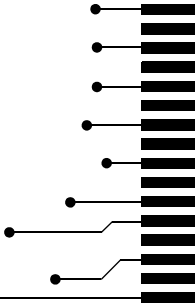
Variables declared using the `__no_init` keyword could, for example, be large input buffers or mapped to special RAM that keeps its content even when the application is turned off.

For information about the `__no_init` keyword, see page 150. Note that to use this keyword, language extensions must be enabled; see `-e`, page 131. For information about the `#pragma object_attribute`, see page 157.

Part 2. Compiler reference

This part of the ARM® IAR C/C++ 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 ARM IAR C/C++ Compiler.

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

Alignment

The alignment of a data object controls how it can be stored in memory. The reason for using alignment is that the ARM core can access 4-byte objects using one assembler instruction only when the object is stored at an address dividable by 4.

Objects with alignment 4 must be stored at an address dividable by 4, while objects with alignment 2 must be stored at addresses dividable by 2.

The ARM IAR C/C++ Compiler ensures this by assigning an alignment to every data type, ensuring that the ARM core will be able to read the data.

Byte order

The ARM core stores data in either little-endian or big-endian byte order. To specify the byte order, use the `--endian` option; see *--endian*, page 132.

In the little-endian byte order, which is default, the *least* significant byte is stored at the lowest address in memory. The *most* significant byte is stored at the highest address.

In the big-endian byte order, the *most* significant byte is stored at the lowest address in memory. The *least* significant byte is stored at the highest address.

Basic data types

The compiler supports all ISO/ANSI C basic data types.

INTEGER TYPES

The following table gives the size and range of each integer data type:

Data type	Size	Range	Alignment
char	8 bits	0 to 255	1
signed char	8 bits	-128 to 127	1
unsigned char	8 bits	0 to 255	1
short	16 bits	-32768 to 32767	2
signed short	16 bits	-32768 to 32767	2
unsigned short	16 bits	0 to 65535	2
int	32 bits	-2 ³¹ to 2 ³¹ -1	4
signed int	32 bits	-2 ³¹ to 2 ³¹ -1	4
unsigned int	32 bits	0 to 2 ³² -1	4
long	32 bits	-2 ³¹ to 2 ³¹ -1	4
signed long	32 bits	-2 ³¹ to 2 ³¹ -1	4
unsigned long	32 bits	0 to 2 ³² -1	4
long long	64 bits	-2 ⁶³ to 2 ⁶³ -1	4
signed long long	64 bits	-2 ⁶³ to 2 ⁶³ -1	4
unsigned long long	64 bits	0 to 2 ⁶⁴ -1	4

Table 25: Integer types

Signed variables are represented using the two’s complement form.

Bool

The `bool` data type is supported by default in the C++ language. If you have enabled language extensions, the `bool` type can also be used in C source code if you include the `stdbool.h` file. This will also enable the Boolean values `false` and `true`.

The enum type

ISO/ANSI C specifies that constants defined using the `enum` construction should be representable using the type `int`. The compiler will use the shortest signed or unsigned type required to contain the values.

When IAR Systems language extensions are enabled, and in C++, the `const` and `enum` types can also be of the type `long` or `unsigned long`.

The char type

The `char` type is by default unsigned in the compiler, but the `--char_is_signed` compiler option allows you to make it signed. Note, however, that the library is compiled with the `char` type as unsigned.

The wchar_t type

The `wchar_t` data type is an integer type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locals.

The `wchar_t` data type is supported by default in the C++ language. To use the `wchar_t` type also in C source code, you must include the `stddef.h` file from the runtime library.

Bitfields

In ISO/ANSI C, `int` and `unsigned int` can be used as the base type for integer bitfields. In the ARM IAR C/C++ Compiler, any integer type can be used as the base type when language extensions are enabled.

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

By default, the compiler places bitfield members from the least significant to the most significant bit in the container type.

By using the directive `#pragma bitfields=reversed`, the bitfield members are placed from the most significant to the least significant bit.

FLOATING-POINT TYPES

Floating-point values are represented by 32- and 64-bit numbers in standard IEEE format.

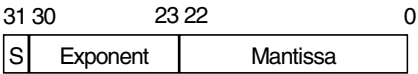
The ranges and sizes for the different floating-point types are:

Type	Size	Range (+/-)	Decimals	Exponent	Mantissa
float	32 bits	±1.18E-38 to ±3.39E+38	7	8 bits	23 bits
double	64 bits	±2.23E-308 to ±1.79E+308	15	11 bits	52 bits

Table 26: Floating-point types

32-bit floating-point format

The data type `float` is represented by the 32-bit floating-point format. The representation of a 32-bit floating-point number as an integer is:



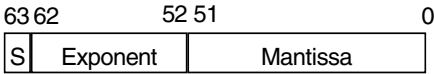
The value of the number is:

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

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

64-bit floating-point format

The data type `double` is represented by the 64-bit floating-point format. The representation of a 64-bit floating-point number as an integer is:



The value of the number is:

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

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

Special cases

The following applies to both 32-bit and 64-bit floating-point formats:

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

$(-1)^S * 2^{(1-BIAS)} * 0.Mantissa$

where `BIAS` is 127 and 1023 for 32-bit and 64-bit floating-point values, respectively.

Pointer types

The ARM IAR C/C++ Compiler has two basic types of pointers: code pointers and data pointers.

CODE POINTERS

The size of all code pointers is 32 bits and the range is `0x0-0xFFFFFFFF`.

When function pointer types are declared, attributes are inserted before the `*` sign, for example:

```
typedef void (__thumb __interwork * IntHandler) (void);
```

This can be rewritten using `#pragma` directives:

```
#pragma type_attribute=__thumb __interwork
typedef void IntHandler_function(void);
typedef IntHandler_function *IntHandler;
```

DATA POINTERS

There is one data pointer available. Its size is 32 bits and the range is `0x0-0xFFFFFFFF`.

CASTING

Casts between pointers have the following characteristics:

- Casting a *value* of an integer type to a pointer of a smaller type is performed by truncation
- Casting a *value* of an integer type to a pointer of a larger type is performed by zero extension
- Casting a *pointer type* to a smaller integer type is performed by truncation
- Casting a *pointer type* to a larger integer type is performed by zero extension
- Casting a *data pointer* to a function pointer and vice versa is illegal
- Casting a *function pointer* to an integer type gives an undefined result.

size_t

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

ptrdiff_t

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

intptr_t

`intptr_t` is a signed integer type large enough to contain a `void *`. In the ARM IAR C/C++ Compiler, the size of `intptr_t` is 32 bits. `intptr_t` is defined in the system include file `stdint.h`.

uintptr_t

`uintptr_t` is equivalent to `intptr_t`, with the exception that it is unsigned. `uintptr_t` is defined in the system include file `stdint.h`.

Structure types

The members of a `struct` are stored sequentially in the order in which they are declared: the first member has the lowest memory address.

ALIGNMENT

The `struct` and `union` types inherit the alignment requirements of their members. In addition, the size of a `struct` is adjusted to allow arrays of aligned structure objects.

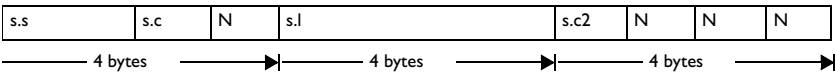
GENERAL LAYOUT

Members of a `struct` (fields) are always allocated in the order given in the declaration. The members are placed in memory according to the given alignment (offsets).

Example

```
struct {
    short s; /* stored in byte 0 and 1 */
    char c; /* stored in byte 2 */
           /* pad in byte 3 */
    long l; /* stored in byte 4, 5, 6, and 7 */
    char c2; /* stored in byte 8 */
           /* pad in byte 9, 10 and 11 */
} s;
```

The following diagram shows the layout in memory:

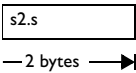


where N are pad bytes. The alignment of the struct is 4 bytes and its size is 12 bytes.

The structure itself has the same alignment as the field with the highest alignment requirement. For example:

```
struct {
    short s;
} s2;
```

The struct s2 is aligned on an address that can be divided by 2:



PACKED STRUCTURE TYPES

The #pragma pack directive is used for changing the alignment requirements of the members of a structure. This will change the way the layout of the structure is performed. The members will be placed in the same order as when declared, but there might be less pad space between members.

Example

```
#pragma pack(1)
struct {
    short s;
    char c;
    long l;
    char c2;
} s;
```

will be placed:



For more information, see *Rearranging elements in a structure*, page 93.

Type and object attributes

The ARM IAR C/C++ Compiler provides a set of attributes that support specific features of the ARM core. There are two types of attributes—*type attributes* and *object attributes*.

Type attributes affect the *external functionality* of the data object or function. For instance, how an object is placed in memory, or in other words, how it is accessed.

Object attributes affect the *internal functionality* of the data object or function.

To understand the syntax rules for the different attributes, it is important to be familiar with the concepts of the type attributes and the object attributes.

For information about how to use attributes to modify data, see the chapter *Data storage*. For information about how to use attributes to modify functions, see the chapter *Functions*. For detailed information about each attribute, see *Descriptions of extended keywords*, page 148.

TYPE ATTRIBUTES

Type attributes define how a function is called, or how a data object is accessed. This means that type attributes must be specified both when they are defined and in the declaration.

You can either place the type attributes directly in your source code, or use the pragma directive `#pragma type_attribute`.

The following general type attributes are available:

- *Function type attributes* change the calling convention of a function: `__arm`, `__fiq`, `__interwork`, `__irq`, `__monitor`, `__swi`, and `__thumb`
- *Data type attributes*: `const` and `volatile`

For each level of indirection, you can specify as many type attributes as required.

OBJECT ATTRIBUTES

Object attributes affect function and data objects, but not how the function is called or how the data is accessed. The object attribute does not affect the object interface. This means that an object attribute does not need to be present in the declaration of an object.

The following object attributes are available:

- Object attributes that can be used for *variables*: `__no_init`
- Object attributes that can be used for *functions* and *variables*: `location`, `@`, and `__root`
- Object attributes that can be used for *functions*: `intrinsic`, `__ramfunc`, and `vector`

Note: The `intrinsic` attribute is reserved for compiler internal use only.

You can specify as many object attributes as required.

DECLARING OBJECTS IN C SOURCE FILES

When declaring objects, note that the IAR-specific attributes work exactly like `const`. One exception to this is attributes that are declared in front of the type specifier apply to all declared objects.

Data types in C++

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

Segment reference

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

For information about how to define segments in the linker command file, see *Customizing the linker command file*, page 25.

Summary of segments

The table below lists the segments that are available in the ARM IAR C/C++ Compiler. Note that *located* denotes absolute location using the @ operator or the #pragma location directive. The XLINK segment memory type CODE, CONST, or DATA indicates whether the segment should be placed in ROM or RAM memory areas; see Table 3, *XLINK segment memory types*, page 24.

Segment	Description	Type
CODE	Holds code	CODE
CODE_I	Holds code declared __ramfunc	DATA
CODE_ID	Holds code copied to CODE_I at startup (ROM)	CONST
CSTACK	Holds the stack used by C or C++ programs	DATA
DATA_AC	Located initialized const objects	CONST
DATA_AN	Located objects, declared with the __no_init keyword	DATA
DATA_C	Constant data, including string literals	CONST
DATA_I	Static and global variables declared with non-zero initial values	DATA
DATA_ID	Initial values copied to DATA_I at cstartup	CONST
DATA_N	Static and global variables, declared with the __no_init keyword	DATA
DATA_Z	Static and global variables, declared without an initial value or with zero initial values	DATA
DIFUNCT	Holds pointers to code, typically C++ constructors, which should be executed by cstartup before main is called.	CODE
HEAP	Heap segment for dynamically allocated data	DATA

Table 27: Segment summary

Segment	Description	Type
ICODE	Startup code	CODE
INITTAB	Addresses and sizes of segments to be initialized at startup	CONST
INTVEC	Reset and exception vectors	CODE
IRQ_STACK	Stack for interrupt requests, IRQ, exceptions	DATA
SWITAB	Software interrupt vector table	CODE

Table 27: Segment summary (Continued)

Descriptions of segments

The following section gives reference information about each segment. Many of the extended keywords supported by the compiler are mentioned here. For detailed information about the keywords, see the chapter *Extended keywords*.

CODE Holds program code; this code will be executed in ROM.

Linker segment type

CODE

Type

Read-only.

Memory range

This segment can be placed anywhere in memory.

CODE_I Holds program code declared `__ramfunc`; this code will be executed in RAM. The code is copied from CODE_ID during initialization.

Linker segment type

DATA

Type

Read/(write).

Memory range

This segment can be placed anywhere in memory.

`CODE_ID` Permanent storage for program code declared `__ramfunc` that will be executed in RAM. The code is copied to `CODE_I` during initialization.

Linker segment type

`CONST`

Type

Read-only.

Memory range

This segment can be placed anywhere in memory.

`CSTACK` Holds the internal data stack. This segment and its length is 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`

Type

Read/write.

Memory range

This segment can be placed anywhere in memory.

`DIFUNCT` Holds pointers to constructor blocks in C++.

Linker segment type

`CODE`

Type

Read-only.

Memory range

This segment can be placed anywhere in memory.

HEAP Holds dynamically allocated data, in other words data used by `malloc` and `free`, and in C++, `new` and `delete`.

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

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

where `HEAP_SIZE` is the size of the heap, *start* is the lowest possible memory location, and *end* is the highest possible memory location. Other segments may also be placed in the same range, as long as there is room enough for all of them within that range.

Linker segment type

DATA

Type

Read/write.

Memory range

This segment can be placed anywhere in memory.

DATA_AC Holds `const` declared objects that have an absolute address and which are either explicitly initialized to any value, or implicitly initialized to zero by the compiler. These objects are given an absolute location using the `@` operator or the `#pragma location` directive. Because this segment contains objects which already have a fixed address, it should not be defined in the linker command file.

Linker segment type

CONST

Type

Read-only.

Memory range

This segment can be placed anywhere in memory.

`DATA_AN` Holds objects, declared with the `__no_init` keyword, that have an absolute address. Because this segment contains objects which already have a fixed address, it should not be defined in the linker command file.

Linker segment type

DATA

Type

Read/write.

Memory range

This segment can be placed anywhere in memory.

`DATA_C` Holds constant data, including string literals.

Linker segment type

CONST

Type

Read-only.

Memory range

This segment can be placed anywhere in memory.

`DATA_I` Holds static and global initialized variables that have been declared with non-zero initial values. The initial values are copied by `cstartup` from the `DATA_ID` segment during initialization.

Linker segment type

DATA

Type

Read/write.

Memory range

This segment can be placed anywhere in memory.

DATA_ID Holds initial values for the variables located in the `DATA_I` segment. These values are copied by `cstartup` from `DATA_ID` to `DATA_I` during system initialization.

Linker segment type

CONST

Type

Read-only.

Memory range

This segment can be placed anywhere in memory.

DATA_N Holds variables to be placed in non-volatile memory. These have been allocated by the compiler, declared `__no_init` or created `__no_init` by use of the `#pragma memory` directive.

Linker segment type

DATA

Type

Read/write.

Memory range

This segment can be placed anywhere in memory.

DATA_Z Holds static and global variables that have been declared without an initial value or with a zero-initialized value. Standard C specifies that such variables be set to zero before they are encountered by the program, so they are set to zero by `cstartup` during initialization.

Linker segment type

DATA

Type

Read/write.

Memory range

This segment can be placed anywhere in memory.

ICODE Holds startup code; these functions are reached by a branch instruction from INTVEC.

Linker segment type

CODE

Type

Read-only.

Memory range

This segment can be placed anywhere within the first 32 Mbytes of memory so that it can be reached from the INTVEC segment.

INITTAB Holds the table containing addresses and sizes of segments that need to be initialized at startup.

Linker segment type

CONST

Type

Read-only.

Memory range

This segment can be placed anywhere in memory.

INTVEC Holds the reset vector and exceptions vectors which contain branch instructions to `cstartup`, interrupt service routines, etc.

Linker segment type

CODE

Type

Read-only.

Memory range

Must be placed at address range 0x00 to 0x3F.

IRQ_STACK This stack is used when servicing IRQ exceptions. Other stacks may be added as needed for servicing other exception types: FIQ, SVC, ABF, and UND. The `cstartup.s79` file must be modified to initialize the exception stack pointers used.

Linker segment type

DATA

Type

Read/write.

Memory range

This segment can be placed anywhere in memory.

SWITAB Holds the software interrupt vector table.

Linker segment type

CODE

Type

Read-only.

Memory range

This segment can be placed anywhere in memory.

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

Setting command line options

To set compiler options from the command line, include them on the command line after the `iccarm` 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:

```
iccarm 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`:

```
iccarm 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:

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

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

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

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

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

```
iccarm 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`:

```
iccarm prog -l ---r
```

SPECIFYING ENVIRONMENT VARIABLES

Compiler options can also be specified in the `QCCARM` 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 ARM IAR C/C++ Compiler:

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

Table 28: Environment variables

ERROR RETURN CODES

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

The following command line error codes are supported:

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

Table 29: Error return codes

Options summary

The following table summarizes the compiler command line options:

Command line option	Description
<code>--char_is_signed</code>	char is treated as signed char
<code>--cpu=core</code>	Selects processor variant

Table 30: Compiler options summary

Command line option	Description
--cpu_mode={arm a thumb t}	Sets the default mode for functions
-Dsymbol [=value]	Defines preprocessor symbols
--debug	Generates debug information
--dependencies[=[i][m]] {filename directory}	Lists file dependencies
--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
--diagnostics_tables {filename directory}	Lists all diagnostic messages
-e	Enables language extensions
--ec++	Enables Standard Embedded C++ syntax
--eec++	Enables Extended Embedded C++ syntax
--enable_multibytes	Enables support for multibyte characters
--endian={big b little l}	Specifies byte order
-f filename	Extends the command line
--fpu={VFPv1 VFPv2 VFP9-S none}	Selects type of floating-point unit
--header_context	Lists all referred source files
-Ipath	Specifies include file path
--interwork	Generates interworking code
-l[a A b B c C D][N][H] {filename directory}	Creates list file
--library_module	Makes library module
--migration_preprocessor_extensions	Extends the preprocessor
--module_name=name	Sets object module name
--no_clustering	Disables static clustering
--no_code_motion	Disables code motion optimization
--no_cse	Disables common sub-expression elimination

Table 30: Compiler options summary (Continued)

Command line option	Description
<code>--no_inline</code>	Disables function inlining
<code>--no_scheduling</code>	Disables instruction scheduling
<code>--no_tbaa</code>	Disables type-based alias analysis
<code>--no_typedefs_in_diagnostics</code>	Prevents use of typedef names in diagnostics
<code>--no_unroll</code>	Disables loop unrolling
<code>--no_warnings</code>	Disables all warnings
<code>--no_wrap_diagnostics</code>	Disables wrapping of diagnostic messages
<code>-o {filename directory}</code>	Sets object filename
<code>--omit_types</code>	Excludes type information
<code>--only_stdout</code>	Uses standard output only
<code>--preinclude includefile</code>	Includes an include file before reading the source file
<code>--preprocess[=[c][n][l]] {filename directory}</code>	Generates preprocessor output
<code>--public_equ symbol[=value]</code>	Defines a global named assembler label
<code>-r</code>	Generates debug information
<code>--remarks</code>	Enables remarks
<code>--require_prototypes</code>	Verifies that prototypes are proper
<code>-s[2 3 6 9]</code>	Optimizes for speed
<code>--segment tag=segmentname</code>	Assigns name to segment
<code>--separate_cluster_for_initialized_variables</code>	Separates initialized and non-initialized variables
<code>--silent</code>	Sets silent operation
<code>--stack_align [4 8]</code>	Sets stack alignment
<code>--strict_ansi</code>	Enables strict ISO/ANSI C
<code>--warnings_affect_exit_code</code>	Warnings affect exit code
<code>--warnings_are_errors</code>	Treats all warnings as errors
<code>-z[2 3 6 9]</code>	Optimizes for size

Table 30: Compiler options summary (Continued)

Descriptions of options

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

`--char_is_signed` `--char_is_signed`

By default, the compiler interprets the `char` type as unsigned. The `--char_is_signed` option causes the compiler to interpret the `char` type as signed instead. This can be useful when you, for example, want to maintain compatibility with another compiler.

Note: The runtime library is compiled without the `--char_is_signed` option. If you use this option, you may get type mismatch warnings from the IAR XLINK Linker, because the library uses unsigned chars.



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

`--cpu` `--cpu=core`

Use this option to select the processor variant for which the code is to be generated. The default is ARM7TDMI.

For additional information, see *Processor variant*, page 5.



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

`--cpu_mode` `--cpu_mode {arm|a|thumb|t}`

Use this option to select the default mode for functions. This setting must be the same for all files included in a program, unless they are all compiled with the `--interwork` option, see page 134, or the `__interwork` keyword is used, page 149.



This option is related to the **CPU mode** 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 1
```

In order to get the equivalence of:

```
#define FOO
```

specify the = sign but nothing after, for example:

```
-DFOO=
```

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

Example

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

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

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

Production version: `iccarm prog`

Test version: `iccarm prog -DTESTVER`



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

```
--debug, -r      --debug
                  -r
```

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

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



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

--dependencies --dependencies=[i] [m] { *filename* | *directory* }

Use this option to make the compiler write information to a file about each source code file opened by the compiler. The following modifiers are available:

Option modifier	Description
i	Lists only the names of files (default)
m	Lists in makefile style

Table 31: Generating a list of dependencies (--dependencies)

If a *filename* is specified, the compiler stores the output in that file.

If a *directory* is specified, the compiler stores the output in that directory, in a file with the extension *i*. The filename will be the same as the name of the compiled source file, unless a different name has been specified with the option -o, in which case that name will be used.

To specify the working directory, replace *directory* with a period (.).

If --dependencies or --dependencies=i is used, the name of each opened source file, including the full path, if available, is output on a separate line. For example:

```
c:\iar\product\include\stdio.h
d:\myproject\include\foo.h
```

If --dependencies=m is used, the output uses makefile style. For each source file, one line containing a makefile dependency rule is produced. Each line consists of the name of the object file, a colon, a space, and the name of a source file. For example:

```
foo.r79: c:\iar\product\include\stdio.h
foo.r79: d:\myproject\include\foo.h
```

Example 1

To generate a listing of file dependencies to the file `listing.i`, use:

```
iccarms prog --dependencies=i listing
```

Example 2

To generate a listing of file dependencies to a file called `listing.i` in the `mypath` directory, you would use:

```
iccarms prog --dependencies mypath\listing
```

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

Example 3

An example of using `--dependencies` with a popular make utility, such as `gmake` (GNU make):

- 1 Set up the rule for compiling files to be something like:

```
%.r79 : %.c
$(ICC) $(ICCFLAGS) $< --dependencies=m $*.d
```

That is, besides producing an object file, the command also produces a dependency file in makefile style (in this example, using the extension `.d`).

- 2 Include all the dependency files in the makefile using, for example:

```
-include $(sources:.c=.d)
```

Because of the `-`, it works the first time, when the `.d` files do not yet exist.

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

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

Example

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

```
--diag_error=Pe117
```



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

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

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

Example

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

```
--diag_remark=Pe177
```



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

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

Use this option to suppress diagnostic messages.

Example

The following example suppresses the warnings Pe117 and Pe177:

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



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

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

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

Example

The following example classifies the remark Pe826 as a warning:

```
--diag_warning=Pe826
```



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

```
--diagnostics_tables --diagnostics_tables {filename|directory}
```

Use this option to list all possible diagnostic messages in a named file. This can be very convenient, for example, if you have used a pragma directive to suppress or change the severity level of any diagnostic messages, but forgot to document why.

This option cannot be given together with other options.

If a *filename* is specified, the compiler stores the output in that file.

If a *directory* is specified, the compiler stores the output in that directory, in a file with the name `diagnostics_tables.txt`. To specify the working directory, replace *directory* with a period (.).

Example 1

To output a list of all possible diagnostic messages to the file `diag.txt`, use:

```
--diagnostics_tables diag
```

Example 2

If you want to generate a table to a file `diagnostics_tables.txt` in the working directory, you could use:

```
--diagnostics_tables .
```

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

```
-e -e
```

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

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

For additional information, see *Special support for embedded systems*, page 9.



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

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

In the ARM IAR C/C++ Compiler, the default language is C. If you use Embedded C++, you must use this option to set the language the compiler uses to Embedded C++.



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

```
--eec++ --eec++
```

In the ARM IAR C/C++ Compiler, the default language is C. If you take advantage of Extended Embedded C++ features like namespaces or the standard template library in your source code, you must use this option to set the language the compiler uses to Extended Embedded C++. See *Extended Embedded C++*, page 46.



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

```
--enable_multibytes --enable_multibytes
```

By default, multibyte characters cannot be used in C or C++ source code. If you use this option, multibyte characters in the source code are interpreted according to the host computer's default setting for multibyte support.

Multibyte characters are allowed in C and C++ style comments, in string literals, and in character constants. They are transferred untouched to the generated code.

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

```
--endian --endian={big|b|little|l}
```

Specifies the byte order of the generated code and data. By default, the compiler generates code in little-endian byte order. See also *Byte order*, page 7.



This option is related to the **Endian mode** option in the **General** category in the IAR Embedded Workbench.

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

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

Both C and C++ style comments are allowed in the file. Double quotes behave as in the Microsoft Windows command line environment.

Example

For example, you could replace the command line:

```
iccarm prog -r "-DUsername=John Smith" -DUserId=463760
```

with

```
iccarm prog -r -f userinfo
```

if the file `userinfo.xcl` contains:

```
"-DUsername=John Smith"
-DUserId=463760
```

```
--fpu --fpu={VFPv1|VFPv2|VFP9-S|none}
```

Use this option to generate code that carries out floating-point operations using a Vector Floating Point (VFP) coprocessor. By selecting a VFP coprocessor, you will override the use of the software floating-point library for all supported floating-point operations.

Select `VFPv1` support if you have a vector floating-point unit conforming to architecture VFPv1, such as the VFP10 rev 0. Similarly, select `VFPv2` on a system that implements a VFP unit conforming to architecture VFPv2, such as the VFP10 rev 1.

VFP9-S is an implementation of the VFPv2 architecture that can be used with the ARM9E family of CPU cores. Selecting the `VFP9-S` coprocessor is therefore identical to selecting the `VFPv2` architecture.

By selecting `none` (default) the software floating-point library is used.

```
--header_context --header_context
```

Occasionally, to find the cause of a problem it is necessary to know which header file was included from which source line. Use this option to list, for each diagnostic message, not only the source position of the problem, but also the entire include stack at that point.

```
-I -Ipath
-I path
```

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

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

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

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

it searches the following directories for the file to include:

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

- If the compiler encounters the name of an `#include` file in double quotes, for example:

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

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

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

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

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

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

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

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

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



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

`--interwork` `--interwork`

Use this option to generate interworking code.

In code compiled with this option, functions will by default be of the type `interwork`. It is possible to mix files compiled as `arm` and `thumb` (using the `--cpu_mode` option) as long as they are all compiled with `--interwork`.

`-l` `[a|A|b|B|c|C|D] [N] [H] {filename|directory}`

By default, the compiler does not generate a listing. Use this option to generate a listing to a file.

The following modifiers are available:

Option modifier	Description
a	Assembler list file
A	Assembler file with C or C++ source as comments
b	Basic assembler list file. This file has the same contents as a list file produced with <code>-la</code> , except that none of the extra compiler generated information (runtime model attributes, call frame information, frame size information) is included *
B	Basic assembler list file. This file has the same contents as a list file produced with <code>-lA</code> , except that none of the extra compiler generated information (runtime model attributes, call frame information, frame size information) is included *
c	C or C++ list file
C (default)	C or C++ list file with assembler source as comments
D	C or C++ list file with assembler source as comments, but without instruction offsets and hexadecimal byte values
N	No diagnostics in file
H	Include source lines from header files in output. Without this option, only source lines from the primary source file are included

Table 32: Generating a compiler list file (-l)

*** This makes the list file less useful as input to the assembler, but more useful for reading by a human.**

If a *filename* is specified, the compiler stores the output in that file.

If a *directory* is specified, the compiler stores the output in that directory, in a file with the extension `lst`. The filename will be the same as the name of the compiled source file, unless a different name has been specified with the option `-o`, in which case that name will be used.

To specify the working directory, replace *directory* with a period (`.`).

Example 1

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

```
iccam prog -l list
```

Example 2

If you compile the file `mysource.c` and want to generate a listing to a file `mysource.lst` in the working directory, you could use:

```
icarm mysource -l .
```



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

```
--library_module --library_module
```

Use this option to make the compiler generate a library module rather than a program module. A program module is always included during linking. A library module will only be included if it is referenced in your program.



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

```
--migration_preprocessor_extensions
--migration_preprocessor_extensions
```

If you need to migrate code from an earlier IAR C or C/C++ compiler, you may want to use this option. With this option, the following can be used in preprocessor expressions:

- Floating-point expressions
- Basic type names and `sizeof`
- All symbol names (including typedefs and variables).

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

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

```
--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:

```
icarm prog --module_name=main
```

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

Example

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

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

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



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

`--no_clustering` `--no_clustering`

When static clustering is enabled, static and global variables are arranged so that variables that are accessed in the same function are stored close to each other. This makes it possible for the compiler to use the same base pointer for several accesses. Alignment gaps between variables can also be eliminated.

Use `--no_clustering` to disable static clustering.



This option is related to the **Optimization** options in the **C/C++ Compiler** 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. However, the resulting code may be difficult to debug.

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



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

`--no_cse` `--no_cse`

Use `--no_cse` to disable common subexpression elimination.

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

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



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

`--no_inline` `--no_inline`

Use `--no_inline` to disable function inlining.

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

This optimization, which is performed at optimization level 9, normally reduces execution time and increases code size. The resulting code may also be difficult to debug.

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

Note: This option has no effect at optimization levels below 9.



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

`--no_scheduling` `--no_scheduling`

The ARM IAR C/EC++ Compiler features an instruction scheduler to increase the performance of the generated code. To achieve that goal, the scheduler rearranges the instructions to minimize the number of pipeline stalls emanating from resource conflicts within the microprocessor.

Use `--no_scheduling` to disable the instruction scheduler.



This option is related to the **Optimization** options in the **C/C++ Compiler** category in the IAR Embedded Workbench.

```
--no_tbaa --no_tbaa
```

Use `--no_tbaa` to disable type-based alias analysis. When this options is not used, the compiler is free to assume that objects are only accessed through the declared type or through unsigned char. See *Type-based alias analysis*, page 90 for more information.



This option is related to the **Optimization** options in the **C/C++ Compiler** category in the IAR Embedded Workbench.

```
--no_typedefs_in_diagnostics --no_typedefs_in_diagnostics
```

Normally, when a type is printed by the compiler, most commonly in a diagnostic message of some kind, typedef names that were used in the original type are used whenever they make the resulting text shorter. For example,

```
typedef int (*MyPtr)(char const *);
MyPtr p = "foo";
```

will give an error message like the following:

```
Error[Pe144]: a value of type "char *" cannot be used to
initialize an entity of type "MyPtr"
```

If the `--no_typedefs_in_diagnostics` option is specified, the error message will be like this:

```
Error[Pe144]: a value of type "char *" cannot be used to
initialize an entity of type "int (*)(char const *)"
```

```
--no_unroll --no_unroll
```

Use this option to disable loop unrolling.

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

For small loops, the overhead required to perform the looping can be large compared with the work performed in the loop body.

The loop unrolling optimization duplicates the body several times, reducing the loop overhead. 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.

Note: This option has no effect at optimization levels below 9.



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

`--no_warnings` `--no_warnings`

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



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

`--no_wrap_diagnostics` `--no_wrap_diagnostics`

By default, long lines in compiler diagnostic messages are broken into several lines to make the message easier to read. Use this option to disable line wrapping of diagnostic messages.

`-o` `-o {filename|directory}`

Use the `-o` option to specify an output file for object code.

If a *filename* is specified, the compiler stores the object code in that file.

If a *directory* is specified, the compiler stores the object code in that directory, in a file with the same name as the name of the compiled source file, but with the extension *r79*. To specify the working directory, replace *directory* with a period (*.*).

Example 1

To store the compiler output in a file called `obj.r79` in the `mypath` directory, you would use:

```
iccarm mysource -o mypath\obj
```

Example 2

If you compile the file `mysource.c` and want to store the compiler output in a file `mysource.r79` in the working directory, you could use:

```
iccarm mysource -o .
```



To set the equivalent option in the IAR Embedded Workbench, select **Project>Options>General Options>Output Directories**.

`--omit_types` `--omit_types`

By default, the compiler includes type information about variables and functions in the object output.

Use this option if you do not want the compiler to include this type information in the output. The object file will then only contain type information that is a part of a symbol's name. This means that the linker cannot check symbol references for type correctness, which is useful when you build a library that should not contain type information.

`--only_stdout` `--only_stdout`

Use this option to make the compiler use the standard output stream (`stdout`) also for messages that are normally directed to the error output stream (`stderr`).

`--preinclude` `--preinclude includefile`

Use this option to make the compiler include the specified include file before it starts to read the source file. This is useful if you want to change something in the source code for the entire application, for instance if you want to define a new symbol.

`--preprocess` `--preprocess [= [c] [n] [l]] {filename|directory}`

Use this option to direct preprocessor output to a named file.

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

Command line option	Description
<code>--preprocess=c</code>	Preserve comments
<code>--preprocess=n</code>	Preprocess only
<code>--preprocess=l</code>	Generate #line directives

Table 33: Directing preprocessor output to file (`--preprocess`)

If a *filename* is specified, the compiler stores the output in that file.

If a *directory* is specified, the compiler stores the output in that directory, in a file with the extension `i`. The filename will be the same as the name of the compiled source file, unless a different name has been specified with the option `-o`, in which case that name will be used.

To specify the working directory, replace *directory* with a period (`.`).

Example 1

To store the compiler output with preserved comments to the file `output.i`, use:

```
icccarm prog --preprocess=c output
```

Example 2

If you compile the file `mysource.c` and want to store the compiler output with `#line` directives to a file `mysource.i` in the working directory, you could use:

```
icccarm mysource --preprocess=1 .
```



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

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

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

`-r, --debug` `-r`
 `--debug`

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

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



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

`--remarks` `--remarks`

The least severe diagnostic messages are called remarks (see *Severity levels*, page 207). A remark indicates a source code construct that may cause strange behavior in the generated code.

By default, the compiler does not generate remarks. Use this option to make the compiler generate remarks.



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

`--require_prototypes` `--require_prototypes`

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

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

`-s -s[2|3|6|9]`

Use this option to make the compiler optimize the code for maximum execution speed. If no optimization option is specified, the compiler will use the size optimization `-z3` by default. If the `-s` option is used without specifying the optimization level, speed optimization at level 3 is used by default.

The following table shows how the optimization levels are mapped:

Option modifier	Optimization level
2	None* (Best debug support)
3	Low*
6	Medium
9	High (Maximum optimization)

Table 34: Specifying speed optimization (-s)

***The most important difference between `-s2` and `-s3` is that at level 2, all non-static variables will live during their entire scope.**

A low level of optimization makes it relatively easy to follow the program flow in the debugger, and, conversely, a high level of optimization makes it relatively hard.

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



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

`--segment --segment tag=segmentname`

Use this option to place any part of your application into separate non-default segments. Any segment name can be used. The valid tags are `code` and `data`. By default, data objects are placed into segments named `DATA_*`, where `*` is any of `AC`, `AN`, `C`, `I`, `ID`, `N`, or `Z`.

By using the following command line option when compiling all, or part of your application, you can create a number of new segments named `SRAM_*` for your data objects:

`--segment data=SRAM`

Similarly, code is normally placed into segments named `CODE`, `CODE_I`, and `CODE_ID`. By using the following command line option, the segments `FLASH`, `FLASH_I`, and `FLASH_ID` are created if needed:

```
--segment code=FLASH
```

```
--separate_cluster_for_initialized_variables
```

```
--separate_cluster_for_initialized_variables
```

Separates initialized and non-initialized variables when using variable clustering. Makes the data `*_ID` segments smaller but can result in larger code.

```
--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.

```
--stack_align
```

```
--stack_align [4|8]
```

Use this option to increase the stack alignment at function entry from 4 to 8. The default is 4.

```
--strict_ansi
```

```
--strict_ansi
```

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

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



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

```
--warnings_affect_exit_code
```

```
--warnings_affect_exit_code
```

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



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

`--warnings_are_errors` `--warnings_are_errors`

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

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

For additional information, see `--diag_warning`, page 130 and `#pragma diag_warning`, page 156.



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

`-z` `-z [2 | 3 | 6 | 9]`

Use this option to make the compiler optimize the code for minimum size. If no optimization option is specified, `-z3` is used by default.

The following table shows how the optimization levels are mapped:

Option modifier	Optimization level
2	None* (Best debug support)
3	Low*
6	Medium
9	High (Maximum optimization)

Table 35: Specifying size optimization (-z)

***The most important difference between `-z2` and `-z3` is that at level 2, all non-static variables will live during their entire scope.**

A low level of optimization makes it relatively easy to follow the program flow in the debugger, and, conversely, a high level of optimization makes it relatively hard.

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



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

Extended keywords

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

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

Using extended keywords

You can place the keyword directly in the code, or 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 ARM IAR C/C++ Compiler.



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



Use the `-e` compiler option to enable language extensions. See *-e*, page 131 for additional information.

Summary of extended keywords

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

Extended keyword	Description	Type
<code>__arm</code>	Makes a function execute in ARM mode	Function execution
<code>asm, __asm</code>	Inserts an assembler instruction	--
<code>__fiq</code>	Declares a fast interrupt function	Special function type
<code>__interwork</code>	Controls function calling conventions	Function execution
<code>__intrinsic</code>	Reserved for compiler internal use only	--
<code>__irq</code>	Declares an interrupt function	Special function type
<code>__monitor</code>	Supports atomic execution of a function	Special function type

Table 36: Extended keywords summary

Extended keyword	Description	Type
<code>__nested</code>	Allows an <code>__irq</code> declared interrupt function to be nested, that is, interruptable by the same type of interrupt	Function execution
<code>__no_init</code>	Supports non-volatile memory	Data storage
<code>__ramfunc</code>	Makes a function execute in RAM	Function execution
<code>__root</code>	Ensures that a function or variable is included in the object code even if unused	Function execution
<code>__swi</code>	Declares a software interrupt function	Special function type
<code>__thumb</code>	Makes a function execute in Thumb mode	Function execution

Table 36: Extended keywords summary (Continued)

Descriptions of extended keywords

The following sections give detailed information about each extended keyword.

`__arm` The `__arm` extended keyword makes a function execute in ARM mode. An `__arm` declared function can, unless it is also declared `__interwork`, only be called from functions that also execute in ARM mode.

A function declared `__arm` cannot be declared `__thumb`.

Example

```
__arm int func1(void);
```

`asm`, `__asm` The `asm` and `__asm` extended keywords both insert an assembler instruction. However, when compiling C source code, the `asm` keyword is not available when the option `--strict_ansi` is used. The `__asm` keyword is always available.

Syntax

```
asm ("string");
```

The string can be a valid assembler instruction or an assembler directive.

Note: Not all assembler directives can be inserted using this keyword.

For more information about inline assembler, see *Mixing C and assembler*, page 67.

__fiq This keyword declares a fast interrupt function. All interrupt functions must be compiled in ARM mode; use either the `__arm` keyword or the `#pragma type_attribute=__arm` directive to alter the default behavior if needed.

A function declared `__fiq` does not accept parameters and does not have a return value.

Example

```
__fiq __arm void interrupt_function(void);
```

__interwork A function declared `__interwork` can be called from functions executing in either ARM or Thumb mode.

Example

```
typedef void (__thumb __interwork *IntHandler)(void);
```

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

__irq This keyword declares an interrupt function. All interrupt functions must be compiled in ARM mode; use either the `__arm` keyword or the `#pragma type_attribute=__arm` directive to alter the default behavior if needed.

A function declared `__irq` does not accept parameters and does not have a return value.

Example

```
__irq __arm void interrupt_function(void);
```

__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. The `#pragma type_attribute` directive can also be used.

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 172, and `__enable_interrupt`, page 172.

Read more about monitor functions on page 21.

`__nested` The `__nested` keyword modifies the enter and exit code of an interrupt function to allow for nested interrupts. This allows interrupts to be enabled, which means new interrupts can be served inside an interrupt function, without overwriting the `SPSR` and return address in `R14`. Nested interrupts are only supported for `__irq` declared functions. For more information, see *Nested interrupts*, page 18.

Example

```
__irq __nested __arm void interrup_handler(void);
```

`__no_init` The `__no_init` keyword is used for suppressing initialization of a variable at system startup.

The `__no_init` keyword is placed in front of the type. In this example, `settings` is placed in the non-initialized segment `DATA_N`:

```
__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`.

`__ramfunc` This keyword makes a function execute in RAM. Two code segments will be created: one for the RAM execution, and one for the ROM initialization.

If a function declared `__ramfunc` tries to access ROM, the compiler will issue a warning. This behavior is intended to simplify the creation of *upgrade* routines, for instance, rewriting parts of flash memory. If this is not why you have declared the function `__ramfunc`, you may safely ignore or disable these warnings.

Functions declared `__ramfunc` are by default stored in the segment named `CODE_I`. See the chapter *Segment reference*.

`__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.

`__swi` This keyword declares a software interrupt function. All interrupt functions must be compiled in ARM mode; use either the `__arm` keyword or the `#pragma type_attribute=__arm` directive to alter the default behavior if needed.

A function declared `__swi` accepts arguments and return values. The `__swi` keyword also expects a software interrupt number which is specified with the `#pragma swi_number=number` directive. A `__swi` function can for example be declared in the following way:

```
#pragma swi_number=0x23
__swi __arm int swi_function(int a, int b);
```

The `swi_number` is used as an argument to the generated assembler `SWI` instruction, and can be used to select one software interrupt function in a system containing several such functions.

Software interrupt functions follow the same calling convention regarding parameters and return values as an ordinary function. Four registers (`R0–R3`) are used for passing parameters to a software interrupt function, and one or two registers (`R0–R1`) are used for the return values.

However, software interrupt functions cannot use the stack in the same way as an ordinary function. When an interrupt occurs, the processor switches to supervisor mode where the supervisor stack is used. Arguments can therefore not be passed on the stack if the program is not running in supervisor mode previous to the interrupt. As a consequence only the four registers `R0–R3` can be used for passing parameters.

`__thumb` This keyword makes a function execute in Thumb mode. Unless the function is also declared `__interwork`, the function declared `__thumb` can only be called from functions that also execute in Thumb mode.

A function declared `__thumb` cannot be declared `__arm`.

Example

```
__thumb int func2(void);
```

Pragma directives

This chapter describes the pragma directives of the ARM IAR C/C++ 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 compiler. They are consistent with ISO/ANSI C and are very useful when you want to make sure that the source code is portable.

Summary of pragma directives

The following table shows the pragma directives of the compiler:

Pragma directive	Description
<code>#pragma bitfields</code>	Controls the order of bitfield members
<code>#pragma data_alignment</code>	Gives a variable a higher (more strict) alignment
<code>#pragma diag_default</code>	Changes the severity level of diagnostic messages
<code>#pragma diag_error</code>	Changes the severity level of diagnostic messages
<code>#pragma diag_remark</code>	Changes the severity level of diagnostic messages
<code>#pragma diag_suppress</code>	Suppresses diagnostic messages
<code>#pragma diag_warning</code>	Changes the severity level of diagnostic messages
<code>#pragma include_alias</code>	Specifies an alias for an include file
<code>#pragma inline</code>	Inlines a function
<code>#pragma language</code>	Controls the IAR language extensions
<code>#pragma location</code>	Specifies the absolute address of a variable
<code>#pragma message</code>	Prints a message
<code>#pragma object_attribute</code>	Changes the definition of a variable or a function
<code>#pragma optimize</code>	Specifies type and level of optimization
<code>#pragma pack</code>	Specifies the alignment of structures and union members

Table 37: Pragma directives summary

Pragma directive	Description
#pragma required	Ensures that a symbol which is needed by another symbol is present in the linked output
#pragma rtmodel	Adds a runtime model attribute to the module
#pragma segment	Declares a segment name to be used by intrinsic functions
#pragma swi_number	Sets the interrupt number of a software interrupt function
#pragma type_attribute	Changes the declaration and definitions of a variable or function
#pragma vector	Specifies the vector of an interrupt function

Table 37: Pragma directives summary (Continued)

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

Descriptions of pragma directives

This section gives detailed information about each pragma directive.

All pragma directives using = for value assignment should be entered like:

```
#pragma pragmaname=pragmavalue
or
#pragma pragmaname = pragmavalue
```

```
#pragma bitfields #pragma bitfields={reversed|default}
```

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

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

```
#pragma data_alignment #pragma data_alignment=expression
```

Use this pragma directive to give a variable a higher (more strict) alignment than it would otherwise have. It can be used on variables with static and automatic storage duration.

The value of the constant *expression* must be a power of two (1, 2, 4, etc.).

When you use `#pragma data_alignment` on variables with automatic storage duration, there is an upper limit on the allowed alignment for each function, determined by the calling convention used.

```
#pragma 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 include_alias #pragma include_alias "orig_header" "subst_header"
```

```
#pragma include_alias <orig_header> <subst_header>
```

The `#pragma include_alias` directive makes it possible to provide an alias for a header file. This is useful for substituting one header file with another, and for specifying an absolute path to a relative file.

The parameter *subst_header* is used for specifying an alias for *orig_header*. This pragma directive must appear before the corresponding `#include` directives and *subst_header* must match its corresponding `#include` directive exactly.

Example

```
#pragma include_alias <stdio.h> <C:\MyHeaders\stdio.h>
#include <stdio.h>
```

This example will substitute the relative file `stdio.h` with a counterpart located according to the specified path.

```
#pragma inline #pragma inline[=forced]
```

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

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

Specifying `#pragma inline=forced` disables the compiler's heuristics and forces the inlining. If the inlining fails for some reason, for example if it cannot be used with the function type in question (like `printf`), an error message is emitted.

```
#pragma language #pragma language={extended|default}
```

The `#pragma language` directive is used for turning on the IAR language extensions or for using the language settings specified on the command line:

<code>extended</code>	Turns on the IAR language extensions and turns off the <code>--strict_ansi</code> command line option.
<code>default</code>	Uses the settings specified on the command line.

```
#pragma location #pragma location=address
```

The `#pragma location` directive specifies the location—the absolute address—of the variable whose declaration follows the pragma directive. For example:

```
#pragma location=0xFFFF0400
char PORT1; /* PORT1 is located at address 0xFFFF0400 */
```

The directive can also take a string specifying the segment placement for either a variable or a function, for example:

```
#pragma location="foo"
```

For additional information and examples, see *Located data*, page 31.

```
#pragma message #pragma message(message)
```

Makes the compiler print a message on `stdout` when the file is compiled. For example:

```
#ifdef TESTING
#pragma message("Testing")
#endif
```

```
#pragma object_attribute #pragma object_attribute=keyword
```

The `#pragma object_attribute` directive affects the definition of the identifier that follows immediately after the directive. The object is modified, not its type.

The `__no_init` extended keyword can be used with `#pragma object_attribute` for a variable. It suppresses initialization of the variable at startup.

The `__root` extended keyword can be used with `#pragma object_attribute` for a function or variable. It ensures that a function or data object is included in the object code even if not referenced.

The `__ramfunc` extended keyword can be used with `#pragma object_attribute` for a function. It makes the function execute in RAM.

Example

In the following example, the variable `bar` is placed in the non-initialized segment:

```
#pragma object_attribute=__no_init
char bar;
```

Unlike the directive `#pragma type_attribute` that specifies the storing and accessing of a variable, it is not necessary to specify an object attribute in declarations. The following example declares `bar` without a `#pragma object_attribute`:

```
__no_init char bar;
```

```
#pragma optimize #pragma optimize=token_1 token_2 token_3
```

where *token_n* is one of the following:

<code>s</code>	Optimizes for speed
<code>z</code>	Optimizes for size
<code>2 none 3 low 6 medium 9 high</code>	Specifies level of optimization
<code>no_code_motion</code>	Turns off code motion
<code>no_cse</code>	Turns off common sub-expression elimination
<code>no_inline</code>	Turns off function inlining
<code>no_tbaa</code>	Turns off type-based alias analysis
<code>no_unroll</code>	Turns off loop unrolling

The `#pragma optimize` directive is used for decreasing the optimization level, or for turning off some specific optimizations. This pragma directive only affects the function that follows immediately after the directive.

Note that it is not possible to optimize for speed and size at the same time. Only one of the `s` and `z` tokens can be used. It is also not possible to use macros embedded in this pragma directive. Any such macro will not get expanded by the preprocessor.

Note: If you use the `#pragma optimize` directive to specify an optimization level that is higher than the optimization level you specify using a compiler option, the pragma directive is ignored.

Example

```
#pragma optimize=s 9
int small_and_used_often()
{
    ...
}

#pragma optimize=z 9
int big_and_seldom_used()
{
    ...
}
```

```
#pragma pack ( [[ {push|pop}, ] [name, ] ] [n] )
```

<i>n</i>	Packing alignment, one of: 1, 2, 4, 8, or 16
<i>name</i>	Pushed or popped alignment label

The `#pragma pack` directive is used for specifying the alignment of structures and union members.

`pack (n)` sets the structure alignment to *n*. The `pack (n)` only affects declarations of structures following the pragma directive and to the next `#pragma pack` or end of file.

`pack ()` resets the structure alignment to default.

`pack (push [, name] [, n])` pushes the current alignment with the label *name* and sets alignment to *n*. Note that both *name* and *n* are optional.

`pack (pop [, name] [, n])` pops to the label *name* and sets alignment to *n*. Note that both *name* and *n* are optional.

If *name* is omitted, only top alignment is removed. If *n* is omitted, alignment is set to the value popped from the stack.

Note that accessing an object that is not aligned at its correct alignment requires code that is both larger and slower than the code needed to access the same kind of object when aligned correctly. If there are many accesses to such fields in the program, it is usually better to construct the correct values in a struct that is not packed, and access this instead.

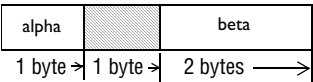
Also, special care is needed when creating and using pointers to misaligned fields. For direct access to such fields in a packed struct, the compiler will emit the correct (slower and larger) code when needed. However, when such a field is accessed through a pointer to the field, the normal (smaller and faster) code for accessing the type of the field is used, which will, in the general case, not work.

Example 1

This example declares a structure without using the `#pragma pack` directive:

```
struct First
{
    char alpha;
    short beta;
};
```

In this example, the structure `First` is not packed and has the following memory layout:



Note that one pad byte has been added.

Example 2

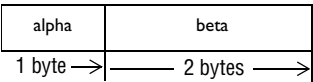
This example declares a similar structure using the `#pragma pack` directive:

```
#pragma pack(1)

struct FirstPacked
{
    char alpha;
    short beta;
};

#pragma pack()
```

In this example, the structure `FirstPacked` is packed and has the following memory layout:



Example 3

This example declares a new structure, `Second`, that contains the structure `FirstPacked` declared in the previous example. The declaration of `Second` is not placed inside a `#pragma pack` block:

```
struct Second
{
```

```

    struct FirstPacked first;
    short gamma;
};

```

The following memory layout is used:



Note that the structure `FirstPacked` will use the memory layout, size, and alignment described in example 2. The alignment of the member `gamma` is 2, which means that alignment of the structure `Second` will become 2 and one pad byte will be added.

```
#pragma required #pragma required=symbol
```

Use the `#pragma required` directive to ensure that a symbol which is needed by another symbol is present in the linked output. The *symbol* can be any statically linked function or variable, and the pragma directive must be placed immediately before a symbol definition.

Use the directive if the requirement for a symbol is not otherwise visible in the application, for example, if a variable is only referenced indirectly through the segment it resides in.

Example

```

void * const myvar_entry @ "MYSEG" = &myvar;
...
#pragma required=myvar_entry
long myvar;

```

```
#pragma rtmodel #pragma rtmodel="key", "value"
```

Use the `#pragma rtmodel` directive to add a runtime model attribute to a module. Use a text string to specify *key* and *value*.

This pragma directive is useful to enforce consistency between modules. All modules that are linked together and define the same runtime attribute key must have the same value for the corresponding key value, or the special value `*`. Using the special value `*` is equivalent to not defining the attribute at all. It can, however, be useful to state explicitly that the module can handle any runtime model.

A module can have several runtime model definitions.

Note: The predefined compiler runtime model attributes start with a double underscore. In order to avoid confusion, this style must not be used in the user-defined attributes.

Example

```
#pragma rtmodel="I2C", "ENABLED"
```

The linker will generate an error if a module that contains this definition is linked with a module that does not have the corresponding runtime model attributes defined.

For more information about runtime model attributes and module consistency, see *Checking module consistency*, page 61.

```
#pragma segment #pragma segment="segment"
```

The `#pragma segment` directive declares a segment name that can be used by the intrinsic functions `__segment_begin` and `__segment_end`.

Example

```
#pragma segment="MYSEG"
```

See also `__sfb`, `__segment_begin`, page 174.

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

```
#pragma swi_number #pragma swi_number=number
```

The `#pragma swi_number` directive is used in conjunction with the `__swi` extended keyword. It is used as an argument to the generated SWI assembler instruction, and can be used for selecting one software interrupt function in a system containing several such functions. For additional information, see *Software interrupts*, page 19.

```
#pragma type_attribute #pragma type_attribute=keyword
```

The following `#pragma type_attribute` directive affects the code generation of the next function that follows immediately after the pragma directive:

<code>__arm</code>	Makes a function execute in ARM mode.
<code>__fiq</code>	Declares a fast interrupt function.
<code>__interwork</code>	Makes a function execute in either ARM or Thumb mode.
<code>__irq</code>	Declares an interrupt function.

<code>__monitor</code>	Specifies a monitor function.
<code>__swi</code>	Declares a software interrupt function.
<code>__thumb</code>	Makes a function execute in Thumb mode.

For interrupt functions, use the `#pragma vector` directive to specify the exception vector.

Example

In the following example, thumb-mode code is generated for the function `foo`.

```
#pragma type_attribute=__thumb
void foo(void)
{
}
```

The following declaration, which use the corresponding extended keyword, is equivalent. See the chapter *Extended keywords* for more details.

```
__thumb void foo(void);
{
}
```

```
#pragma vector #pragma vector=vector
```

The `#pragma vector` directive specifies the vector of an interrupt function whose declaration follows the `pragma` directive.

Example

```
#pragma vector=0x14
__irq __arm void my_handler(void);
```


Predefined symbols

This chapter gives reference information about the predefined preprocessor symbols that are supported in the ARM IAR C/C++ 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	Identifies
<code>__ARM4TM__</code>	Identifies the processor core in use
<code>__ARM5__</code>	Identifies the processor core in use
<code>__ARM5T__</code>	Identifies the processor core in use
<code>__ARM5TM__</code>	Identifies the processor core in use
<code>__ARM5E__</code>	Identifies the processor core in use
<code>__ARMVFP__</code>	Identifies type of floating-point unit
<code>__CORE__</code>	Identifies the chip core in use
<code>__cplusplus</code>	Determines whether the compiler runs in C++ mode
<code>__CPU_MODE__</code>	Identifies the processor mode in use
<code>__DATE__</code>	Determines the date of compilation
<code>__embedded_cplusplus</code>	Determines whether the compiler runs in C++ mode
<code>__FILE__</code>	Identifies the name of the file being compiled
<code>__IAR_SYSTEMS_ICC__</code>	Identifies the IAR compiler platform
<code>__ICCARM__</code>	Identifies the ARM IAR C/C++ Compiler
<code>__LINE__</code>	Determines the current source line number
<code>__LITTLE_ENDIAN__</code>	Identifies the byte order in use
<code>__STDC__</code>	Identifies ISO/ANSI Standard C
<code>__STDC_VERSION__</code>	Identifies the version of ISO/ANSI Standard C in use
<code>__TID__</code>	Identifies the target processor of the IAR compiler in use
<code>__TIME__</code>	Determines the time of compilation
<code>__VER__</code>	Identifies the version number of the IAR compiler in use

Table 38: Predefined symbols summary

Descriptions of predefined symbols

The following section gives reference information about each predefined symbol.

__ARM4TM__ __ARM5__ __ARM5T__ __ARM5TM__ __ARM5E__	<p>When the compiler is generating code for an ARM core, the corresponding of these predefined symbols is defined. Subsets of the current core are also defined but with a smaller value. For example:</p> <pre>#if defined (__ARM5__) #include "inarm.h" #define my_own_CLZ_implementation (VAL) __CLZ(VAL) #endif</pre>
__ARMVFP__	<p>Enabling vector floating-point (VFP) code generation will set the macro __ARMVFP__ to a value of 1 for VFPv1, and a value of 2 for VFPv2. If VFP code generation is disabled (default), the macro will be undefined.</p>
__CORE__	<p>This predefined symbol expands to a number representing the core the compiler is generating code for. Currently the defines in the following example can be used for checking if a certain core level is used or not:</p> <pre>#include "inarm.h" #if __CORE__ >= __ARM5__ //__CLZ is already declared #else #define __CLZ (VAL) my_own_CLZ_implementation (VAL) #endif</pre>
__cplusplus	<p>This predefined symbol expands to the number 199711L when the compiler runs in any of the C++ modes. When the compiler runs in ISO/ANSI C mode, the symbol is undefined.</p> <p>This symbol can be used with <code>#ifdef</code> to detect whether the compiler accepts C++ code. It is particularly useful when creating header files that are to be shared by C and C++ code.</p>

`__CPU_MODE__` This predefined symbol expands to a number reflecting the default CPU mode in use:

Value	CPU mode
1	Thumb
2	ARM

Table 39: Predefined symbols for inspecting the CPU mode

`__DATE__` Use this symbol to identify when the file was compiled. This symbol expands to the date of compilation, which is returned in the form "Mmm dd yyyy", for example "Jan 30 2002".

`__embedded_cplusplus` This predefined symbol expands to the number 1 when the compiler runs in any of the C++ modes. When the compiler runs in ISO/ANSI C mode, the symbol is undefined.

`__FILE__` Use this symbol to identify which file is currently being compiled. This symbol expands to the name of that file.

`__IAR_SYSTEMS_ICC__` This predefined symbol expands to a number that identifies the IAR compiler platform. The current identifier is 6. Note that the number could be higher in a future version of the product.

This symbol can be tested with `#ifdef` to detect whether the code was compiled by a compiler from IAR Systems.

`__ICCARM__` This predefined symbol expands to the number 1 when the code is compiled with the ARM IAR C/C++ Compiler.

`__LINE__` This predefined symbol expands to the current line number of the file currently being compiled.

`__LITTLE_ENDIAN__` This predefined symbol expands to the number 1 when the code is compiled with the little-endian byte order format, and 0 when the code is compiled with the big-endian byte order format.

`__STDC__` This predefined symbol expands to the number 1. This symbol can be tested with `#ifdef` to detect whether the compiler in use adheres to ISO/ANSI C.

__STDC_VERSION__	<div>ISO/ANSI C and version identifier.</div> <div>This predefined symbol expands to the number 199409L.</div> <div>Note: This predefined symbol does not apply in EC++ mode.</div>										
__TID__	<div>Target identifier for the ARM IAR C/C++ Compiler.</div> <div>Expands to the target identifier which contains the following parts:</div> <div><ul style="list-style-type: none">● A one-bit intrinsic flag (<i>i</i>) which is reserved for use by IAR● A target-identifier (<i>t</i>) unique for each IAR compiler. For the ARM compiler, the target identifier is 79● A value (<i>c</i>) reserved for specifying different CPU cores. The value is derived from the setting of the <code>--cpu</code> option:</div> <div><table><tr><th>Value</th><th>CPU core</th></tr><tr><td>0</td><td>Unspecified</td></tr><tr><td>1</td><td>ARM7TDMI</td></tr><tr><td>2</td><td>ARM9TDMI</td></tr><tr><td>3</td><td>ARM9E</td></tr></table></div> <div>Table 40: Values for specifying different CPU cores in __TID__</div> <div>The __TID__ value is constructed as:</div> <div>$((i \ll 15) \mid (t \ll 8) \mid (c \ll 4))$</div> <div>You can extract the values as follows:</div> <div><pre>i = (__TID__ >> 15) & 0x01; /* intrinsic flag */ t = (__TID__ >> 8) & 0x7F; /* target identifier */ c = (__TID__ >> 4) & 0x0F; /* cpu core */</pre></div> <div>To find the value of the target identifier for the current compiler, execute:</div> <div><pre>printf("%ld", (__TID__ >> 8) & 0x7F)</pre></div> <div>Note: Because coding may change or functionality may be entirely removed in future versions, the use of __TID__ is not recommended. We recommend that you use the symbols __ICCARM__ and __CORE__ instead.</div>	Value	CPU core	0	Unspecified	1	ARM7TDMI	2	ARM9TDMI	3	ARM9E
Value	CPU core										
0	Unspecified										
1	ARM7TDMI										
2	ARM9TDMI										
3	ARM9E										
__TIME__	<div>Current time.</div> <div>Expands to the time of compilation in the form <code>hh:mm:ss</code>.</div>										

`__VER__` Compiler version number.

Expands to an integer representing the version number of the compiler. The value of the number is calculated in the following way:

*(100 * the major version number + the minor version number)*

Example

The example below prints a message for version 3.34.

```
#if __VER__ == 334
#pragma message("Compiler version 3.34")
#endif
```

In this example, 3 is the major version number and 34 is the minor version number.

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

The following table summarizes the intrinsic functions:

Intrinsic function	Description
<code>__CLZ</code>	Generates a CLZ instruction
<code>__disable_interrupt</code>	Disables interrupts
<code>__enable_interrupt</code>	Enables interrupts
<code>__get_CPSR</code>	Returns the value of the ARM CPSR (Current Program Status Register)
<code>__MCR</code>	Generates a coprocessor write instruction (MCR).
<code>__MRC</code>	Generates a coprocessor read instruction (MRC).
<code>__no_operation</code>	Generates a NOP instruction
<code>__set_CPSR</code>	Sets the value of the ARM CPSR (Current Program Status Register)
<code>__sfb, __segment_begin</code>	Returns the start address of a segment
<code>__sfe, __segment_end</code>	Returns the end address of a segment
<code>__sfs, __segment_size</code>	Returns the size of a segment
<code>__QADD</code>	Generates a QADD instruction
<code>__QDADD</code>	Generates a QDADD instruction
<code>__QDSUB</code>	Generates a QDSUB instruction
<code>__QSUB</code>	Generates a QSUB instruction

Table 41: Intrinsic functions summary

To use intrinsic functions in an application, include the header file `inarm.h`.

Note that the intrinsic function names start with double underscores, for example:

`__segment_begin`

Descriptions of intrinsic functions

The following section gives reference information about each intrinsic function.

`__CLZ` `unsigned char __CLZ(unsigned long);`

Inserts a CLZ instruction. This intrinsic function requires an ARM v5 architecture.

`__disable_interrupt` `void __disable_interrupt(void);`

In ARM mode, this intrinsic function disables interrupts by setting bits 6 and 7 of the CPSR register.

In Thumb mode, this intrinsic function inserts a function call to a library function.

Note: This intrinsic can only be used in supervisor mode.

`__enable_interrupt` `void __enable_interrupt(void);`

In ARM mode, this intrinsic function enables interrupts by clearing bits 6 and 7 of the CPSR register.

In Thumb mode, this intrinsic function inserts a function call to a library function.

Note: This intrinsic can only be used in supervisor mode.

`__get_CPSR` `unsigned long __get_CPSR(void);`

Returns the value of the ARM CPSR (Current Program Status Register). This intrinsic function requires ARM mode.

`__MCR` `void __MCR(__ul coproc, __ul opcode_1, __ul src, __ul CRn, __ul CRm, __ul opcode_2);`

Generates a coprocessor write instruction (MCR). A value will be written to a coprocessor register. The parameters `coproc`, `opcode_1`, `CRn`, `CRm`, and `opcode_2` will be encoded in the MCR instruction operation code and must therefore be constants. The following parameters are used:

- | | |
|-----------------------|--------------------------------------|
| <code>coproc</code> | The coprocessor number 0..15. |
| <code>opcode_1</code> | Coprocessor-specific operation code. |

<code>src</code>	The value to be written to the coprocessor.
<code>CRn</code>	The coprocessor register to write to.
<code>CRm</code>	Additional coprocessor register; set to zero if not used.
<code>opcode_2</code>	Additional coprocessor-specific operation code; set to zero if not used.

This intrinsic function requires ARM mode.

```
__MRC unsigned long __MRC(__ul coproc, __ul opcode_1, __ul CRn, __ul
                        CRm, __ul opcode_2);
```

Generates a coprocessor read instruction (MRC). Returns the value of the specified coprocessor register. The parameters `coproc`, `opcode_1`, `CRn`, `CRm`, and `opcode_2` will be encoded in the MRC instruction operation code and must therefore be constants. The following parameters are used:

<code>coproc</code>	The coprocessor number 0..15.
<code>opcode_1</code>	Coprocessor-specific operation code.
<code>CRn</code>	The coprocessor register to write to.
<code>CRm</code>	Additional coprocessor register; set to zero if not used.
<code>opcode_2</code>	Additional coprocessor-specific operation code; set to zero if not used.

This intrinsic function requires ARM mode.

```
__no_operation void __no_operation(void);
```

Inserts a dummy MOV instruction.

The intrinsic `no_operation` is equivalent.

```
__set_CPSR void __set_CPSR(unsigned long);
```

Sets the value of the ARM CPSR (Current Program Status Register). Only the control field is changed (bits 0-7). This intrinsic function requires ARM mode.

```
__sfb, __segment_begin void * __segment_begin(segment);
```

Returns the address of the first byte of the named *segment*. The named *segment* must be a string literal that has been declared earlier with the `#pragma segment` directive. See *#pragma segment*, page 162.

Example

```
#pragma segment="MYSEG"
...
segment_start_address = __segment_begin("MYSEG");
```

```
__sfe, __segment_end void * __segment_end(segment);
```

Returns the address of the first byte *after* the named *segment*. The named *segment* must be a string literal that has been declared earlier with the `#pragma segment` directive. See *#pragma segment*, page 162.

Example

```
#pragma segment="MYSEG"
...
segment_end_address = __segment_end("MYSEG");
```

```
__sfs, __segment_size int __segment_size(segment);
```

Returns the size of the named *segment*. The named *segment* must be a string literal that has been declared earlier with the `#pragma segment` directive. See *#pragma segment*, page 162.

Example

```
#pragma segment="MYSEG"
...
segment_size = __segment_size("MYSEG");
```

```
__QADD signed long __QADD(signed long, signed long);
```

Inserts a QADD instruction. This intrinsic function requires an ARM v5E architecture.

```
__QDADD signed long __QDADD(signed long, signed long);
```

Inserts a QDADD instruction. This intrinsic function requires an ARM v5E architecture.

```
__QDSUB signed long __QDSUB(signed long, signed long);
```

Inserts a `QDSUB` instruction. This intrinsic function requires an ARM v5E architecture.

```
__QSUB signed long __QSUB(signed long, signed long);
```

Inserts a `QSUB` instruction. This intrinsic function requires an ARM v5E architecture.

Library functions

This chapter gives an introduction to the C and C++ library functions. It also lists the header files used for accessing library definitions.

Introduction

The ARM IAR C/C++ Compiler comes with the IAR DLIB Library, which is a complete ISO/ANSI C and C++ library. This library also supports floating-point numbers in IEEE 754 format and it can be configured to include different levels of support for locale, file descriptors, multibytes, et cetera.

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

For detailed information about the library functions, see the online documentation supplied with the product. There is also keyword reference information for the DLIB library functions. To obtain reference information for a function, select the function name in the editor window and press F1.

For additional information about library functions, see the chapter *Implementation-defined behavior* in this guide.

HEADER FILES

Your application program gains access to library definitions through header files, which it incorporates using the `#include` directive. The definitions are divided into a number of different header files, each covering a particular functional area, letting you include just those that are required.

It is essential to include the appropriate header file before making any reference to its definitions. Failure to do so can cause the call to fail during execution, or generate error or warning messages at compile time or link time.

LIBRARY OBJECT FILES

Most of the library definitions can be used without modification, that is, directly from the library object files that are supplied with the product. For information about how to choose a runtime library, see *Basic settings for project configuration*, page 5. The linker will include only those routines that are required—directly or indirectly—by your application.

REENTRANCY

A function that can be simultaneously invoked in the main application and in any number of interrupts is reentrant. A library function that uses statically allocated data is therefore not reentrant. Most parts of the DLIB library are reentrant, but the following functions and parts are not reentrant:

<code>atexit</code>	Needs static data
heap functions	Need static data for memory allocation tables
<code>strerror</code>	Needs static data
<code>strtok</code>	Designed by ISO/ANSI standard to need static data
I/O	Every function that uses files in some way. This includes <code>printf</code> , <code>scanf</code> , <code>getchar</code> , and <code>putchar</code> . The functions <code>sprintf</code> and <code>sscanf</code> are not included.

In addition, some functions share the same storage for `errno`. These functions are not reentrant, since an `errno` value resulting from one of these functions can be destroyed by a subsequent use of the function before it has been read. Among these functions are:

`exp`, `exp10`, `ldexp`, `log`, `log10`, `pow`, `sqrt`, `acos`, `asin`, `atan2`,
`cosh`, `sinh`, `strtod`, `strtol`, `strtoul`

Remedies for this are:

- Do not use non-reentrant functions in interrupt service routines
- Guard calls to a non-reentrant function by a mutex, or a secure region, etc.

IAR DLIB Library

The IAR DLIB Library provides most of the important C and C++ library definitions that apply to embedded systems. These are of the following types:

- Adherence to a free-standing implementation of the ISO/ANSI standard for the programming language C. For additional information, see the chapter *Implementation-defined behavior* in this guide.
- Standard C library definitions, for user programs.
- Embedded C++ library definitions, for user programs.
- `CSTARTUP`, the module containing the start-up code. It is described in the *The DLIB runtime environment* chapter in this guide.
- Runtime support libraries; for example, low-level floating-point routines.
- Intrinsic functions, allowing low-level use of ARM features. See the chapter *Intrinsic functions* for more information.

C HEADER FILES

This section lists the header files specific to the DLIB library C definitions. Header files may additionally contain target-specific definitions; these are documented in the chapter *IAR language extensions*.

The following table lists the C header files:

Header file	Usage
<code>assert.h</code>	Enforcing assertions when functions execute
<code>ctype.h</code>	Classifying characters
<code>errno.h</code>	Testing error codes reported by library functions
<code>float.h</code>	Testing floating-point type properties
<code>iso646.h</code>	Using Amendment 1— <code>iso646.h</code> standard header
<code>limits.h</code>	Testing integer type properties
<code>locale.h</code>	Adapting to different cultural conventions
<code>math.h</code>	Computing common mathematical functions
<code>setjmp.h</code>	Executing non-local goto statements
<code>signal.h</code>	Controlling various exceptional conditions
<code>stdarg.h</code>	Accessing a varying number of arguments
<code>stdbool.h</code>	Adds support for the <code>bool</code> data type in C.
<code>stddef.h</code>	Defining several useful types and macros
<code>stdio.h</code>	Performing input and output
<code>stdlib.h</code>	Performing a variety of operations
<code>string.h</code>	Manipulating several kinds of strings
<code>time.h</code>	Converting between various time and date formats
<code>wchar.h</code>	Support for wide characters
<code>wctype.h</code>	Classifying wide characters

Table 42: Traditional standard C header files—DLIB

C++ HEADER FILES

This section lists the C++ header files.

Embedded C++

The following table lists the Embedded C++ header files:

Header file	Usage
complex	Defining a class that supports complex arithmetic
exception	Defining several functions that control exception handling
fstream	Defining several I/O streams classes that manipulate external files
iomanip	Declaring several I/O streams manipulators that take an argument
ios	Defining the class that serves as the base for many I/O streams classes
iosfwd	Declaring several I/O streams classes before they are necessarily defined
iostream	Declaring the I/O streams objects that manipulate the standard streams
istream	Defining the class that performs extractions
new	Declaring several functions that allocate and free storage
ostream	Defining the class that performs insertions
sstream	Defining several I/O streams classes that manipulate string containers
stdexcept	Defining several classes useful for reporting exceptions
streambuf	Defining classes that buffer I/O streams operations
string	Defining a class that implements a string container
strstream	Defining several I/O streams classes that manipulate in-memory character sequences

Table 43: Embedded C++ header files

The following table lists additional C++ header files:

Header file	Usage
fstream.h	Defining several I/O stream classes that manipulate external files
iomanip.h	Declaring several I/O streams manipulators that take an argument
iostream.h	Declaring the I/O streams objects that manipulate the standard streams
new.h	Declaring several functions that allocate and free storage

Table 44: Additional Embedded C++ header files—DLIB

Extended Embedded C++ standard template library

The following table lists the Extended Embedded C++ standard template library (STL) header files:

Header file	Description
algorithm	Defines several common operations on sequences

Table 45: Standard template library header files

Header file	Description
deque	A deque sequence container
functional	Defines several function objects
hash_map	A map associative container, based on a hash algorithm
hash_set	A set associative container, based on a hash algorithm
iterator	Defines common iterators, and operations on iterators
list	A doubly-linked list sequence container
map	A map associative container
memory	Defines facilities for managing memory
numeric	Performs generalized numeric operations on sequences
queue	A queue sequence container
set	A set associative container
slist	A singly-linked list sequence container
stack	A stack sequence container
utility	Defines several utility components
vector	A vector sequence container

Table 45: Standard template library header files (Continued)

Using standard C libraries in C++

The C++ library works in conjunction with 15 of the header files from the standard C library, sometimes with small alterations. The header files come in two forms—new and traditional—for example, `cassert` and `assert.h`.

The following table shows the new header files:

Header file	Usage
cassert	Enforcing assertions when functions execute
cctype	Classifying characters
cerrno	Testing error codes reported by library functions
cfloat	Testing floating-point type properties
climits	Testing integer type properties
locale	Adapting to different cultural conventions
cmath	Computing common mathematical functions
csetjmp	Executing non-local goto statements
csignal	Controlling various exceptional conditions

Table 46: New standard C header files—DLIB

Header file	Usage
<code>cstdarg</code>	Accessing a varying number of arguments
<code>cstdint</code>	Defining several useful types and macros
<code>cstdio</code>	Performing input and output
<code>cstdlib</code>	Performing a variety of operations
<code>cstring</code>	Manipulating several kinds of strings
<code>ctime</code>	Converting between various time and date formats

Table 46: New standard C header files—DLIB (Continued)

LIBRARY FUNCTIONS AS INTRINSIC FUNCTIONS

The following C library functions will under some circumstances be handled as intrinsic functions and will generate inline code instead of an ordinary function call:

```
memcpy
memset
strcat
strcmp
strcpy
strlen
```


Implementation-defined behavior

This chapter describes how the ARM IAR C/C++ Compiler handles the implementation-defined areas of the C language.

ISO 9899:1990, the International Organization for Standardization standard - *Programming Languages - C* (revision and redesign of ANSI X3.159-1989, American National Standard), changed by the ISO Amendment 1:1994, *Technical Corrigendum 1*, and *Technical Corrigendum 2*, contains an appendix called *Portability Issues*. The ISO appendix lists areas of the C language that ISO leaves open to each particular implementation.

Note: The ARM IAR C/C++ Compiler adheres to a freestanding implementation of the ISO standard for the C programming language. This means that parts of a standard library can be excluded in the implementation.

Descriptions of implementation-defined behavior

This section follows the same order as the ISO appendix. Each item covered includes references to the ISO chapter and section (in parenthesis) that explains the implementation-defined behavior.

TRANSLATION

Diagnostics (5.1.1.3)

Diagnostics are produced in the form:

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

where *filename* is the name of the source file in which the error was encountered, *linenumber* is the line number at which the compiler detected the error, *level* is the level of seriousness of the message (remark, warning, error, or fatal error), *tag* is a unique tag that identifies the message, and *message* is an explanatory message, possibly several lines.

ENVIRONMENT

Arguments to main (5.1.2.2.1)

The function called at program startup is called `main`. There is no prototype declared for `main`, and the only definition supported for `main` is:

```
int main(void)
```

To change this behavior, see *Customizing system initialization*, page 51.

Interactive devices (5.1.2.3)

The streams `stdin` and `stdout` are treated as interactive devices.

IDENTIFIERS

Significant characters without external linkage (6.1.2)

The number of significant initial characters in an identifier without external linkage is 200.

Significant characters with external linkage (6.1.2)

The number of significant initial characters in an identifier with external linkage is 200.

Case distinctions are significant (6.1.2)

Identifiers with external linkage are treated as case-sensitive.

CHARACTERS

Source and execution character sets (5.2.1)

The source character set is the set of legal characters that can appear in source files. The default source character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the source character set will be the host computer's default character set.

The execution character set is the set of legal characters that can appear in the execution environment. The default execution character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the execution character set will be the host computer's default character set. The runtime library needs a multibyte character scanner to support a multibyte execution character set.

See *Locale*, page 55.

Bits per character in execution character set (5.2.4.2.1)

The number of bits in a character is represented by the manifest constant `CHAR_BIT`. The standard include file `limits.h` defines `CHAR_BIT` as 8.

Mapping of characters (6.1.3.4)

The mapping of members of the source character set (in character and string literals) to members of the execution character set is made in a one-to-one way. In other words, the same representation value is used for each member in the character sets except for the escape sequences listed in the ISO standard.

Unrepresented character constants (6.1.3.4)

The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or in the extended character set for a wide character constant generates a diagnostic message, and will be truncated to fit the execution character set.

Character constant with more than one character (6.1.3.4)

An integer character constant that contains more than one character will be treated as an integer constant. The value will be calculated by treating the leftmost character as the most significant character, and the rightmost character as the least significant character, in an integer constant. A diagnostic message will be issued if the value cannot be represented in an integer constant.

A wide character constant that contains more than one multibyte character generates a diagnostic message.

Converting multibyte characters (6.1.3.4)

The only locale supported—that is, the only locale supplied with the IAR C/C++ Compiler—is the ‘C’ locale. If you use the command line option `--enable_multibytes`, the runtime library will support multibyte characters if you add a locale with multibyte support or a multibyte character scanner to the library.

See *Locale*, page 55.

Range of ‘plain’ char (6.2.1.1)

A ‘plain’ `char` has the same range as an unsigned `char`.

INTEGERS

Range of integer values (6.1.2.5)

The representation of integer values are in the two's complement form. The most significant bit holds the sign; 1 for negative, 0 for positive and zero.

See *Basic data types*, page 104, for information about the ranges for the different integer types: `char`, `short`, `int`, and `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. In other words, a large enough value will be converted into a negative value.

Signed bitwise operations (6.3)

Bitwise operations on signed integers work the same way as bitwise operations on unsigned integers; in other words, the sign-bit will be treated as any other bit.

Sign of the remainder on integer division (6.3.5)

The sign of the remainder on integer division is the same as the sign of the dividend.

Negative valued signed right shifts (6.3.7)

The result of a right-shift of a negative-valued signed integral type preserves the sign-bit. For example, shifting `0xFF00` down one step yields `0xFF80`.

FLOATING POINT

Representation of floating-point values (6.1.2.5)

The representation and sets of the various floating-point numbers adheres to IEEE 854–1987. A typical floating-point number is built up of a sign-bit (*s*), a biased exponent (*e*), and a mantissa (*m*).

See *Floating-point types*, page 105, 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 107, for information about *size_t*.

Conversion from/to pointers (6.3.4)

See *Casting*, page 107, for information about casting of data pointers and function pointers.

ptrdiff_t (6.3.6, 7.1.1)

See *ptrdiff_t*, page 108, for information about the *ptrdiff_t*.

REGISTERS**Honoring the register keyword (6.5.1)**

User requests for register variables are not honored.

STRUCTURES, UNIONS, ENUMERATIONS, AND BITFIELDS**Improper access to a union (6.3.2.3)**

If a union gets its value stored through a member and is then accessed using a member of a different type, the result is solely dependent on the internal storage of the first member.

Padding and alignment of structure members (6.5.2.1)

See the section *Basic data types*, page 104, for information about the alignment requirement for data objects.

Sign of 'plain' bitfields (6.5.2.1)

A 'plain' `int` bitfield is treated as a signed `int` bitfield. All integer types are allowed as bitfields.

Allocation order of bitfields within a unit (6.5.2.1)

Bitfields are allocated within an integer from least-significant to most-significant bit.

Can bitfields straddle a storage-unit boundary (6.5.2.1)

Bitfields cannot straddle a storage-unit boundary for the chosen bitfield integer type.

Integer type chosen to represent enumeration types (6.5.2.2)

The chosen integer type for a specific enumeration type depends on the enumeration constants defined for the enumeration type. The chosen integer type is the smallest possible.

QUALIFIERS

Access to volatile objects (6.5.3)

Any reference to an object with volatile qualified type is an access.

DECLARATORS

Maximum numbers of declarators (6.5.4)

The number of declarators is not limited. The number is limited only by the available memory.

STATEMENTS

Maximum number of case statements (6.6.4.2)

The number of case statements (case values) in a switch statement is not limited. The number is limited only by the available memory.

PREPROCESSING DIRECTIVES

Character constants and conditional inclusion (6.8.1)

The character set used in the preprocessor directives is the same as the execution character set. The preprocessor recognizes negative character values if a 'plain' character is treated as a signed character.

Including bracketed filenames (6.8.2)

For file specifications enclosed in angle brackets, the preprocessor does not search directories of the parent files. A parent file is the file that contains the `#include` directive. Instead, it begins by searching for the file in the directories specified on the compiler command line.

Including quoted filenames (6.8.2)

For file specifications enclosed in quotes, the preprocessor directory search begins with the directories of the parent file, then proceeds through the directories of any grandparent files. Thus, searching begins relative to the directory containing the source file currently being processed. If there is no grandparent file and the file has not been found, the search continues as if the filename was enclosed in angle brackets.

Character sequences (6.8.2)

Preprocessor directives use the source character set, with the exception of escape sequences. Thus, to specify a path for an include file, use only one backslash:

```
#include "mydirectory\myfile"
```

Within source code, two backslashes are necessary:

```
file = fopen("mydirectory\\myfile", "rt");
```

Recognized pragma directives (6.8.6)

The following pragma directives are recognized:

```
alignment
ARGUSED
baseaddr
bitfields
can_instantiate
codeseg
constseg
cspy_support
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
module_name
none
no_pch
NOTREACHED
object_attribute
once
optimize
pack
__printf_args
public_equ
rtmodel
__scanf_args
system_include
type_attribute
VARARGS
vector
warnings

```

For a description of the pragma directives, see the chapter *Pragma directives*.

Default `__DATE__` and `__TIME__` (6.8.8)

The definitions for `__TIME__` and `__DATE__` are always available.

IAR DLIB LIBRARY FUNCTIONS

The information in this section is valid only if the runtime library configuration you have chosen supports file descriptors. See the chapter *The DLIB runtime environment* for more information about runtime library configurations.

NULL macro (7.1.6)

The `NULL` macro is defined to 0.

Diagnostic printed by the `assert` function (7.2)

The `assert()` function prints:

```
filename:linenr expression -- assertion failed
```

when the parameter evaluates to zero.

Domain errors (7.5.1)

NaN (Not a Number) will be returned by the mathematic functions on domain errors.

Underflow of floating-point values sets `errno` to `ERANGE` (7.5.1)

The mathematics functions set the integer expression `errno` to `ERANGE` (a macro in `errno.h`) on underflow range errors.

`fmod()` functionality (7.5.6.4)

If the second argument to `fmod()` is zero, the function returns NaN; `errno` is set to `EDOM`.

`signal()` (7.7.1.1)

The signal part of the library is not supported.

Note: Low-level interface functions exist in the library, but will not perform anything. Use the template source code to implement application-specific signal handling. See *Signal and raise*, page 58.

Terminating newline character (7.9.2)

`stdout` stream functions recognize either `newline` or end of file (EOF) as the terminating character for a line.

Blank lines (7.9.2)

Space characters written to the `stdout` stream immediately before a newline character are preserved. There is no way to read the line through the `stdin` stream that was written through the `stdout` stream.

Null characters appended to data written to binary streams (7.9.2)

No null characters are appended to data written to binary streams.

Files (7.9.3)

Whether a write operation on a text stream causes the associated file to be truncated beyond that point, depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 54.

`remove()` (7.9.4.1)

The effect of a remove operation on an open file depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 54.

rename() (7.9.4.2)

The effect of renaming a file to an already existing filename depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 54.

%p in printf() (7.9.6.1)

The argument to a `%p` conversion specifier, print pointer, to `printf()` is treated as having the type `void *`. The value will be printed as a hexadecimal number, similar to using the `%x` conversion specifier.

%p in scanf() (7.9.6.2)

The `%p` conversion specifier, scan pointer, to `scanf()` reads a hexadecimal number and converts it into a value with the type `void *`.

Reading ranges in scanf() (7.9.6.2)

A - (dash) character is always treated as a range symbol.

File position errors (7.9.9.1, 7.9.9.4)

On file position errors, the functions `fgetpos` and `ftell` store `EFPOS` in `errno`.

Message generated by perror() (7.9.10.4)

The generated message is:

usersuppliedprefix: error message

Allocating zero bytes of memory (7.10.3)

The `calloc()`, `malloc()`, and `realloc()` functions accept zero as an argument. Memory will be allocated, a valid pointer to that memory is returned, and the memory block can be modified later by `realloc`.

Behavior of abort() (7.10.4.1)

The `abort()` function does not flush stream buffers, and it does not handle files, because this is an unsupported feature.

Behavior of exit() (7.10.4.3)

The argument passed to the `exit` function will be the return value returned by the `main` function to `cstartup`.

Environment (7.10.4.4)

The set of available environment names and the method for altering the environment list is described in *Environment interaction*, page 57.

system() (7.10.4.5)

How the command processor works depends on how you have implemented the `system` function. See *Environment interaction*, page 57.

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 <i>nnn</i>

Table 47: Message returned by `strerror()`—IAR DLIB library

The time zone (7.12.1)

The local time zone and daylight savings time implementation is described in *Time*, page 59.

clock() (7.12.2.1)

From where the system clock starts counting depends on how you have implemented the `clock` function. See *Time*, page 59.

IAR language extensions

This chapter describes IAR language extensions to the ISO/ANSI standard for the C programming language. All extensions can also be used for the C++ programming language.



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



See the compiler options `-e` on page 131 and `--strict_ansi` on page 144 for information about how to enable and disable language extensions from the command line.

Why should language extensions be used?

By using language extensions, you gain full control over the resources and features of the target core, and can thereby fine-tune your application.

If you want to use the source code with different compilers, note that language extensions may require minor modifications before the code can be compiled. A compiler typically supports core-specific language extensions as well as vendor-specific ones.

Descriptions of language extensions

This section gives an overview of available language extensions.

Memory, type, and object attributes

Entities such as variables and functions may be declared with memory, type, and object attributes. The syntax follows the syntax for qualifiers—such as `const`—but the semantics is different.

- A memory attribute controls the placement of the entity. There can be only one memory attribute.
- A type attribute controls aspects of the object visible to the surrounding context. There can be many different type attributes, and they must be included when the object is declared.
- An object attribute only has to be specified at the definition, but not at the declaration, of an object. The object attribute does not affect the object interface.

See the *Extended keywords* chapter for a complete list of attributes.

Placement at an absolute address or in a named segment

The @ operator or the directive #pragma location can be used for placing a variable at an absolute address, or placing a variable or function in a named segment. The named segment can either be a predefined segment, or a user-defined segment.

Note: Placing variables and functions into named segments can also be done using the command line option --segment.

Example 1

```
__no_init int x @ 0x1000;
```

An absolute declared variable cannot have an initializer, which means the variable must also be __no_init or const declared.

Example 2

```
void test(void) @ "MYOWNSEGMENT"
{
    ...
}
```

Note that all segments, both user-defined and predefined, must be assigned a location, which is done in the linker command file.

_Pragma

The preprocessor operator _Pragma can be used in defines and has the equivalent effect of the pragma directive. The syntax is:

```
_Pragma("string")
```

where *string* follows the syntax for the corresponding pragma directive. For example:

```
#if NO_OPTIMIZE
    #define NOOPT _Pragma("optimize=2")
#else
    #define NOOPT
#endif
```

See the chapter *Pragma directives*.

Variadic macros

Variadic macros are the preprocessor macro equivalents of printf style functions.

Syntax

```
#define P(...)      __VA_ARGS__
#define P(x,y,...)  x + y + __VA_ARGS__
```

Here, `__VA_ARGS__` will contain all variadic arguments concatenated, including the separating commas.

Example

```
#if DEBUG
#define DEBUG_TRACE(...) printf(S,__VA_ARGS__)
#else
#define DEBUG_TRACE(...)
#endif
...
DEBUG_TRACE("The value is:%d\n",value);
```

will result in:

```
printf("The value is:%d\n",value);
```

Inline assembler

Inline assembler can be used for inserting assembler instructions in the generated function.

The syntax for inline assembler is:

```
asm("SWI 5");
```

In strict ISO/ANSI mode, the use of inline assembler is disabled.

For more details about inline assembler, see *Mixing C and assembler*, page 67.

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, for example, four, it must be stored on an address that is divisible by four.

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

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

Another processor might only be able to read 2 bytes at a time; in that environment, the alignment for a 4-byte `long` integer might be 2.

A structure type will inherit the alignment from its components.

All objects must have a size that is a multiple of the alignment. Otherwise, only the first element of an array would be placed in accordance with the alignment requirements.

In the following example, the alignment of the structure is 4, under the assumption that `long` has alignment 4. Its size is 8, even though only 5 bytes are effectively used.

```
struct str {
    long a;
    char b;
};
```

In standard C, the size of an object can be accessed using the `sizeof` operator.

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 (that is, one without a name) defines an unnamed object (and not a type) whose members are promoted to the surrounding scope. External anonymous structure types are allowed.

For example, the structure `str` in the following example contains an anonymous union. The members of the union are accessed using the names `b` and `c`, for example `obj.b`.

Without anonymous structure types, the union would have to be named—for example `u`—and the member elements accessed using the syntax `obj.u.b`.

```
struct str
{
    int a;
    union
    {
        int b;
        int c;
    };
};

struct str obj;
```

Bitfields and non-standard types

In ISO/ANSI C, a bitfield must be of the type `int` or `unsigned int`. Using IAR language extensions, any integer types and enums may be used.

For example, in the following structure an `unsigned char` is used for holding three bits. The advantage is that the struct will be smaller.

```
struct str
{
    unsigned char bitOne    : 1;
    unsigned char bitTwo    : 1;
    unsigned char bitThree  : 1;
};
```

This matches G.5.8 in the appendix of the ISO standard, *ISO Portability Issues*.

Incomplete arrays at end of structs

The last element of a `struct` may be an incomplete array. This is useful because one chunk of memory can be allocated for the `struct` itself and for the array, regardless of the size of the array.

Note: The array may not be the only member of the `struct`. If that was the case, then the size of the `struct` would be zero, which is not allowed in ISO/ANSI C.

Example

```
struct str
{
    char a;
    unsigned long b[];
};
```

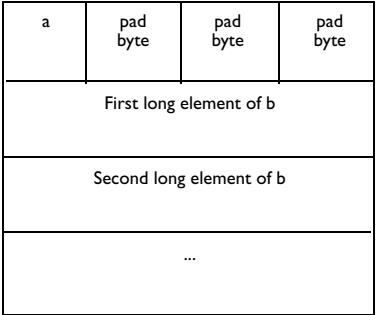
```
struct str * GetAStr(int size)
{
    return malloc(sizeof(struct str) +
                  sizeof(unsigned long) * size);
}

void UseStr(struct str * s)
{
    s->b[10] = 0;
}
```

The `struct` will inherit the alignment requirements from all elements, including the alignment of the incomplete array. The array itself will not be included in the size of the struct. However, the alignment requirements will ensure that the struct will end exactly at the beginning of the array; this is known as padding.

In the example, the alignment of `struct str` will be 4 and the size is also 4. (Assuming a processor where the alignment of `unsigned long` is 4.)

The memory layout of `struct str` is described in the following figure.



Arrays of incomplete types

An array may have an incomplete `struct`, `union`, or `enum` type as its element type. The types must be completed before the array is used (if it is), or by the end of the compilation unit (if it is not).

Empty translation units

A translation unit (source file) is allowed to be empty, that is, it does not have to contain any declarations.

In strict ISO/ANSI mode, a warning is issued if the translation unit is empty.

Example

The following source file is only used in a debug build. (In a debug, build the `NDEBUG` preprocessor flag is undefined.) Since the entire contents of the file is conditionally compiled using the preprocessor, the translation unit will be empty when the application is compiled in release mode. Without this extension, this would be considered an error.

```
#ifndef NDEBUG

void PrintStatusToTerminal()
{
    /* Do something */
}

#endif
```

Comments at the end of preprocessor directives

This extension, which makes it legal to place text after preprocessor directives, is enabled, unless strict ISO/ANSI mode is used. This language extension exists to support compilation of old legacy code; we do *not* recommend that you write new code in this fashion.

Example

```
#ifdef FOO

    ... something ...

#endif FOO /* This is allowed but not recommended. */
```

Forward declaration of enums

The IAR Systems language extensions allow that you first declare the name of an `enum` and later resolve it by specifying the brace-enclosed list.

Extra comma at end of enum list

It is allowed to place an extra comma at the end of an `enum` list. In strict ISO/ANSI mode, a warning is issued.

Note: ISO/ANSI C allows extra commas in similar situations, for example after the last element of the initializers to an array. The reason is, that it is easy to get the commas wrong if parts of the list are moved using a normal cut-and-paste operation.

Example

```
enum
{
    kOne,
    kTwo,    /* This is now allowed. */
};
```

Missing semicolon at end of struct or union specifier

A warning is issued if the semicolon at the end of a `struct` or `union` specifier is missing.

NULL and void

In operations on pointers, a pointer to `void` is always implicitly converted to another type if necessary, and a null pointer constant is always implicitly converted to a null pointer of the right type if necessary. In ISO/ANSI C, some operators allow such things, while others do not allow them.

A label preceding a "}"

In ISO/ANSI C, a label must be followed by at least one statement. Therefore, it is illegal to place the label at the end of a block. In the ARM IAR C/C++ Compiler, a warning is issued.

To create a standard-compliant C program (so that you will not have to see the warning), you can place an empty statement after the label. An empty statement is a single `;` (semi-colon).

Example

```
void test()
{
    if (...) goto end;

    /* Do something */

    end: /* Illegal at the end of block. */
}
```

Note: This also applies to the labels of `switch` statements.

The following piece of code will generate a warning:

```
switch (x)
{
case 1:
    ...;
    break;

default:
}
```

A good way to convert this into a standard-compliant C program is to place a `break;` statement after the `default:` label.

Empty declarations

An empty declaration (a semicolon by itself) is allowed, but a remark is issued (provided that remarks are enabled).

This is useful when preprocessor macros are used that could expand to nothing. Consider the following example. In a debug build, the macros `DEBUG_ENTER` and `DEBUG_LEAVE` could be defined to something useful. However, in a release build they could expand into nothing, leaving the `;` character in the code:

```
void test()
{
    DEBUG_ENTER();

    do_something();

    DEBUG_LEAVE();
}
```

Single value initialization

ISO/ANSI C requires that all initializer expressions of static arrays, `structs`, and `unions` are enclosed in braces.

Single-value initializers are allowed to appear without braces, but a warning is issued.

Example

In the ARM IAR C/C++ Compiler, the following expression is allowed:

```
struct str
{
    int a;
} x = 10;
```

Casting pointers to integers in static initializers

In an initializer, a pointer constant value may be cast to an integral type if the integral type is large enough to contain it.

In the example below, the first initialization is correct because it is possible to cast the 32-bit address to a 32-bit unsigned long variable.

However, it is illegal to use the 32-bit address of `a` as initializer for a 16-bit value.

```
int a;

unsigned long apl = (unsigned long)&a;    /* correct */
unsigned short aps = (unsigned short)&a;  /* illegal */
```

Hexadecimal floating-point constants

Floating-point constants can be given in hexadecimal style. The syntax is `0xMANTp{+|-}EXP`, where *MANT* is the mantissa in hexadecimal digits, including an optional `.` (decimal point), and *EXP* is the exponent with decimal digits, representing an exponent of 2.

Examples

```
0x1p0 is 1
0xA.8p2 is 10.5*2^2
```

Using the `bool` data type in C

To use the `bool` type in C source code, you must include the `stdbool.h` file. (The `bool` data type is supported by default in C++.)

Taking the address of a register variable

In ISO/ANSI C, it is illegal to take the address of a variable specified as a register variable.

The ARM IAR C/C++ 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 a synonym for `double`.

Repeated typedefs

Redeclarations of `typedef` that occur in the same scope are allowed, but a warning is issued.

Mixing pointer types

Assignment and pointer difference is allowed between pointers to types that are interchangeable but not identical; for example, `unsigned char *` and `char *`. This includes pointers to integral types of the same size. A warning is issued.

Assignment of a string constant to a pointer to any kind of character is allowed, and no warning will be issued.

Non-top level const

Assignment of pointers is allowed in cases where the destination type has added type qualifiers that are not at the top level (for example, `int **` to `int const **`). It is also allowed to compare and take the difference of such pointers.

Declarations in other scopes

External and static declarations in other scopes are visible. In the following example, the variable `y` can be used at the end of the function, even though it should only be visible in the body of the `if` statement. A warning is issued.

```
int test(int x)
{
    if (x)
    {
        extern int y;
        y = 1;
    }

    return y;
}
```

Non-lvalue arrays

A non-lvalue array expression is converted to a pointer to the first element of the array when it is used.

Diagnostics

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

Message format

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

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

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

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

Use the option `--diagnostics_tables` to list all possible compiler diagnostic messages in a named file.

Severity levels

The diagnostics are divided into different levels of severity:

Remark

A diagnostic message that is produced when the compiler finds a source code construct that can possibly lead to erroneous behavior in the generated code. Remarks are by default not issued, but can be enabled, see `--remarks`, page 142.

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 140.

Error

A diagnostic that is produced when the compiler has found a construct which clearly violates the C or C++ language rules, such that code cannot be produced. An error will produce a non-zero exit code.

Fatal error

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

SETTING THE SEVERITY LEVEL

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

See *Options summary*, page 123, for a description of the compiler options that are available for setting severity levels.

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

INTERNAL ERROR

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

Internal error: *message*

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

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

Index

A

- absolute location 33
 - #pragma location 157
- absolute placement 196
- algorithm (STL header file) 180
- alignment 103
 - forcing stricter 154
- anonymous structures 94
- applications
 - building 4
 - initializing 50
 - terminating 50
- architecture, ARM xv
- ARGFRAME (compiler function directive) 22
- ARM
 - architecture xv
 - instruction set. xv
- Arm and Thumb code
 - overview 15
- ARM (CPU mode) 6
 - __ARMVFP__ (predefined symbol) 166
 - __ARM4TM__ (predefined symbol) 166
 - __ARM5__ (predefined symbol) 166
 - __ARM5E__ (predefined symbol) 166
 - __ARM5T__ (predefined symbol) 166
 - __ARM5TM__ (predefined symbol) 166
- arrays
 - implementation-defined behavior. 187
- asm (extended keyword) 148
- asm (inline assembler) 69
- assembler directives
 - CFI 81
 - CODE16 69
 - CODE32 69
 - DC32 69

- ENDMOD 64
- EQU 142
- MODULE 64
- PUBLIC 142
- REQUIRE 64
- RSEG 64
- RTMODEL 62
- assembler instructions
 - SWI 19
- assembler language interface 67
 - creating skeleton code 70
- assembler list file 22
- assembler routines, calling from C 70
- assembler, inline 69, 96, 197
- assert.h (library header file) 179
- assumptions (programming experience) xv
- atomic operations, performing 149
- attributes 110
- auto variables 11–12
 - saving stack space 96

B

- big endian (byte order) 7
- bitfields
 - data representation 105
 - hints 93
 - implementation-defined behavior. 187
- bitfields (pragma directive) 105, 154
- bool (data type) 104
 - supported in C code 104
- bool (data type), adding support for 179
- byte order 103
 - big-endian 7
 - little-endian 7
 - specifying 132

C

<code>__CORE__</code> (predefined symbol)	166
<code>__CPU_MODE__</code> (predefined symbol)	167
C and C++ linkage	73
C calling convention	73
C header files	179
call chains	96
call stack	81
callee-save registers, stored on stack	12
calling convention	
C	73
C++	72
calloc (standard library function)	13
cassert (library header file)	181
cast operators	
in Extended EC++	84
missing from Embedded C++	84
casting, of pointers and integers	107
cctype (library header file)	181
cerrno (library header file)	181
CFI (assembler directive)	81
cfloat (library header file)	181
char (data type), signed and unsigned	105, 126
characters, implementation-defined behavior	184
<code>--char_is_signed</code> (compiler option)	126
classes	85
CLIB	
documentation	177
header files	177
library object files	177
climits (library header file)	181
clocale (library header file)	181
<code>__close</code> (library function)	55
cmath (library header file)	181
code	
ARM and Thumb, overview	15
excluding when linking	64
placement of	113

placement of startup code	31
Code motion (compiler option)	90
code motion, disabling	137
code pointers	107
CODE (segment)	32, 114
CODE_I (segment)	114
CODE_ID (segment)	115
CODE16 (assembler directive)	69
CODE32 (assembler directive)	69
common sub-expression elimination, disabling	138
Common-subexpr elimination (compiler option)	89
compiler environment variables	123
compiler error return codes	123
compiler listing, generating	134
compiler object file	
including debug information	127, 142
specifying filename	140
compiler options	
Code motion	90
Common-subexpr elimination	89
Function inlining	90
Instruction scheduling	92
Loop unrolling	89
setting	121
specifying parameters	122
Static clustering	92
summary	123
Type-based alias analysis	90
typographic convention	xviii
<code>-D</code>	126
<code>-e</code>	131
<code>-f</code>	132
<code>-I</code>	133
<code>-l</code>	71, 134
<code>-o</code>	140
<code>-r</code>	127, 142
<code>-s</code>	143
<code>-z</code>	145
<code>--char_is_signed</code>	126

- cpu 126
- cpu_mode 126
- debug. 127, 142
- dependencies 128
- diagnostics_tables 130
- diag_error 129
- diag_remark 129
- diag_suppress 130
- diag_warning 130
- ec++ 131
- eec++ 131
- enable_multibytes 131
- endian 132
- fpu 133
- header_context 133
- interwork 134
- library_module 136
- migration_preprocessor_extensions 136
- module_name 136
- no_clustering 137
- no_code_motion 137
- no_cse 138
- no_inline 138–139
- no_scheduling 138
- no_tbaa 139
- no_typedefs_in_diagnostics 139
- no_unroll 139
- no_warnings 140
- no_wrap_diagnostics 140
- omit_types 141
- only_stdout 141
- preinclude 141
- preprocess 141
- public_equ 142
- remarks 142
- require_prototypes 142
- segment 143
- separate_cluster_for_initialized_variables 144
- silent 144
- stack_align 144
- strict_ansi 144
- warnings_affect_exit_code 123, 144
- warnings_are_errors 145
- compiler version number 169
- compiling, from the command line 4
- complex numbers, supported in Embedded C++ 84
- complex (library header file) 180
- computer style, typographic convention xviii
- configuration
 - basic project settings 5
- configuration symbols, in library configuration files 47
- consistency, module 61
- const_cast() (cast operator) 84
- conventions, typographic xviii
- copy initialization, of segments 32
- copyright notice ii
- cpu (compiler option) 126
- CPU modes 6
- CPU variant, specifying on command line 126
- cpu_mode (compiler option) 126
- _cpu_mode (runtime model attribute) 63
- csetjmp (library header file) 181
- csignal (library header file) 181
- CSTACK (segment) 115
- example 29
- See also* stack
- cstartup, customizing 51
- cstartup, implementation 63
- cstdarg (library header file) 182
- cstddef (library header file) 182
- cstdio (library header file) 182
- cstdlib (library header file) 182
- cstring (library header file) 182
- ctime (library header file) 182
- ctype.h (library header file) 179
- customization
 - _low_level_init 51

C++	
calling convention	72
features excluded from EC++	83
<i>See also</i> Embedded C++ and Extended Embedded C++	
terminology	xviii
C++ header files	179–180
C-SPY	
STL container support	86
C-SPY, low-level interface	60
C_INCLUDE (environment variable)	123, 133

D

data	
alignment of	103
excluding when linking	64
placement of	113
data pointers	107
data representation	103
data storage	11
data types	104
floating point	105
integers	104
DATA_AC (segment)	116
data_alignment (pragma directive)	154
DATA_AN (segment)	117
DATA_C (segment)	117
DATA_I (segment)	117
DATA_ID (segment)	118
DATA_N (segment)	118
DATA_Z (segment)	118
__DATE__ (predefined symbol)	167
date (library function), configuring support for	59
DC32 (assembler directive)	69
--debug (compiler option)	127, 142
debug information, including in object file	127, 142
declaration, of functions	73
declarators, implementation-defined behavior	188
delete (keyword)	13

--dependencies (compiler option)	128
deque (STL header file)	181
diagnostic messages	207
classifying as errors	129
classifying as remarks	129
classifying as warnings	130
disabling warnings	140
disabling wrapping of	140
enabling remarks	142
listing all used	130
suppressing	130
--diagnostics_tables (compiler option)	130
diag_default (pragma directive)	155
--diag_error (compiler option)	129
diag_error (pragma directive)	155
--diag_remark (compiler option)	129
diag_remark (pragma directive)	155
--diag_suppress (compiler option)	130
diag_suppress (pragma directive)	155
--diag_warning (compiler option)	130
diag_warning (pragma directive)	156
DIFUNCT (segment)	33
directives	
function	22
pragma	9, 153
disclaimer	ii
DLIB	8, 178
documentation	177
document conventions	xviii
documentation, library	177
double (data type)	105
dynamic initialization	44–45, 49
in Embedded C++	33
dynamic memory	13

E

--ec++ (compiler option)	131
EC++ header files	180

- eec++ (compiler option) 131
- Embedded C++. 83
 - absolute location 33
 - declaring functions. 22
 - differences from C++. 83
 - dynamic initialization in 33
 - enabling 131
 - function linkage 73
 - language extensions 83
 - overview 83
 - static member variables 33
- enable_multibytes (compiler option) 131
- endian (compiler option) 132
- __endian (runtime model attribute) 63
- ENDMOD (assembler directive). 64
- enumerations, implementation-defined behavior. 187
- enum, data representation 104
- environment
 - implementation-defined behavior. 184
- environment variables. 123
 - C_INCLUDE 123, 133
 - QCCARM 123
- EQU (assembler directive) 142
- errno.h (library header file). 179
- error messages 208
 - classifying 129
- error return codes 123
- exception handling, missing from Embedded C++ 83
- exception stacks 30
- exception vectors 32
- exception (library header file). 180
- experience, programming xv
- export keyword, missing from Extended EC++ 85
- Extended Embedded C++. 84
 - enabling 131
 - standard template library (STL). 180
- extended keywords 147
 - asm 148
 - enabling 131

- functions 15
- overview 9
- summary 147
- using 147
 - __arm 148
 - __fiq 149
 - __interwork. 149
 - __intrinsic 149
 - __irq 149
 - using in pragma directives. 163
 - __monitor 149
 - using in pragma directives. 163
 - __nested 150
 - __no_init. 99, 150
 - __ramfunc. 16, 150
 - __root 150
 - __swi 151
 - __thumb 151

F

- f (compiler option). 132
- fast interrupts 17
- fatal error messages 208
- __FILE__ (predefined symbol). 167
- file dependencies, tracking 128
- file paths, specifying for #include files 133
- filename, of object file 140
- float (data type). 105
- floating-point constants
 - hexadecimal notation 204
 - hints 93
- floating-point format. 105
 - hints 93
 - implementation-defined behavior. 186
 - special cases. 106
 - 32-bits 106
 - 64-bits 106
- floating-point unit. 133

float.h (library header file)	179
formats	
floating-point values	105
standard IEEE (floating point)	105
_formatted_write (library function)	43
--fpu (compiler option)	133
fragmentation, of heap memory	13
free (standard library function)	13
fstream (library header file)	180
fstream.h (library header file)	180
FUNCALL (compiler function directive)	22
function directives	22
function execution, in RAM	16
Function inlining (compiler option)	90
function inlining, disabling	138–139
function prototypes	96
function type information	
omitting in object output	141
FUNCTION (compiler function directive)	22
functional (STL header file)	181
functions	
declaring	73
executing	11
extended keywords	15
interrupt	21
intrinsic	67, 96
monitor	21
omitting type info	141
overview	15
parameters	75
placing in segments	34
recursive	96
storing data on stack	12–13
reentrancy (DLIB)	178
return values from	77

G

getenv (library function), configuring support for	57
--	----

getzone (library function), configuring support for	59
glossary	xv
guidelines, reading	xv

H

hash_map (STL header file)	181
hash_set (STL header file)	181
header files	
C	179
CLIB	177
C++	179–180
EC++	180
special function registers	98
stdbool.h	104, 179
stddef.h	105
STL	180
using as templates	99
--header_context (compiler option)	133
heap	13
changing default size	31
size	30–31
storing data	11
HEAP (segment)	30, 116
hidden parameters	75
hints	
optimization	95

I

-I (compiler option)	133
IAR Technical Support	208
_IAR_SYSTEMS_ICC_ (predefined symbol)	167
ICCARM (predefined symbol)	167
ICODE (segment)	31, 119
identifiers, implementation-defined behavior	184
IEEE format, floating-point values	105
implementation, cstartup	63
implementation-defined behavior	183

- inarm.h (header file) 171
- inheritance, in Embedded C++ 83
- initialization
 - dynamic 45
- initialization, dynamic 44, 49
- INITTAB (segment) 119
- inline assembler 69, 96, 197
 - See also* assembler language interface
- inline (pragma directive) 156
- inlining of functions, in compiler 90
- Instruction scheduling (compiler option) 92
- instruction set, ARM xv
- int (data type) 104
- integers 104
 - casting 107
 - implementation-defined behavior 186
 - intptr_t 108
 - ptrdiff_t 108
 - size_t 107
 - uintptr_t 108
- internal error 208
- interrupt functions 17
 - fast interrupts 17
 - installing 20
 - installing in vector table 20
 - INTVEC segment 119
 - nested interrupts 18
 - operations 20
 - placement in memory 32
 - software interrupts 19
- interrupt handler 19
- interrupt vectors, specifying with pragma directive 163
- interrupts
 - disabling 149
 - disabling during function execution 21
 - processor state 12
- interworking code 6
 - generating 134
- intptr_t (integer type) 108
- __intrinsic (extended keyword) 149
- intrinsic functions 96
 - overview 67
 - summary 171
- __CLZ 172
- __disable_interrupt 172
- __enable_interrupt 172
- __get_CPSR 172
- __MCR 172
- __MRC 173
- __no_operation 173
- __QADD 174
- __QDADD 174
- __QDSUB 175
- __QSUB 175
- __segment_begin 174
- __segment_end 174
- __segment_size 174
- __set_CPSR 173
- __sfb 174
- __sfe 174
- __sfs 174
- INTVEC (segment) 32, 119
- iomanip (library header file) 180
- iomanip.h (library header file) 180
- ios (library header file) 180
- iosfwd (library header file) 180
- iostream (library header file) 180
- iostream.h (library header file) 180
- __irq (extended keyword)
 - using in pragma directives 163
- IRQ_STACK (segment) 120
- ISO/ANSI C
 - C++ features excluded from EC++ 83
 - language extensions 195
 - specifying strict usage 144
- iso646.h (library header file) 179
- istream (library header file) 180
- iterator (STL header file) 181

K

keywords, extended. 9, 147

L

-l (compiler option)	71, 134
__LITTLE_ENDIAN__ (predefined symbol)	167
language extensions	
descriptions	195
Embedded C++	83
enabling	131
language (pragma directive)	157
libraries.	4
runtime.	40
standard template library	180
library configuration file, modifying.	48
library documentation.	177
library features, missing from Embedded C++	84
library functions	177
choosing printf formatter	43
choosing scanf formatter	44
choosing sprintf formatter	43
choosing sscanf formatter	44
remove	55
rename	55
summary	179
__close	55
__lseek	55
__open	55
__read	55
__write	55
library modules, creating	136
library object files	
CLIB	177
--library_module (compiler option)	136
limits.h (library header file)	179
__LINE__ (predefined symbol)	167
linkage, C and C++	73

linker command files	
contents	24
customizing	25, 27–32
introduction	24
template	25
using the -Z command	26
viewing default	29
linking, from the command line	4
list (STL header file)	181
listing, generating	134
literature, recommended	xvii
little endian (byte order)	7
locale.h (library header file)	179
location (pragma directive)	33–34, 157
LOCFRAME (compiler function directive)	22
long (data type)	104
Loop unrolling (compiler option)	89
loop unrolling, disabling	139
loop-invariant expressions	90
low-level processor operations	171
__low_level_init, customizing	51
__lseek (library function)	55

M

macros	
variadic	196
malloc (standard library function)	13
map (STL header file)	181
math.h (library header file)	179
memory	
allocating in Embedded C++	13
dynamic	13
heap	13
non-initialized	99
RAM, saving	96
releasing in Embedded C++	13
stack.	11
stack, saving.	96

- static 11
- used by executing functions 11
- used by global or static variables 11
- memory management, type-safe 83
- memory (STL header file) 181
- message (pragma directive) 157
- migration_preprocessor_extensions (compiler option) .. 136
- module consistency 61
- rtmodel 161
- module name, specifying 136
- MODULE (assembler directive) 64
- modules, assembler 64
- module_name (compiler option) 136
- __monitor (extended keyword) 149
- using in pragma directives 163
- monitor functions 21, 149
- multibyte character support 131
- multiple inheritance, missing from Embedded C++ 83
- mutable attribute, in Extended EC++ 86

N

- namespace support
 - in Extended EC++ 84, 86
 - missing from Embedded C++ 84
- name, specifying for object file 140
- __nested (extended keyword) 150
- nested interrupts 18
- new (keyword) 13
- new (library header file) 180
- new.h (library header file) 180
- non-initialized variables 99
- non-scalar parameters 96
- no_code_motion (compiler option) 137
- no_cse (compiler option) 138
- __no_init (extended keyword) 99, 150
- no_inline (compiler option) 138–139
- (compiler option) 139
- no_unroll (compiler option) 139

- no_warnings (compiler option) 140
- no_wrap_diagnostics (compiler option) 140
- numeric (STL header file) 181

O

- o (compiler option) 140
- object filename, specifying 140
- object module name, specifying 136
- object_attribute (pragma directive) 99, 157
- omit_types (compiler option) 141
- only_stdout (compiler option) 141
- _open (library function) 55
- operators
 - @ 33–34
- optimization
 - code motion, disabling 137
 - common sub-expression elimination, disabling 138
 - configuration 7
 - function inlining, disabling 138–139
 - hints 95
 - loop unrolling, disabling 139
 - size, specifying 145
 - speed, specifying 143
 - types and levels 88
- optimization techniques
 - code motion 90
 - common-subexpression elimination 89
 - function inlining 90
 - instruction scheduling 92
 - loop unrolling 89
 - static clustering 92
 - type-based alias analysis 90
- optimize (pragma directive) 158
- options summary, compiler 123
- ostream (library header file) 180
- output
 - specifying 5
 - specifying file name 5

output files, from XLINK	5
output, preprocessor	141

P

pack (pragma directive)	109, 159
packed structure types	109
parameters	
function	75
hidden	75
non-scalar	96
register	75
specifying	122
stack	75–76
typographic convention	xviii
permanent registers	74
placement of code and data	113
pointers	
casting	107
code	107
data	107
implementation-defined behavior	187
using instead of large non-scalar parameters	96
polymorphism, in Embedded C++	83
_Pragma (preprocessor operator)	196
pragma directives	9
bitfields	105, 154
data_alignment	154
diag_default	155
diag_error	155
diag_remark	155
diag_suppress	155
diag_warning	156
inline	156
language	157
location	33–34, 157
message	157
object_attribute	99, 157
optimize	158

pack	109, 159
required	161
rtmodel	161
segment	162
summary	153
swi_number	162
syntax	154
type_attribute	162
vector	163
predefined symbols	
overview	10
summary	165
ARMVFP	166
ARM4TM	166
ARM5E	166
ARM5TM	166
ARM5T	166
ARM5	166
CORE	166
_cplusplus	166
_CPU_MODE_	167
DATE	167
_embedded_cplusplus	167
FILE	167
_IAR_SYSTEMS_ICC_	167
ICCARM	167
LINE	167
_LITTLE_ENDIAN_	167
STDC	167
_STDC_VERSION_	168
TID	168
TIME	168
VER	169
--preinclude (compiler option)	141
--preprocess (compiler option)	141
preprocessing directives, implementation-defined behavior	188
preprocessor output	141
preprocessor symbols, defining	126
preprocessor, extending	136

prerequisites (programming experience) xv
 print formatter, selecting 44
 printf (library function) 43
 printf (library function), choosing formatter 43
 processor operations, low-level 171
 processor variant, specifying on command line 126
 programming experience, required xv
 programming hints 95
 ptrdiff_t (integer type) 108
 PUBLIC (assembler directive) 142
 --public_equ (compiler option) 142

Q

QCCARM (environment variable) 123
 qualifiers, implementation-defined behavior 188
 queue (STL header file) 181

R

-r (compiler option) 127, 142
 raise (library function), configuring support for 58
 RAM execution 16
 RAM memory, saving 96
 _read (library function) 55
 read formatter, selecting 45
 reading guidelines xv
 reading, recommended xvii
 realloc (standard library function) 13
 recursive functions 96
 storing data on stack 12–13
 reentrancy (DLIB) 178
 reference information, typographic convention xviii
 register parameters 75
 registered trademarks ii
 registers
 assigning to parameters 75
 callee-save, stored on stack 12
 implementation-defined behavior 187

 permanent 74
 scratch 74
 reinterpret_cast() (cast operator) 84
 remark (diagnostic message)
 classifying 129
 enabling 142
 --remarks (compiler option) 142
 remarks (diagnostic message) 207
 remove (library function) 55
 rename (library function) 55
 REQUIRE (assembler directive) 64
 required (pragma directive) 161
 --require_prototypes (compiler option) 142
 return values, from functions 77
 _root (extended keyword) 150
 routines, time-critical 67, 171
 RSEG (assembler directive) 64
 RTMODEL (assembler directive) 62
 rtmodel (pragma directive) 161
 rtti support, missing from STL 84
 _rt_version (runtime model attribute) 63
 runtime libraries 40
 introduction 177
 naming convention 41
 summary 40
 runtime model attributes 61
 StackAlign4 63
 StackAlign8 63
 __cpu_mode 63
 __endian 63
 __rt_version 63
 runtime type information, missing from Embedded C++ . . 83

S

-s (compiler option) 143
 scanf (library function), choosing formatter 44
 scatter loading 32
 scratch registers 74

segment memory types, in XLINK	24
segment parts, unused	64
segment (pragma directive)	162
segments	113
CODE	32, 114
CODE_I	114
CODE_ID	115
CSTACK	115
example	29
DATA_AC	116
DATA_AN	117
DATA_C	117
DATA_I	117
DATA_ID	118
DATA_N	118
DATA_Z	118
DIFUNCT	33
HEAP	30, 116
ICODE	31, 119
INITTAB	119
introduction	23
INTVEC	32, 119
IRQ_STACK	120
summary	113
SWITAB	120
__segment_begin (intrinsic function)	174
__segment_end (intrinsic function)	174
semaphores, operations on	149
set (STL header file)	181
setjmp.h (library header file)	179
settings, basic for project configuration	5
severity level, of diagnostic messages	207
specifying	208
__sfb (intrinsic function)	174
__sfe (intrinsic function)	174
SFR (special function registers)	98
declaring extern	35
short (data type)	104
signal (library function), configuring support for	58
signal.h (library header file)	179
signed char (data type)	104–105
specifying	126
signed int (data type)	104
signed long (data type)	104
signed short (data type)	104
--silent (compiler option)	144
silent operation, specifying	144
64-bits (floating-point format)	106
size optimization, specifying	145
size_t (integer type)	107
skeleton code, creating for assembler language interface	70
slist (STL header file)	181
software interrupt handler	19
software interrupts	19
source files, list all referred	133
special function registers (SFR)	98
special function types	
overview	10
speed optimization, specifying	143
sprintf (library function)	43
sprintf (library function), choosing formatter	43
sscanf (library function), choosing formatter	44
sstream (library header file)	180
stack	11
advantages and problems using	12
contents of	12
exception	30
function usage	11
internal data	115
saving space	96
size	30
stack alignment	
specifying	144
stack parameters	75–76
stack pointer	12
stack (STL header file)	181
StackAlign4 (runtime model attribute)	63
StackAlign8 (runtime model attribute)	63

standard error 141
 standard output, specifying 141
 standard template library (STL)
 in Extended EC++ 84, 86, 180
 missing from Embedded C++ 84
 startup code
 placement of 31
 See also ICODE
 startup, system 50
 statements, implementation-defined behavior 188
 Static clustering (compiler option) 92
 static memory 11
 static overlay 22
 static_cast() (cast operator) 84
 std namespace, missing from EC++
 and Extended EC++ 86
 stdarg.h (library header file) 179
 stdbool.h (library header file) 104, 179
 __STDC__ (predefined symbol) 167
 __STDC_VERSION__ (predefined symbol) 168
 stddef.h (library header file) 105, 179
 stderr 55, 141
 stdexcept (library header file) 180
 stdin 55
 stdio.h (library header file) 179
 stdlib.h (library header file) 179
 stdout 55, 141
 streambuf (library header file) 180
 streams, supported in Embedded C++ 84
 --strict_ansi (compiler option) 144
 string (library header file) 180
 strings, supported in Embedded C++ 84
 string.h (library header file) 179
 strstream (library header file) 180
 strtod (library function), configuring support for 59
 structure types
 alignment 108–109
 layout 108
 packed 109

structures
 anonymous 94
 implementation-defined behavior 187
 Support, Technical 208
 SWITAB (segment) 120
 swi_number (pragma directive) 162
 symbols
 predefined
 overview of 10
 preprocessor, defining 126
 system startup 50
 system termination 50
 system (library function), configuring support for 57

T

Technical Support, IAR 208
 template support
 in Extended EC++ 84–85
 missing from Embedded C++ 83
 termination, system 50
 terminology xv, xviii
 32-bits (floating-point format) 106
 this (pointer) 72
 Thumb (CPU mode) 6
 __TID__ (predefined symbol) 168
 __TIME__ (predefined symbol) 168
 time (library function), configuring support for 59
 time-critical routines 67, 171
 time.h (library header file) 179
 tips, programming 95
 trademarks ii
 translation, implementation-defined behavior 183
 trap vectors, specifying with pragma directive 163
 type information, omitting 141
 Type-based alias analysis (compiler option) 90
 type-safe memory management 83
 type_attribute (pragma directive) 162
 typographic conventions xviii

U

uintptr_t (integer type)	108
unions	
anonymous	94
implementation-defined behavior	187
unsigned char (data type)	104–105
changing to signed char	126
unsigned int (data type)	104
unsigned short (data type)	104
utility (STL header file)	181

V

variable type information	
omitting in object output	141
variables	
auto	11–12, 96
defined inside a function	11
global	
placement in memory	11
local. <i>See</i> auto variables	
non-initialized	99
omitting type info	141
placing at absolute addresses	33
placing in named segments	33
placing in segments	34
static, placement in memory	11
vector floating-point unit	133
vector table	20
vector (pragma directive)	163
vector (STL header file)	181
__VER__ (predefined symbol)	169
version, of compiler	169
VFP	133

W

warnings	207
--------------------	-----

classifying	130
disabling	140
exit code.	144
--warnings_affect_exit_code (compiler option)	123
--warnings_are_errors (compiler option)	145
wchar.h (library header file)	179
wchar_t (data type)	
adding support for in C	105
wctype.h (library header file)	179
__write (library function).	55

X

XLINK options	
-Q.	32
XLINK output files.	5
XLINK segment memory types	24

Z

-z (compiler option)	145
--------------------------------	-----

Symbols

#include files, specifying	133
#pragma directives	
location	34
-D (compiler option)	126
-e (compiler option)	131
-f (compiler option).	132
-I (compiler option).	133
-l (compiler option).	71, 134
-o (compiler option)	140
-Q (XLINK option).	32
-r (compiler option).	127, 142
-s (compiler option)	143
-z (compiler option)	145
--char_is_signed (compiler option).	126
--cpu (compiler option).	126

- cpu_mode (compiler option) 126
- debug (compiler option) 127, 142
- dependencies (compiler option) 128
- diagnostics_tables (compiler option) 130
- diag_error (compiler option) 129
- diag_remark (compiler option) 129
- diag_suppress (compiler option) 130
- diag_warning (compiler option) 130
- ec++ (compiler option) 131
- eec++ (compiler option) 131
- enable_multibytes (compiler option) 131
- endian (compiler option) 132
- fpu (compiler option) 133
- header_context (compiler option) 133
- interwork (compiler option) 134
- library_module (compiler option) 136
- migration_preprocessor_extensions (compiler option) 136
- module_name (compiler option) 136
- no_clustering (compiler option) 137
- no_code_motion (compiler option) 137
- no_cse (compiler option) 138
- no_inline (compiler option) 138–139
- no_scheduling (compiler option) 138
- no_tbaa (compiler option) 139
- no_typedefs_in_diagnostics (compiler option) 139
- no_unroll (compiler option) 139
- no_warnings (compiler option) 140
- no_wrap_diagnostics (compiler option) 140
- omit_types (compiler option) 141
- only_stdout (compiler option) 141
- preinclude (compiler option) 141
- preprocess (compiler option) 141
- remarks (compiler option) 142
- require_prototypes (compiler option) 142
- segment (compiler option) 143
- separate_cluster_for_initialized_variables
 (compiler option) 144
- silent (compiler option) 144
- stack_align (compiler option) 144
- strict_ansi (compiler option) 144
- warnings_affect_exit_code (compiler option) 123, 144
- warnings_are_errors (compiler option) 145
- @ (operator) 33–34
- _arm (extended keyword) 148
- _ARMVFP_ (predefined symbol) 166
- _ARM4TM_ (predefined symbol) 166
- _ARM5E_ (predefined symbol) 166
- _ARM5TM_ (predefined symbol) 166
- _ARM5T_ (predefined symbol) 166
- _ARM5_ (predefined symbol) 166
- _close (library function) 55
- _CLZ (intrinsic function) 172
- _CORE_ (predefined symbol) 166
- _cplusplus (predefined symbol) 166
- _cpu_mode (runtime model attribute) 63
- _CPU_MODE_ (predefined symbol) 167
- _DATE_ (predefined symbol) 167
- _disable_interrupt (intrinsic function) 172
- _embedded_cplusplus (predefined symbol) 167
- _enable_interrupt (intrinsic function) 172
- _endian (runtime model attribute) 63
- _FILE_ (predefined symbol) 167
- _fiq (extended keyword) 149
- _get_CPSR (intrinsic function) 172
- _IAR_SYSTEMS_ICC_ (predefined symbol) 167
- _ICCARM_ (predefined symbol) 167
- _interwork (extended keyword) 149
- _intrinsic (extended keyword) 149
- _irq (extended keyword) 149
 - using in pragma directives 163
- _LINE_ (predefined symbol) 167
- _LITTLE_ENDIAN_ (predefined symbol) 167
- _low_level_init, customizing 51
- _lseek (library function) 55
- _MCR (intrinsic function) 172
- _monitor (extended keyword) 149
 - using in pragma directives 163
- _MRC (intrinsic function) 173

__nested (extended keyword)	150
__no_init (extended keyword)	99, 150
__no_operation (intrinsic function)	173
__open (library function)	55
__QADD (intrinsic function)	174
__QDADD (intrinsic function)	174
__QDSUB (intrinsic function)	175
__QSUB (intrinsic function)	175
__ramfunc (extended keyword)	16, 150
__read (library function)	55
__root (extended keyword)	150
__rt_version (runtime model attribute)	63
__segment_begin (intrinsic function)	174
__segment_end (intrinsic function)	174
__segment_size (intrinsic function)	174
__set_CPSR (intrinsic function)	173
__sfb (intrinsic function)	174
__sfe (intrinsic function)	174
__sfs (intrinsic function)	174
__STDC__ (predefined symbol)	167
__STDC_VERSION__ (predefined symbol)	168
__swi (extended keyword)	151
__thumb (extended keyword)	151
__TID__ (predefined symbol)	168
__TIME__ (predefined symbol)	168
__VER__ (predefined symbol)	169
__write (library function)	55
_formatted_write (library function)	43
_Pragma (preprocessor operator)	196

Numerics

32-bits (floating-point format)	106
64-bits (floating-point format)	106