IAR C/C++ Development Guide

Compiling and linking

for Advanced RISC Machines Ltd's

ARM® Cores

EDITION NOTICE

Fourth edition: June 2009

Part number: DARM-4

This guide applies to version 5.4*x* of IAR Embedded Workbench® for ARM.

The *IAR C/C++ Development Guide for ARM®* replaces all versions of the ARM *IAR C/C++ Compiler Reference Guide* and the *IAR Linker and Library Tools Reference Guide*.

Internal reference: R10, 5.5.*x*, ISUD.

Brief contents

Contents

Tables

Preface

Welcome to the IAR C/C++ Development Guide for ARM®. The purpose of this guide is to provide you with detailed reference information that can help you to use the build tools 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

Read this guide if you plan to develop an application using the C or C++ language for the ARM core and need detailed reference information on how to use the build tools. You should have working knowledge of:

- 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 computer.

How to use this guide

When you start using the IAR C/C++ compiler and linker for ARM, you should read *Part 1. Using the build tools* in this guide.

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

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

What this guide contains

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

Part 1. Using the build tools

- *Introduction to the IAR build tools* gives an introduction to the IAR build tools, which includes an overview of the tools, the programming languages, the available device support, and extensions provided for supporting specific features of the ARM core.
- *Developing embedded applications* gives the information you need to get started developing your embedded software using the IAR build tools.
- *Data storage* describes how to store data in memory.
- *Functions* gives a brief overview of function-related extensions—mechanisms for controlling functions—and describes some of these mechanisms in more detail.
- *Linking using ILINK* describes the linking process using the IAR ILINK Linker and the related concepts.
- *Linking your application* lists aspects that you must consider when linking your application, including using ILINK options and tailoring the linker configuration file.
- *The DLIB runtime environment* describes the DLIB runtime environment in which an application executes. It covers how you can modify it by setting options, overriding default library modules, or building your own library. The chapter also describes system initialization introducing the file `cstartup`, how to use modules for locale, and file I/O.
- *Assembler language interface* contains information required when parts of an application are written in assembler language. This includes the calling convention.
- *Using C++* gives an overview of the two levels of C++ support: The industry-standard EC++ and IAR Extended EC++.
- *Application-related considerations* discusses a selected range of application issues related to using the compiler and linker.
- *Efficient coding for embedded applications* gives hints about how to write code that compiles to efficient code for an embedded application.

Part 2. Reference information

- *External interface details* provides reference information about how the compiler and linker interact with their environment—the invocation syntax, methods for passing options to the compiler and linker, environment variables, the include file search procedure, and the different types of compiler and linker output. The chapter also describes how the diagnostic system works.

- *Compiler options* explains how to set options, gives a summary of the options, and contains detailed reference information for each compiler option.
- *Linker options* gives a summary of the options, and contains detailed reference information for each linker option.
- *Data representation* describes the available data types, pointers, and structure types. This chapter also gives information about type and object attributes.
- *Compiler extensions* gives a brief overview of the compiler extensions to the ISO/ANSI C standard. More specifically the chapter describes the available C language extensions.
- *Extended keywords* gives reference information about each of the ARM-specific keywords that are extensions to the standard C/C++ language.
- *Pragma directives* gives reference information about the pragma directives.
- *Intrinsic functions* gives reference information about functions to use for accessing ARM-specific low-level features.
- *The preprocessor* gives a brief overview of the preprocessor, including reference information about the different preprocessor directives, symbols, and other related information.
- *Library functions* gives an introduction to the C or C++ library functions, and summarizes the header files.
- *The linker configuration file* describes the purpose of the linker configuration file and describes its contents.
- *Section reference* gives reference information about the use of sections.
- *IAR utilities* describes the IAR utilities that handle the ELF and DWARF object formats.
- *Implementation-defined behavior* describes how the compiler handles the implementation-defined areas of the C language standard.

Glossary

The glossary contains definitions of programming terms.

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 IDE and the IAR C-SPY Debugger®, refer to the *IAR Embedded Workbench® IDE User Guide for ARM®*
- Programming for the ARM IAR Assembler, refer to the *ARM® IAR Assembler Reference Guide*

- Using the IAR DLIB Library functions, refer to the online help system
- Porting application code and projects created with a previous IAR Embedded Workbench for ARM, refer to the *ARM® IAR Embedded Workbench® Migration Guide*
- Using the MISRA-C:1998 rules or the MISRA-C:2004 rules, refer to the *IAR Embedded Workbench® MISRA C:1998 Reference Guide* or the *IAR Embedded Workbench® MISRA C:2004 Reference Guide*, respectively.

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

FURTHER READING

These books might be of interest to you when using the IAR Systems development tools:

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

We recommend that you visit these web sites:

- The Advanced RISC Machines Ltd web site, **www.arm.com**, contains information and news about the ARM cores, as well as information about the ARM Embedded Application Binary Interface (AEABI).
- The IAR Systems web site, **www.iar.com**, holds application notes and other product information.
- The web site **www.SevensAndNines.com**, maintained by IAR Systems, provides an online user community and resource site for ARM developers.

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

Document conventions

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

When referring to a directory in your product installation, for example `arm\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench 5.`*`n`*`\arm\doc`.

TYPOGRAPHIC CONVENTIONS

This guide uses the following typographic conventions:

Style	Used for
`computer`	• Source code examples and file paths. • Text on the command line. • Binary, hexadecimal, and octal numbers.
`parameter`	A placeholder for an actual value used as a parameter, for example *`filename`*`.h` where *`filename`* represents the name of the file.
[`option`**]**	An optional part of a command, where `[]` is part of the described syntax.
{`option`**}**	A mandatory part of a command, where `{}` is part of the described syntax.
`[option]`	An optional part of a command.
`a|b|c`	Alternatives in a command.
`{a|b|c}`	A mandatory part of a command with alternatives.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
italic	• A cross-reference within this guide or to another guide. • Emphasis.
…	An ellipsis indicates that the previous item can be repeated an arbitrary number of times.
	Identifies instructions specific to the IAR Embedded Workbench® IDE interface.
	Identifies instructions specific to the command line interface.

Table 1: Typographic conventions used in this guide

Style	Used for
	Identifies helpful tips and programming hints.
	Identifies warnings.

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

NAMING CONVENTIONS

The following naming conventions are used for the products and tools from IAR Systems® referred to in this guide:

Brand name	Generic term
IAR Embedded Workbench® for ARM	IAR Embedded Workbench®
IAR Embedded Workbench® IDE for ARM	the IDE
IAR C-SPY® Debugger for ARM	C-SPY, the debugger
IAR C-SPY® Simulator	the simulator
IAR C/C++ Compiler™ for ARM	the compiler
IAR Assembler™ for ARM	the assembler
IAR ILINK™ Linker	ILINK, the linker
IAR DLIB Library™	the DLIB library

Table 2: Naming conventions used in this guide

Part 1. Using the build tools

This part of the *IAR C/C++ Development Guide for ARM®* includes these chapters:

- Introduction to the IAR build tools
- Developing embedded applications
- Data storage
- Functions
- Linking using ILINK
- Linking your application
- The DLIB runtime environment
- Assembler language interface
- Using C++
- Application-related considerations
- Efficient coding for embedded applications.

Introduction to the IAR build tools

This chapter gives an introduction to the IAR build tools for the ARM core, which means you will get an overview of:

- The IAR build tools—the build interfaces, compiler, assembler, and linker
- The programming languages
- The available device support
- The extensions provided by the IAR C/C++ Compiler for ARM to support specific features of the ARM core.

The IAR build tools—an overview

In the IAR product installation you can find a set of tools, code examples, and user documentation, all suitable for developing software for ARM-based embedded applications. The tools allow you to develop your application in C, C++, or in assembler language.

For a more detailed product overview, see the *IAR Embedded Workbench® IDE User Guide for ARM®*. There you can also read about the debugger.

IAR Embedded Workbench® is a very powerful Integrated Development Environment (IDE) that allows you to develop and manage complete embedded application projects. It provides an easy-to-learn and highly efficient development environment with maximum code inheritance capabilities, comprehensive and specific target support. IAR Embedded Workbench promotes a useful working methodology, and thus a significant reduction of the development time.

The compiler, assembler, and linker can also be run from a command line environment, if you want to use them as external tools in an already established project environment.

IAR C/C++ COMPILER

The IAR C/C++ Compiler for ARM is a state-of-the-art compiler that offers the standard features of the C and C++ languages, plus extensions designed to take advantage of the ARM-specific facilities.

IAR ASSEMBLER

The IAR Assembler for ARM is a powerful relocating macro assembler with a versatile set of directives and expression operators. The assembler features a built-in C language preprocessor and supports conditional assembly.

The IAR Assembler for ARM uses the same mnemonics and operand syntax as the Advanced RISC Machines Ltd ARM Assembler, which simplifies the migration of existing code. For detailed information, see the *ARM® IAR Assembler Reference Guide*.

THE IAR ILINK LINKER

The IAR ILINK Linker is a powerful, flexible software tool for use in the development of embedded controller applications. It is equally well suited for linking small, single-file, absolute assembler programs as it is for linking large, relocatable input, multi-module, C/C++, or mixed C/C++ and assembler programs.

SPECIFIC ELF TOOLS

Because ILINK both uses and produces industry-standard ELF and DWARF as object format, additional IAR utilities that handle these formats can be used:

- The IAR Archive Tool—`iarchive`—creates and manipulates a library (archive) of several ELF object files
- The IAR ELF Tool—`ielftool`—performs various transformations on an ELF executable image (such as, fill, checksum, format conversion etc)
- The IAR ARM ELF Dumper—`ielfdumparm`—creates a text representation of the contents of an ELF relocatable or executable image
- The IAR ELF Object Tool—`iobjmanip`—is used for performing low-level manipulation of ELF object files
- The IAR Absolute Symbol Exporter—`isymexport`—exports absolute symbols from a ROM image file, so that they can be used when linking an add-on application.

Note: These ELF utilities are well-suited for object files produced by the tools from IAR Systems. Thus, we recommend using them instead of the GNU binary utilities.

EXTERNAL TOOLS

For information about how to extend the tool chain in the IDE, see the *IAR Embedded Workbench® IDE User Guide for ARM®.*

IAR language overview

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

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

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

For information about how the compiler handles the implementation-defined areas of the C language, see the chapter *Implementation-defined behavior*.

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 Extended Embedded C++, see the chapter *Using C++*.

Device support

To get a smooth start with your product development, the IAR product installation comes with wide range of device-specific support.

SUPPORTED ARM DEVICES

The IAR C/C++ Compiler for ARM supports several different ARM cores and devices based on the instruction sets version 4, 5, 6, 6M, and 7. The object code that the compiler generates is not always binary compatible between the cores. Therefore it is crucial to specify a processor option to the compiler. The default core is ARM7TDMI.

PRECONFIGURED SUPPORT FILES

The IAR product installation contains a vast amount of preconfigured files for supporting different devices.

Header files for I/O

Standard peripheral units are defined in device-specific I/O header files with the filename extension h. The product package supplies I/O files for all devices that are available at the time of the product release. You can find these files in the arm\inc\<*vendor*> directory. Make sure to include the appropriate include file in your application source files. If you need additional I/O header files, they can be created using one of the provided ones as a template. For detailed information about the header file format, see EWARM_HeaderFormat.pdf located in the arm\doc\ directory.

Device description files

The debugger handles several of the device-specific requirements, such as definitions of peripheral registers and groups of these, by using device description files. These files are located in the arm\inc directory and they have the filename extension ddf. To read more about these files, see the *IAR Embedded Workbench® IDE User Guide for ARM®* and EWARM_DDFFormat.pdf located in the arm\doc\ directory.

EXAMPLES FOR GETTING STARTED

The arm\examples directory contains several hundreds of examples of working applications to give you a smooth start with your development. The complexity of the examples ranges from simple LED blink to USB mass storage controllers. There are examples provided for most of the supported devices.

Special support for embedded systems

This section briefly describes the extensions provided by the compiler to support specific features of the ARM core.

EXTENDED KEYWORDS

The 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 IDE.

The command line option `-e` makes the extended keywords available, and reserves them so that they cannot be used as variable names. See, *-e*, page 169 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 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 *The preprocessor*.

SPECIAL FUNCTION TYPES

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

For detailed information, see *Primitives for interrupts, concurrency, and OS-related programming*, page 30.

ACCESSING LOW-LEVEL FEATURES

For hardware-related parts of your application, accessing low-level features is essential. The 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 91.

Developing embedded applications

This chapter provides the information you need to get started developing your embedded software for the ARM core using the IAR build tools.

First, you will get an overview of the tasks related to embedded software development, followed by an overview of the build process, including the steps involved for compiling and linking an application.

Next, the program flow of an executing application is described.

Finally, you will get an overview of the basic settings needed for a project.

Developing embedded software using IAR build tools

Typically, embedded software written for a dedicated microcontroller is designed as an endless loop waiting for some external events to happen. The software is located in ROM and executes on reset. You must consider several hardware and software factors when you write this kind of software.

MAPPING OF INTERNAL AND EXTERNAL MEMORY

Embedded systems typically contain various types of memory, such as on-chip RAM, external DRAM or SRAM, ROM, EEPROM, or flash memory.

As an embedded software developer, you must understand the features of the different memory types. For example, on-chip RAM is often faster than other types of memories, and variables that are accessed often would in time-critical applications benefit from being placed here. Conversely, some configuration data might be accessed seldom but must maintain their value after power off, so they should be saved in EEPROM or flash memory.

For efficient memory usage, the compiler provides several mechanisms for controlling placement of functions and data objects in memory. For an overview see *Controlling data and function placement in memory*, page 131. The linker places sections of code in memory according to the directives you specify in the linker configuration file, see *Placing code and data—the linker configuration file*, page 42.

COMMUNICATION WITH PERIPHERAL UNITS

If external devices are connected to the microcontroller, you might need to initialize and control the signalling interface, for example by using chip select pins, and detect and handle external interrupt signals. Typically, this must be initialized and controlled at runtime. The normal way to do this is to use special function registers, or SFRs. These are typically available at dedicated addresses, containing bits that control the chip configuration.

Standard peripheral units are defined in device-specific I/O header files with the filename extension `h`. See *Device support*, page 5. For an example, see *Accessing special function registers*, page 142.

EVENT HANDLING

In embedded systems, using *interrupts* is a method for handling external events immediately; for example, detecting that a button was pressed. In general, when an interrupt occurs in the code, the core simply stops executing the code it runs, and starts executing an interrupt routine instead.

The compiler supports the following processor exception types: interrupts, software interrupts, and fast interrupts, which means that you can write your interrupt routines in C, see *Interrupt functions*, page 31.

SYSTEM STARTUP

In all embedded systems, system startup code is executed to initialize the system—both the hardware and the software system—before the `main` function of the application is called. The CPU imposes this by starting execution from a fixed memory address.

As an embedded software developer, you must ensure that the startup code is located at the dedicated memory addresses, or can be accessed using a pointer from the vector table. This means that startup code and the initial vector table must be placed in non-volatile memory, such as ROM, EPROM, or flash.

A C/C++ application further needs to initialize all global variables. This initialization is handled by the linker and the system startup code in conjunction. For more information, see *Application execution—an overview*, page 14.

REAL-TIME OPERATING SYSTEMS

In many cases, the embedded application is the only software running in the system. However, using an RTOS has some advantages.

For example, the timing of high-priority tasks is not affected by other parts of the program which are executed in lower priority tasks. This typically makes a program more deterministic and can reduce power consumption by using the CPU efficiently and putting the CPU in a lower-power state when idle.

Using an RTOS can make your program easier to read and maintain, and in many cases smaller as well. Application code can be cleanly separated in tasks which are truly independent of each other. This makes teamwork easier, as the development work can be easily split into separate tasks which are handled by one developer or a group of developers.

Finally, using an RTOS reduces the hardware dependence and creates a clean interface to the application, making it easier to port the program to different target hardware.

INTEROPERABILITY WITH OTHER BUILD TOOLS

The IAR compiler and linker provide support for AEABI, the Embedded Application Binary Interface for ARM. For more information about this interface specification, see the `www.arm.com` web site.

The advantage of this interface is the interoperability between vendors supporting it; an application can be built up of libraries of object files produced by different vendors and linked with a linker from any vendor, as long as they adhere to the AEABI standard.

AEABI specifies full compatibility for C and C++ object code, and for the C library. The AEABI does not include specifications for the C++ library.

For more information about the AEABI support in the IAR build tools, see *AEABI compliance*, page 123.

The ARM IAR build tools version 5.xx are not fully compatible with earlier versions of the product, see the *ARM® IAR Embedded Workbench® Migration Guide* for more information.

The build process—an overview

This section gives an overview of the build process; how the various build tools—compiler, assembler, and linker—fit together, going from source code to an executable image.

To get familiar with the process in practice, you should run one or more of the tutorials available in the *IAR Embedded Workbench® IDE User Guide for ARM®*.

THE TRANSLATION PROCESS

There are two tools in the IDE that translate application source files to intermediary object files. The IAR C/C++ Compiler and the IAR relocatable assembler. Both produce relocatable object files in the industry-standard format ELF, including the DWARF format for debug information.

Note: The compiler can also be used for translating C/C++ source code into assembler source code. If required, you can modify the assembler source code which then can be assembled into object code. For more information about the IAR Assembler, see the *ARM® IAR Assembler Reference Guide.*

This illustration shows the translation process:

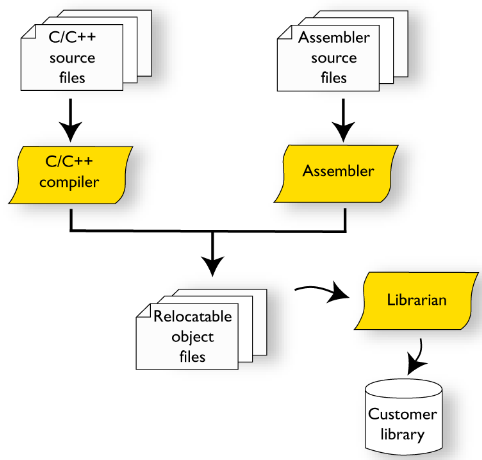


Figure 1: The build process before linking

After the translation, you can choose to pack any number of modules into an archive, or in other words, a library. The important reason you should use libraries is that each module in a library is conditionally linked in the application, or in other words, is only included in the application if the module is used directly or indirectly by a module supplied as an object file. Optionally, you can create a library; then use the IAR utility `iarchive`.

THE LINKING PROCESS

The relocatable modules, in object files and libraries, produced by the IAR compiler and assembler cannot be executed as is. To become an executable application, they must be *linked*.

Note: Modules produced by a toolset from another vendor can be included in the build as well. Be aware that this also might require a compiler utility library from the same vendor.

The IAR ILINK Linker (`ilinkarm.exe`) is used for building the final application. Normally, ILINK requires the following information as input:

- Several object files and possibly certain libraries
- A program start label (set by default)
- The linker configuration file that describes placement of code and data in the memory of the target system.

This illustration shows the linking process:

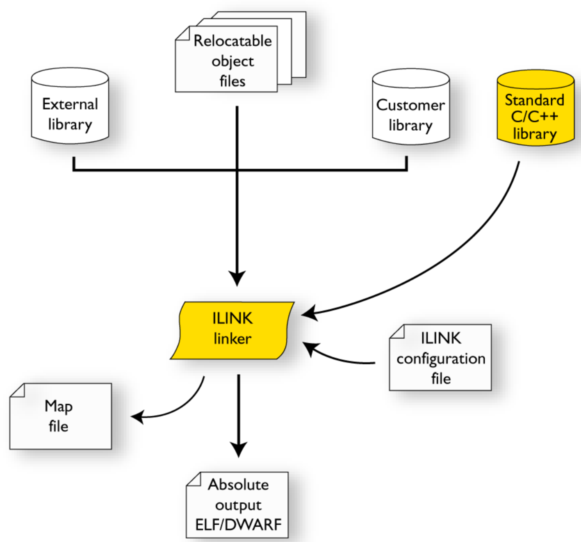


Figure 2: The linking process

Note: The standard C/C++ library contains support routines for the compiler, and the implementation of the C/C++ standard library functions.

During the linking, ILINK might produce error messages and logging messages on `stdout` and `stderr`. The log messages are useful for understanding why an application was linked the way it was, for example, why a module was included or a section removed.

For an in-depth description of the procedure performed by ILINK, see *The linking process*, page 41.

AFTER LINKING

The IAR ILINK Linker produces an absolute object file in ELF format that contains the executable image. After linking, the produced absolute executable image can be used for:

- Loading into the IAR C-SPY Debugger or any other external debugger that reads ELF and DWARF.
- Programming to a flash/PROM using a flash/PROM programmer. Before this is possible, the actual bytes in the image must be converted into the standard Motorola 32-bit S-record format or the Intel Hex-32 format. For this, use `ielftool`, see *The IAR ELF Tool—ielftool*, page 330.

This illustration shows the possible uses of the absolute output ELF/DWARF file:

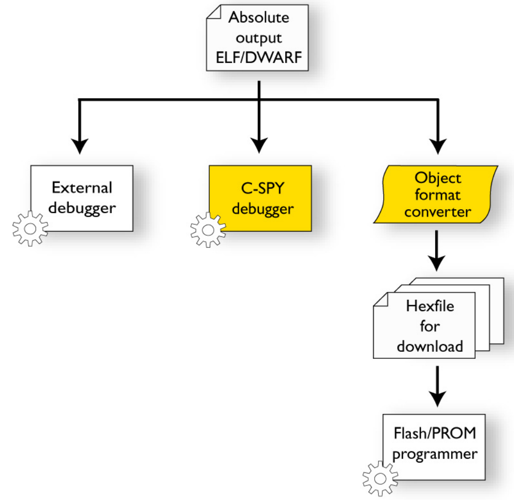


Figure 3: Possible uses of the absolute output ELF/DWARF file

Application execution—an overview

This section gives an overview of the execution of an embedded application divided into three phases, the:

- Initialization phase
- Execution phase
- Termination phase.

THE INITIALIZATION PHASE

Initialization is executed when an application is started (the CPU is reset) but before the `main` function is entered. The initialization phase can for simplicity be divided into:

- Hardware initialization, which generally at least initializes the stack pointer.

  The hardware initialization is typically performed in the system startup code `cstartup.s` and if required, by an extra low-level routine that you provide. It might include resetting/starting the rest of the hardware, setting up the CPU, etc, in preparation for the software C/C++ system initialization.
- Software C/C++ system initialization

  Typically, this includes assuring that every global (statically linked) C/C++ symbol receives its proper initialization value before the `main` function is called.
- Application initialization

  This depends entirely on your application. Typically, it can include setting up an RTOS kernel and starting initial tasks for an RTOS-driven application. For a bare-bone application, it can include setting up various interrupts, initializing communication, initializing devices, etc.

For a ROM/flash-based system, constants and functions are already placed in ROM. All symbols placed in RAM must be initialized before the `main` function is called. The linker has already divided the available RAM into different areas for variables, stack, heap, etc.

The following sequence of illustrations gives a simplified overview of the different stages of the initialization.

- When an application is started, the system startup code first performs hardware initialization, such as initialization of the stack pointer to point at the end of the predefined stack area:

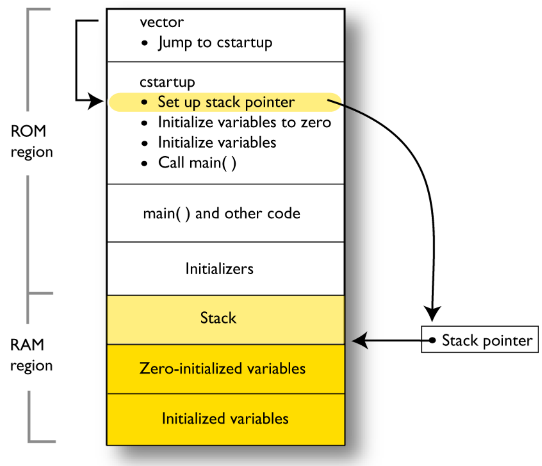


Figure 4: Initializing hardware

2 Then, memories that should be zero-initialized are cleared, in other words, filled with zeros:

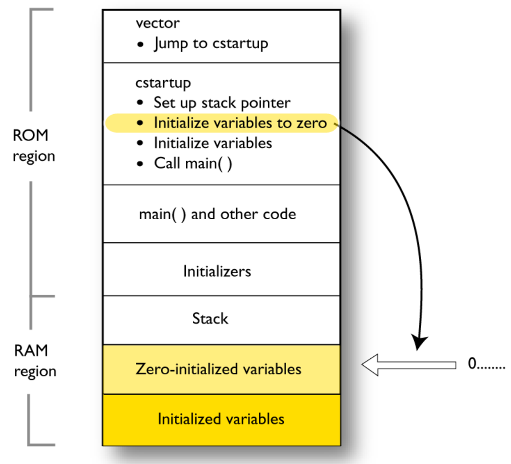


Figure 5: Zero-initializing variables

Typically, this is data referred to as *zero-initialized data*; variables declared as, for example, `int i = 0;`

3 For *initialized data*, data declared, for example, like `int i = 6;` the initializers are copied from ROM to RAM:

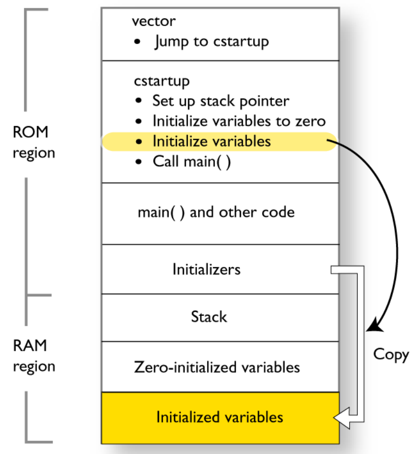


Figure 6: Initializing variables

4 Finally, the `main` function is called:

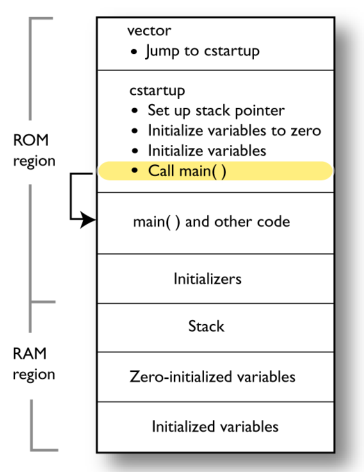


Figure 7: Calling main

For a detailed description about each stage, see *System startup and termination*, page 72. For more details about initialization of data, see *Initialization at system startup*, page 45.

THE EXECUTION PHASE

The software of an embedded application is typically implemented as a loop which is either interrupt-driven or uses polling for controlling external interaction or internal events. For an interrupt-driven system, the interrupts are typically initialized at the beginning of the `main` function.

In a system with real-time behavior and where responsiveness is critical, a multi-task system might be required. This means that your application software should be complemented with a real-time operating system. In this case, the RTOS and the different tasks must also be initialized at the beginning of the `main` function.

THE TERMINATION PHASE

Typically, the execution of an embedded application should never end. If it does, you must define a proper end behavior.

To terminate an application in a controlled way, either call one of the standard C library functions `exit`, `_Exit`, or `abort`, or return from `main`. If you return from `main`, the `exit` function is executed, which means that C++ destructors for static and global variables are called (C++ only) and all open files are closed.

Of course, in case of incorrect program logic, the application might terminate in an uncontrolled and abnormal way—a system crash.

To read more about this, see *System termination*, page 75.

Basic project configuration

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

```
iccarm myfile.c
```

On the command line, this line can be used for starting ILINK:

```
ilinkarm myfile.o myfile2.o -o a.out --config my_configfile.icf
```

In this example, `myfile.o` and `myfile2.o` are object files, and `my_configfile.icf` is the linker configuration file. The option `-o` specifies the name of the output file.

Note: By default, the label where the application starts is `__iar_program_start`. You can use the `--entry` command line option to change this.

However, you must specify some additional options. This section gives an overview of the basic settings for the project setup that are needed to make the compiler and linker generate the best code for the ARM device you are using. You can specify the options either from the command line interface or in the IDE.

You need settings for:

- Processor configuration, that is processor variant, CPU mode, interworking, VFP and floating-point arithmetic, and byte order
- Optimization settings
- Runtime environment
- Customizing the ILINK configuration, see the chapter *Linking your application*
- AEBI compliance, see *AEABI compliance*, page 123.

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

PROCESSOR CONFIGURATION

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

Processor variant

The IAR C/C++ Compiler for ARM supports several different ARM cores and devices based on the instruction sets version 4, 5, 6, and 7. All supported cores support Thumb instructions and 64-bit multiply instructions. The object code that the compiler generates is not always binary compatible between the cores. Therefore it is crucial to specify a processor option to the compiler. The default core is ARM7TDMI.

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

Use the `--cpu` option to specify the ARM core; see *--cpu*, page 162 for syntax information.

CPU mode

The IAR C/C++ Compiler for ARM 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 *IAR Embedded Workbench® IDE User Guide for ARM®* for information about setting the **Processor variant** or **Chip** option in the IDE.

Use the `--arm` or `--thumb` option to specify the CPU mode for your project; see *--arm*, page 162 and *--thumb*, page 187, for syntax information.

Interworking

When code is compiled with the `--interwork` option, ARM and Thumb code can be freely mixed. Interworking functions can be called from both ARM and Thumb code. Interworking is default for devices based on the instruction sets version 5, 6, and 7, or when using the `--aeabi` compiler option.

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

Use the `--interwork` option to specify interworking capabilities for your project; see *--interwork*, page 173, 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 *IAR Embedded Workbench® IDE User Guide for ARM®* for information about setting the **FPU** option in the IDE.

Use the `--fpu` option to use the coprocessor for floating-point operations; see *--fpu*, page 172, for syntax information.

Byte order

The IAR C/EC++ Compiler for ARM 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 *IAR Embedded Workbench® IDE User Guide for ARM®* for information about setting the **Endian mode** option in the IDE.

Use the `--endian` option to specify the byte order for your project; see *--endian*, page 170, for syntax information.

OPTIMIZATION FOR SPEED AND SIZE

The compiler is a state-of-the-art compiler with an optimizer that performs, among other things, dead-code elimination, constant propagation, inlining, common subexpression elimination, 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 for the highest level you can choose between different optimization goals—*size*, *speed*, or *balanced*. Most optimizations will make the application both smaller and faster. However, when this is not the case, the compiler uses the selected optimization goal to decide how to perform the optimization.

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

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

RUNTIME ENVIRONMENT

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

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

The runtime library you choose can be one of the prebuilt libraries, or a library that you customized and built yourself. The 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, you do not need to choose a runtime library.

Note: Some tailoring might be required, for example to meet hardware requirements.

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 IDE or the command line.

Setting up for the runtime environment in the IDE

The library is automatically chosen by the linker according to the settings you made in **Project>Options>General Options**, on the pages **Library Configuration**, **Library Options**, and **Library Usage**.

Note that for the DLIB library there are different configurations—Normal and Full—which include different levels of support for locale, file descriptors, multibyte characters, et cetera. See *Library configurations*, page 63, 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.

Setting up for the runtime environment from the command line

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

Command line	Description
`-I arm\inc`	Specifies the include path to device-specific I/O definition files.
`--dlib_config C:\...\`*`configfile`*`.h`	Specifies the library configuration file, either `DLib_Config_Normal.h` or `DLib_Config_Full.h`

Table 3: Command line options for specifying library and dependency files

Normally, it is not needed to specify a library file explicitly, as ILINK automatically uses the correct library file.

For a list of all prebuilt library object files for the IAR DLIB Library, see *Using a prebuilt library*, page 64. Here you also get information about how the object files correspond to the dependent project options, and the corresponding configuration files. Make sure to use the object file that matches your other project options.

Setting library and runtime environment options

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

- The formatters used by the functions `printf`, `scanf`, and their variants, see *Choosing formatters for printf and scanf*, page 67.
- The size of the stack and the heap, see *Setting up the stack*, page 52, and *Setting up the heap*, page 52, respectively.

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. Finally, detailed information about data storage on the stack and the heap is provided.

Introduction

An 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.

DIFFERENT WAYS TO STORE DATA

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

- Auto variables.

  All variables that are local to a function, except those declared static, are stored on the stack. These variables can be used as long as the function executes. When the function returns to its caller, the memory space is no longer valid.
- Global variables and local variables declared `static`.

  In this case, the memory is allocated once and for all. The word static in this context means that the amount of memory allocated for this kind of variables does not change while the application is running. The ARM core has one single address space and the compiler supports full memory addressing.
- Dynamically allocated data.

  An application can allocate data on the *heap*, where the data 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 dynamically allocated data in systems with a limited amount of memory, or systems that are expected to run for a long time. For more information, see *Dynamic memory on the heap*, page 27.

Auto variables—on the stack

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

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

THE STACK

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 *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 returns. 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 here. */
  return &x; /* Incorrect */
}
```

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++, a special keyword, `new`, allocates memory and runs constructors. Memory allocated with `new` must be released using the keyword `delete`.

Potential problems

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

The heap can become exhausted if your application uses too much memory. It can also become full if memory that no longer is in use was not 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 no piece of free memory is large enough for the object, even though the sum of the sizes of the free memory exceeds the size of the object.

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

Functions

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

Function-related extensions

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

- Generate code for the different CPU modes ARM and Thumb
- Make functions execute in RAM
- Use primitives for interrupts, concurrency, and OS-related programming
- Facilitate function optimization
- Access hardware features.

The compiler uses compiler options, extended keywords, pragma directives, and intrinsic functions to support this.

For more information about optimizations, see *Efficient coding for embedded applications*, page 127. For information about the available intrinsic functions for accessing hardware operations, see the chapter *Intrinsic functions*.

ARM and Thumb code

The IAR C/C++ Compiler for ARM can generate code for either the 32-bit ARM, or the 16-bit Thumb or Thumb2 instruction set. Use the `--cpu_mode` option, alternatively the `--arm` or `--thumb` options, 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 code is interworking.

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. There is a limit of 4 Mbytes per code module.

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 215.

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.

Execution in RAM

The `__ramfunc` keyword makes a function execute in RAM, or in other words places the function in a section that has read/write attributes. The function is copied from ROM to RAM at system startup just like any initialized variable, see *System startup and termination*, page 72.

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)
```

For more details, see *Initializing code—copying ROM to RAM*, page 55.

Primitives for interrupts, concurrency, and OS-related programming

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

- The extended keywords `__irq`, `__fiq`, `__swi`, and `__nested`
- The intrinsic functions `__enable_interrupt`, `__disable_interrupt`, `__get_interrupt_state`, and `__set_interrupt_state`.

Note: ARM Cortex-M has a different interrupt mechanism than other ARM devices, and for these devices a different set of primitives is available. For more details, see *Interrupts for ARM Cortex-M*, page 36.

INTERRUPT FUNCTIONS

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

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

The compiler supports interrupts, software interrupts, and fast interrupts. For each interrupt type, an interrupt routine can be written.

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 override the default behavior.

Each interrupt routine is associated with a vector address/instruction in the exception vector table, which is specified in the ARM cores documentation. The interrupt vector is the address in the exception vector table. For the ARM cores, the exception vector table starts at address `0x0`.

To define an interrupt function, the `__irq` or the `__fiq` keyword can be used. For example:

```
__irq __arm void IRQ_Handler(void)
{
  /* Do something */
}
```

See the ARM cores documentation for more information about the interrupt vector table.

INSTALLING EXCEPTION FUNCTIONS

All interrupt functions and software interrupt handlers must be installed in the vector table. This is done in assembler language in the system startup file `cstartup.s`.

The default implementation of the ARM exception vector table in the standard runtime library jumps to predefined functions that implement an infinite loop. Any exception that occurs for an event not handled by your application will therefore be caught in the infinite loop `(B. )`.

The predefined functions are defined as weak symbols. A weak symbol is only included by the linker as long as no duplicate symbol is found. If another symbol is defined with the same name, it will take precedence. Your application can therefore simply define its own exception function by just defining it using the correct name.

These exception function names are defined in `cstartup.s` and referred to by the library exception vector code:

```
Undefined_Handler
SWI_Handler
Prefetch_Handler
Abort_Handler
IRQ_Handler
FIQ_Handler
```

To implement your own exception handler, define a function using the appropriate exception function name from the list above.

For example to add an interrupt function in C, it is sufficient to define an interrupt function named `IRQ_Handler`:

```
__irq __arm void IRQ_Handler()
{
}
```

An interrupt function must have C linkage, read more in *Calling convention*, page 97.

If you use C++, an interrupt function could look, for example, like this:

```
extern "C"
{
__irq __arm void IRQ_Handler(void);
}

__irq __arm void IRQ_Handler()
{
}
```

No other changes are needed.

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.

- To declare an interrupt function, use the `__irq` extended keyword or the `#pragma type_attribute=__irq` directive. For syntax information, see *__irq*, page 238, and *type_attribute*, page 257, respectively.

- To declare a fast interrupt function, use the `__fiq` extended keyword or the `#pragma type_attribute=__fiq` directive. For syntax information, see *__fiq*, page 237, and *type_attribute*, page 257, 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.s` 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.
  VICVectAddr = 0;               // Acknowledge interrupt in VIC.
  interrupt_task = (void(*)())vector;

  __enable_interrupt();          // Allow other IRQ interrupts
                                    to be serviced from this
                                    point.
  (*interrupt_task)();           // Execute the task associated
                                    with this interrupt.
}
```

Note: The `__nested` keyword requires the processor mode to be in either User or System mode.

SOFTWARE INTERRUPTS

Software interrupt functions are slightly more complex than other interrupt functions, in the way that they need a software interrupt handler (a dispatcher), are invoked (called) from running application software, and that they accept arguments and have return values. The mechanisms for calling a software interrupt function and how the software interrupt handler dispatches the call to the actual software interrupt function is described here.

Calling a software interrupt function

To call a software interrupt function from your application source code, the assembler instruction `SVC #immed` is used, where `immed` is an integer value that is referred to as the software interrupt number—or `swi_number`—in this guide. The compiler provides an easy way to implicitly generate this instruction from C/C++ source code, by using the `__swi` keyword and the `#pragma swi_number` directive when declaring the function.

A `__swi` function can for example be declared like this:

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

In this case, the assembler instruction `SVC 0x23` will be generated where the function is called.

Software interrupt functions follow the same calling convention regarding parameters and return values as an ordinary function, except for the stack usage, see *Calling convention*, page 97.

For more information, see *__swi*, page 241, and *swi_number*, page 257, respectively.

The software interrupt handler and functions

The interrupt handler, for example `SWI_Handler` works as a dispatcher for software interrupt functions. It is invoked from the interrupt vector and is responsible for retrieving the software interrupt number and then calling the proper software interrupt function. The `SWI_Handler` must be written in assembler as there is no way to retrieve the software interrupt number from C/C++ source code.

The software interrupt functions

The software interrupt functions can be written in C or C++. Use the `__swi` keyword in a function definition to make the compiler generate a return sequence suited for a specific software interrupt function. The `#pragma swi_number` directive is not needed in the interrupt function definition.

For more information, see *__swi*, page 241.

Setting up the software interrupt stack pointer

If software interrupts will be used in your application, then the software interrupt stack pointer (`SVC_STACK`) must be set up and some space must be allocated for the stack. The `SVC_STACK` pointer can be setup together with the other stacks in the `cstartup.s` file. As an example, see the set up of the interrupt stack pointer. Relevant space for the `SVC_STACK` pointer is set up in the linker configuration file, see *Setting up the stack*, page 52.

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.

INTERRUPTS FOR ARM CORTEX-M

ARM Cortex-M has a different interrupt mechanism than previous ARM architectures, which means the primitives provided by the compiler are also different.

On ARM Cortex-M, an interrupt service routine enters and returns in the same way as a normal function, which means no special keywords are required. Thus, the keywords `__irq`, `__fiq`, and `__nested` are not available when you compile for ARM Cortex-M.

These exception function names are defined in `cstartup_M.c` and `cstartup_M.s`. They are referred to by the library exception vector code:

```
NMI_Handler
HardFault_Handler
MemManage_Handler
BusFault_Handler
UsageFault_Handler
SVC_Handler
DebugMon_Handler
PendSV_Handler
SysTick_Handler
```

The vector table is implemented as an array. It should always have the name `__vector_table`, because `cmain` refers to that symbol and C-SPY looks for that symbol when determining where the vector table is located.

The predefined exception functions are defined as weak symbols. A weak symbol is only included by the linker as long as no duplicate symbol is found. If another symbol is defined with the same name, it will take precedence. Your application can therefore simply define its own exception function by just defining it using the correct name from the list above. If you need other interrupts or other exception handlers, you must make a copy of the `cstartup_M.c` or `cstartup_M.s` file and make the proper addition to the vector table.

The intrinsic functions `__get_CPSR` and `__set_CPSR` are not available when you compile for ARM Cortex-M. Instead, if you need to get or set values of these or other registers, you can use inline assembler. For more information, see *Passing values between C and assembler objects*, page 143.

C++ AND SPECIAL FUNCTION TYPES

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

Special function types can be used for static member functions. For example, in the following example, the function `handler` is declared as an interrupt function:

```
class Device
{
  static __irq void handler();
};
```

Linking using ILINK

This chapter describes the linking process using the IAR ILINK Linker and the related concepts—first with an overview and then in more detail.

Linking—an overview

The IAR ILINK Linker is a powerful, flexible software tool for use in the development of embedded applications. It is equally well suited for linking small, single-file, absolute assembler programs as it is for linking large, relocatable, multi-module, C/C++, or mixed C/C++ and assembler programs.

ILINK combines one or more relocatable object files—produced by the IAR Systems compiler or assembler—with selected parts of one or more object libraries to produce an executable image in the industry-standard format *Executable and Linking Format* (ELF).

ILINK will automatically load only those library modules—user libraries and standard C or C++ library variants—that are actually needed by the application you are linking. Further, ILINK eliminates duplicate sections and sections that are not required.

ILINK can link both ARM and Thumb code, as well as a combination of them. By automatically inserting additional instructions (veneers), ILINK will assure that the destination will be reached for any calls and branches, and that the processor state is switched when required. For more details about how to generate veneers, see *Veneers*, page 57.

ILINK uses a *configuration file* where you can specify separate locations for code and data areas of your target system memory map. This file also supports automatic handling of the application's initialization phase, which means initializing global variable areas and code areas by copying initializers and possibly decompressing them as well.

The final output produced by ILINK is an absolute object file containing the executable image in the ELF (including DWARF for debug information) format. The file can be downloaded to C-SPY or any other debugger that supports ELF/DWARF, or it can be programmed into EPROM.

To handle ELF files, various tools are included. For a list of included utilities, see *Specific ELF tools*, page 4.

Modules and sections

Each relocatable object file contains one module, which consists of:

- Several sections of code or data
- Runtime attributes specifying various types of information, for example the used device
- Optionally, debug information in DWARF format
- A symbol table of all global symbols and all external symbols used.

A *section* is a logical entity containing a piece of data or code that should be placed at a physical location in memory. A section can consist of several *section fragments*, typically one for each variable or function (symbols). A section can be placed either in RAM or in ROM. In a normal embedded application, sections that are placed in RAM do not have any content, they only occupy space.

Each section has a name and a type attribute that determines the content. The type attribute is used (together with the name) for selecting sections for the ILINK configuration. The most commonly used attributes are:

code	Executable code
readonly	Constant variables
readwrite	Initialized variables
zeroinit	Zero-initialized variables

Note: In addition to these section types—sections that contain the code and data that are part of your application—a final object file will contain many other types of sections, for example sections that contain debugging information or other type of meta information.

A section is the smallest linkable unit; but if possible, ILINK can exclude smaller units—section fragments—from the final application. For more information, see *Keeping modules*, page 51, and *Keeping symbols and sections*, page 52.

At compile time, data and functions are placed in different sections. At link time, one of the most important functions of the linker is to assign execute addresses to the various sections used by the application.

The IAR build tools have many predefined section names. See the chapter *Section reference* for more details about each section.

The linking process

The relocatable modules in object files and libraries, produced by the IAR compiler and assembler, cannot be executed as is. To become an executable application, they must be *linked*.

Note: Modules produced by a toolset from another vendor can be included in the build as well, as long as the module is AEABI (ARM Embedded Application Binary Interface) compliant. Be aware that this also might require a compiler utility library from the same vendor.

The IAR ILINK Linker is used for the link process. It normally performs the following procedure (note that some of the steps can be turned off by command line options or by directives in the linker configuration file):

- Determine which modules to include in the application. Modules provided in object files are always included. A module in a library file is only included if it provides a definition for a global symbol that is referenced from an included module.
- Select which standard library files to use. The selection is based on attributes of the included modules. These libraries are then used for satisfying any still outstanding undefined symbols.
- Determine which sections/section fragments from the included modules to include in the application. Only those sections/section fragments that are actually needed by the application are included. There are several ways to determine of which sections/section fragments that are needed, for example, the `__root` object attribute, the `#pragma required` directive, and the `keep` linker directive. In case of duplicate sections, only one is included.
- Where appropriate, arrange for the initialization of initialized variables and code in RAM. The `initialize` directive causes the linker to create extra sections to enable copying from ROM to RAM. Each section that will be initialized by copying is divided into two sections, one for the ROM part and one for the RAM part. If manual initialization is not used, the linker also arranges for the startup code to perform the initialization.
- Determine where to place each section according to the section placement directives in the *linker configuration file*. Sections that are to be initialized by copying appear twice in the matching against placement directives, once for the ROM part and once for the RAM part, with different attributes. During the placement, the linker also adds any required veneers to make a code reference reach its destination or to switch CPU modes.
- Produce an absolute object file that contains the executable image and any debug information provided. This involves resolving symbolic references between sections, and locating relocatable values.

- Optionally, produce a map file that lists the result of the section placement, the address of each global symbol, and finally, a summary of memory usage for each module and library.

This illustration shows the linking process:

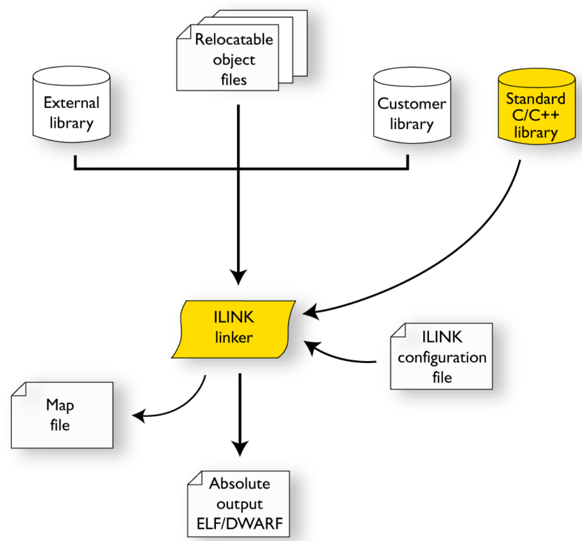


Figure 8: The linking process

During the linking, ILINK might produce error messages and logging messages on `stdout` and `stderr`. The log messages are useful for understanding why an application was linked as it was. For example, why a module or section (or section fragment) was included.

Note: To see the actual content of an ELF object file, use `ielfdumparm`. See *The IAR ELF Dumper for ARM—ielfdumparm*, page 336.

Placing code and data—the linker configuration file

The placement of sections in memory is performed by the IAR ILINK Linker. It uses the *linker configuration file* where you can define how ILINK should treat each section and how they should be placed into the available memories.

A typical linker configuration file contains definitions of:

- Available addressable memories

- Populated regions of those memories
- How to treat input sections
- Created sections
- How to place sections into the available regions.

The file consists of a sequence of declarative directives. This means that the linking process will be governed by all directives at the same time.

To use the same source code with different derivatives, just rebuild the code with the appropriate configuration file.

A SIMPLE EXAMPLE OF A CONFIGURATION FILE

A simple configuration file can look like this:

```
/* The memory space denoting the maximum possible amount
   of addressable memory */
define memory Mem with size = 4G;

/* Memory regions in an address space */
define region ROM = Mem:[from 0x00000 size 0x10000];
define region RAM = Mem:[from 0x20000 size 0x10000];

/* Create a stack */
define block STACK with size = 0x1000, alignment = 8 { };

/* Handle initialization */
do not initialize { section .noinit };
initialize by copy { readwrite }; /* Initialize RW sections,
                                     exclude zero-initialized
                                     sections */

/* Place startup code at a fixed address */
place at start of ROM { readonly section .cstartup };

/* Place code and data */
place in ROM { readonly }; /* Place constants and initializers in
                              ROM: .rodata and .data_init    */
place in RAM { readwrite,  /* Place .data, .bss, and .noinit */
               block STACK }; /* and STACK                   */
```

This configuration file defines one addressable memory `Mem` with the maximum of 4 Gbytes of memory. Further, it defines a ROM region and a RAM region in `Mem`, namely `ROM` and `RAM`. Each region has the size of 64 Kbytes.

The file then creates an empty block called `STACK` with a size of 4 Kbytes in which the application stack will reside. To create a *block* is the basic method which you can use to get detailed control of placement, size, etc. It can be used for grouping sections, but also as in this example, to specify the size and placement of an area of memory.

Next, the file defines how to handle the initialization of variables, read/write type (`readwrite`) sections. In this example, the initializers are placed in ROM and copied at startup of the application to the RAM area. By default, ILINK may compress the initializers if this appears to be advantageous.

The last part of the configuration file handles the actual placement of all the sections into the available regions. First, the startup code—defined to reside in the read-only (`readonly`) section `.cstartup`—is placed at the start of the `ROM` region, that is at address `0x10000`. Note that the part within `{}` is referred to as *section selection* and it selects the sections for which the directive should be applied to. Then the rest of the read-only sections are placed in the `ROM` region. Note that the section selection `{ readonly section .cstartup }` takes precedence over the more generic section selection `{ readonly }`.

Finally, the read/write (`readwrite`) sections and the `STACK` block are placed in the `RAM` region.

This illustration gives a schematic overview of how the application is placed in memory:

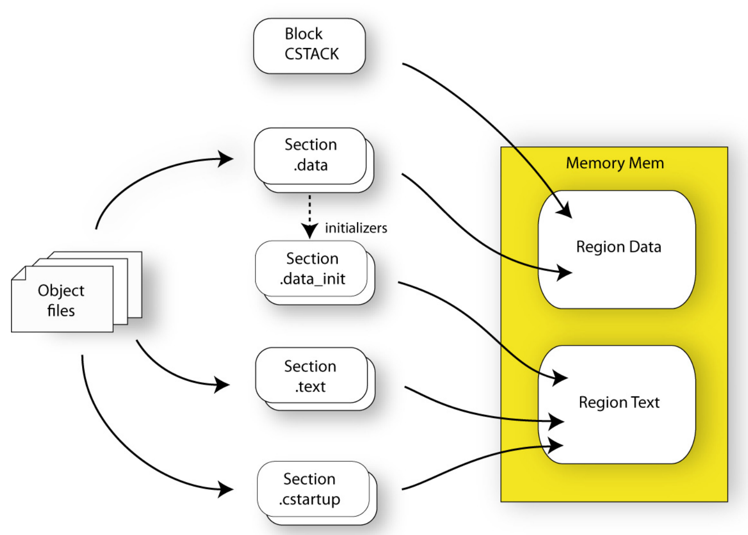


Figure 9: Application in memory

In addition to these standard directives, a configuration file can contain directives that define how to:

- Map a memory that can be addressed in multiple ways
- Handle conditional directives
- Create symbols with values that can be used in the application
- More in detail, select the sections a directive should be applied to
- More in detail, initialize code and data.

For more details and examples about customizing the linker configuration file, see the chapter *Linking your application*.

For reference information about the linker configuration file, see the chapter *The linker configuration file*.

Initialization at system startup

In ISO/ANSI C, all static variables—variables that are allocated at a fixed memory address—must be initialized by the runtime system to a known value at application startup. This value is either an explicit value assigned to the variable, or if no value is given, it is cleared to zero. In the compiler, there is one exception to this rule and that is variables declared `__no_init` which are not initialized at all.

The compiler generates a specific type of section for each type of variable initialization:

Categories of declared data	Source	Section type	Section name	Section content
Zero-initialized data	`int i;`	Read/write data, zero-init	`.bss`	None
Zero-initialized data	`int i = 0;`	Read/write data, zero-init	`.bss`	None
Initialized data (non-zero)	`int i = 6;`	Read/write data	`.data`	The initializer
Non-initialized data	`__no_init int i;`	Read/write data, zero-init	`.noinit`	None
Constants	`const int i = 6;`	Read-only data	`.rodata`	The constant
Code	`__ramfunc void myfunc() {}`	Read/write code	`.textrw`	The code

Table 4: Sections holding initialized data

Note: Clustering of static variables might group zero-initialized variables together with initialized data in `.data`.

For a summary of all supported sections, see the chapter *Section reference*.

THE INITIALIZATION PROCESS

Initialization of data is handled by ILINK and the system startup code in conjunction.

To configure the initialization of variables, you must consider these issues:

- Sections that should be zero-initialized are handled automatically by ILINK; they should only be placed in RAM
- Sections that should be initialized, except for zero-initialized sections, should be listed in an `initialize` directive

  Normally during linking, a section that should be initialized is split in two sections, where the original initialized section will keep the name. The contents are placed in the new initializer section, which will keep the original name suffixed with `_init`. The initializers should be placed in ROM and the initialized sections in RAM, by means of placement directives. The most common example is the `.data` section that the linker splits in `.data` and `.data_init`.
- Sections that contains constants should not be initialized; they should only be placed in flash/ROM
- Sections holding `__no_init` declared variables should not be initialized and thus should be listed in a `do not initialize` directive. They should also be placed in RAM.

In the linker configuration file, it can look like this:

```
/* Handle initialization */
do not initialize { section .noinit };
initialize by copy { readwrite }; /* Initialize RW sections,
                                     exclude zero-initialized
                                     sections */

/* Place startup code at a fixed address */
place at start of ROM { readonly section .cstartup };

/* Place code and data */
place in ROM { readonly }; /* Place constants and initializers in
                              ROM: .rodata and .data_init    */
place in RAM { readwrite,  /* Place .data, .bss, and .noinit */
               block STACK }; /* and STACK                   */
```

For detailed information and examples about how to configure the initialization, see *Linking considerations*, page 47.

Linking your application

This chapter lists aspects that you must consider when linking your application. This includes using ILINK options and tailoring the linker configuration file.

Finally, this chapter provides some hints for troubleshooting.

Linking considerations

Before you can link your application, you must set up the configuration required by ILINK. Typically, you must consider:

- Defining your own memory areas
- Placing sections
- Keeping modules in the application
- Keeping symbols and sections in the application
- Application startup
- Setting up the stack and heap
- Setting up the `atexit` limit
- Changing the default initialization
- Symbols for controlling the application
- Standard library handling
- Other output formats than ELF/DWARF
- Veneers.

CHOOSING A LINKER CONFIGURATION FILE

The `config` directory contains two ready-made templates for the linker configuration file:

- `generic.icf`, designed for all cores except for Cortex-M cores
- `generic_cortex.icf`, designed for all Cortex-M cores

These files contain the information required by ILINK. The only change you will normally have to make to the supplied configuration file is to customize the start and end addresses of each region so they fit the target system memory map. If, for example, your application uses additional external RAM, you must also add details about the external RAM memory area.

To edit a linker configuration file, use the editor in the IDE, or any other suitable editor. Alternatively, choose **Project>Options>Linker** and click the **Edit** button on the **Config** page to open the dedicated linker configuration file editor.

Remember not to change the original template file. We recommend that you make a copy in the working directory, and modify the copy instead. If you are using the linker configuration file editor in the IDE, the IDE will make a copy for you.

Each project in the IDE should have a reference to one, and only one, linker configuration file. This file can be edited, but for the majority of all projects it is sufficient to configure the vital parameters in **Project>Options>Linker>Config**.

DEFINING YOUR OWN MEMORY AREAS

The default configuration file that you selected has predefined ROM and RAM regions. This example will be used as a starting-point for all further examples in this chapter:

```
/* Define the addressable memory */
define memory Mem with size = 4G;

/* Define a region named ROM with start address 0 and to be 64
Kbytes large */
define region ROM = Mem:[from 0 size 0x10000];

/* Define a region named RAM with start address 0x20000 and to be
64 Kbytes large */
define region RAM = Mem:[from 0x20000 size 0x10000];
```

Each region definition must be tailored for the actual hardware.

To find out how much of each memory that was filled with code and data after linking, inspect the memory summary in the map file (command line option `--map`).

Adding an additional region

To add an additional region, use the `define region` directive, for example:

```
/* Define a 2nd ROM region to start at address 0x80000 and to be
128 Kbytes large */
define region ROM2 = Mem:[from 0x80000 size 0x20000];
```

Merging different areas into one region

If the region is comprised of several areas, use a region expression to merge the different areas into one region, for example:

```
/* Define the 2nd ROM region to have two areas. The first with
the start address 0x80000 and 128 Kbytes large, and the 2nd with
the start address 0xC0000 and 32 Kbytes large */
```

```
define region ROM2 = Mem:[from 0x80000 size 0x20000]
                   | Mem:[from 0xC0000 size 0x08000];
```

or equivalently

```
define region ROM2 = Mem:[from 0x80000 to 0xC7FFF]
                    –Mem:[from 0xA0000 to 0xBFFFF];
```

Adding a region in a new memory

To add a region in a new memory, write:

```
/* Define a 2nd addressable memory */
define memory Mem2 with size = 64k;
/* Define a region for constants with start address 0 and 64
Kbytes large */
define region CONSTANT = Mem2:[from 0 size 0x10000];
```

Defining the unit size for a new memory

If the new memory is not byte-oriented (8-bits per byte) you should define what unit size to use:

```
/* Define the bit addressable memory */
define memory Bit with size = 256, unitbitsize = 1;
```

Sharing memories

If your core can address a physical memory either by:

- Several different addressing modes; addresses in different defined memories are actually the same physical entity
- Using different addresses in the same memory, for example some bits in the address are not connected to the physical memory

the `define sharing` directive must be used. For example:

```
/* First 32 Kbytes of Mem2 are mirrored in the last 32 Kbytes */
define sharing Mem2:[from 0 size 0x8000] <=>
                Mem2:[from 0x8000 size 0x8000];
/* Bit memory is mapped in the first 32 bytes of Mem2 */
define sharing Bit:[from 0 size 256] <=> Mem2:[from 0 size 32];
```

The sharing directive instructs ILINK to allocate contents in all connected memories if any content is placed in one memory.

PLACING SECTIONS

The default configuration file that you selected places all predefined sections in memory, but there are situations when you might want to modify this. For example, if you want to place the section that holds constant symbols in the `CONSTANT` region instead of in the default place. In this case, use the `place in` directive, for example:

```
/* Place sections with readonly content in the ROM region */
place in ROM {readonly};
/* Place the constant symbols in the CONSTANT region */
place in CONSTANT {readonly section .rodata};
```

Note: Placing a section—used by the IAR build tools—in a different memory which use a different way of referring to its content, will fail.

For the result of each placement directive after linking, inspect the placement summary in the map file (the command line option `--map`).

Placing a section at a specific address in memory

To place a section at a specific address in memory, use the `place at` directive, for example:

```
/* Place section .vectors at address 0 */
place at address Mem:[0] {readonly section .vectors};
```

Placing a section first or last in a region

To place a section first or last in a region is similar, for example:

```
/* Place section .vectors at start of ROM */
place at start of ROM {readonly section .vectors};
```

Declare and place your own sections

To declare new sections—in addition to the ones used by the IAR build tools—to hold specific parts of your code or data, use mechanisms in the compiler and assembler. For example:

```
/* Places a variable in your own section MyOwnSection. */
const int MyVariable @ "MyOwnSection" = 5;
```

```
        name    createSection

        /* Create a section */
        section myOwnSection:CONST

        /* And fill it with constant bytes */
        dcb     5, 6, 7, 8

        end
```

To place your new section, the original `place in ROM {readonly};` directive is sufficient.

However, to place the section `MyOwnSection` explicitly, update the linker configuration file with a `place in` directive, for example:

```
/* Place MyOwnSection in the ROM region */
place in ROM {readonly section MyOwnSection};
```

RESERVING SPACE IN RAM

Often, an application must have an empty uninitialized memory area to be used for temporary storage, for example a heap or a stack. It is easiest to achieve this at link time. You must create a block with a specified size and then place it in a memory.

In the linker configuration file, it can look like this:

```
define block TempStorage with size = 0x1000, alignment = 4 { };
place in RAM { block TempStorage };
```

To retrieve the start of the allocated memory from the application, the source code could look like this:

```
/* Declares a section */
#pragma section = "TempStorage"

char *TempStorage()
{
  /* Return start address of section TempStorage. */
  return __section_begin("TempStorage");
}
```

KEEPING MODULES

If a module is linked as an object file, it is always kept. That is, it will contribute to the linked application. However, if a module is part of a library, it is included only if it is symbolically referred to from other parts of the application. This is true, even if the library module contains a root symbol. To assure that such a library module is always included, use `iarchive` to extract the module from the library, see *The IAR Archive Tool—iarchive*, page 323.

For information about included and excluded modules, inspect the log file (the command line option `--log modules`).

For more information about modules, see *Modules and sections*, page 40.

KEEPING SYMBOLS AND SECTIONS

By default, ILINK removes any sections, section fragments, and global symbols that are not needed by the application. To retain a symbol that does not appear to be needed—or actually, the section fragment it is defined in—you can either use the root attribute on the symbol in your C/C++ or assembler source code, or use the ILINK option `--keep`. To retain sections based on attribute names or object names, use the directive `keep` in the linker configuration file.

To prevent ILINK from excluding sections and section fragments, use the command line options `--no_remove` or `--no_fragments`, respectively.

For information about included and excluded symbols and sections, inspect the log file (the command line option `--log sections`).

For more information about the linking procedure for keeping symbols and sections, see *The linking process*, page 41.

APPLICATION STARTUP

By default, the point where the application starts execution is defined by the `__iar_program_start` label, which is defined to point at the start of the `cstartup.s` file. The label is also communicated via ELF to any debugger that is used.

To change the start point of the application to another label, use the ILINK option `--entry`; see *--entry*, page 196.

SETTING UP THE STACK

The size of the `CSTACK` block is defined in the linker configuration file. To change the allocated amount of memory, change the block definition for `CSTACK`:

```
define block CSTACK with size = 0x2000, alignment = 8{ };
define block IRQ_STACK with size = 64, alignment = 8{ };
```

Specify an appropriate size for your application.

To read more about the stack, see *Stack considerations*, page 115.

SETTING UP THE HEAP

The size of the heap is defined in the linker configuration file as a block:

```
define block HEAP with size = 0x1000, alignment = 8{ };
place in RAM {block HEAP};
```

Specify the appropriate size for your application.

SETTING UP THE ATEXIT LIMIT

By default, the `atexit` function can be called a maximum of 32 times from your application. To either increase or decrease this number, add a line to your configuration file. For example, to reserve room for 10 calls instead, write:

```
define symbol __iar_maximum_atexit_calls = 10;
```

CHANGING THE DEFAULT INITIALIZATION

By default, memory initialization is performed during application startup. ILINK sets up the initialization process and chooses a suitable packing method. If the default initialization process does not suit your application and you want more precise control over the initialization process, these alternatives are available:

- Choosing the packing algorithm
- Overriding the default copy-initialize function
- Manual initialization
- Initializing code—copying ROM to RAM.

For information about the performed initializations, inspect the log file (the command line option `--log initialization`).

Choosing packing algorithm

To override the default packing algorithm, write for example:

```
initialize by copy with packing = lzw { readwrite };
```

To read more about the available packing algorithms, see *Initialize directive*, page 305.

Overriding default copy-initialize function

To override the default function that copies the initializers to the RAM memory, supply the `copy routine` parameter to the `initialize by copy` directive. Your function will be called at program start as many times as needed. This can be useful when special code is required for the copy.

This example shows how it can look in the linker configuration file:

```
/* Initialize special sections */
initialize by copy with packing = none, copy routine =
                   myInitializers { section .special };
place in RAM { section .special };
place in ROM { section .special_init };
```

Your routine should look like this:

```
void myInitializers(char *dst,
                    char const *src,
                    unsigned long size);
```

See the system startup code for an exact type definition.

Manual initialization

The `initialize manually` directive lets you take complete control over initialization. For each involved section, ILINK creates an extra section that contains the initialization data, but makes no arrangements for the actual copying. This directive is, for example, useful for overlays:

```
/* Sections MYOVERLAY1 and MYOVERLAY2 will be overlaid in
MyOverlay */
define overlay MyOverlay { section MYOVERLAY1 };
define overlay MyOverlay { section MYOVERLAY2 };

/* Split the overlay sections but without initialization during
system startup */
initialize manually { section MYOVERLAY* };

/* Place the initializer sections in a block each */
define block MyOverlay1InRom { section MYOVERLAY1_init };
define block MyOverlay2InRom { section MYOVERLAY2_init };

/* Place the overlay and the initializers for it */
place in RAM { overlay MyOverlay };
place in ROM { block MyOverlay1InRom, block MyOverlay2InRom };
```

The application can then start a specific overlay by copying, as in this case, ROM to RAM:

```
#include <string.h>


/* Declare the sections. */

#pragma section = "MyOverlay"
#pragma section = "MyOverlay1InRom"
```

```
/* Function that switches in image 1 into the overlay. */

void SwitchToOverlay1()
{
  char *targetAddr     = __section_begin("MyOverlay");
  char *sourceAddr     = __section_begin("MyOverlay1InRom");
  char *sourceAddrEnd  = __section_end("MyOverlay1InRom");
  int size = sourceAddrEnd - sourceAddr;

  memcpy(targetAddr, sourceAddrEnd, size);
}
```

Initializing code—copying ROM to RAM

Sometimes, an application copies pieces of code from flash/ROM to RAM. This can be easily achieved by ILINK for whole code regions. However, for individual functions, the `__ramfunc` keyword can be used, see *Execution in RAM*, page 30

List the code sections that should be initialized in an `initialize` directive and then place the initializer and initialized sections in ROM and RAM, respectively.

In the linker configuration file, it can look like this:

```
/* Split the RAMCODE section into a readonly and a readwrite
section */
initialize by copy { section RAMCODE };

/* Place both in a block */
define block RamCode { section RAMCODE }
define block RamCodeInit { section RAMCODE_init };

/* Place them in ROM and RAM */
place in ROM { block RamCodeInit };
place in RAM { block RamCode };
```

The block definitions makes it possible to refer to the start and end of the blocks from the application.

For more examples, see *Interaction between the tools and your application*, page 117.

Running all code from RAM

If you want to copy the entire application from ROM to RAM at program startup, use the `initilize by copy` directive, for example:

```
initialize by copy { readonly, readwrite }
```

The `readwrite` pattern will match all statically initialized variables and arrange for them to be initialized at startup. The `readonly` pattern will do the same for all read-only code and data, except for code and data needed for the initialization.

To reduce the ROM space that is needed, it might be useful to compress the data with one of the available packing algorithms. For example,

```
initialize by copy { readonly, readwrite } with packing lzw
```

To read more about the available compression algorithms, see *Initialize directive*, page 305.

Because the function `__low_level_init`, if present, is called before initialization, it, and anything it needs, will not be copied from ROM to RAM either. In some circumstances—for example, if the ROM contents are no longer available to the program after startup—you might need to avoid using the same functions during startup and in the rest of the code.

If anything else should not be copied, include it in an `except` clause. This can apply to, for example, the interrupt vector table.

It is also recommended to exclude the C++ dynamic initialization table from being copied to RAM, as it is typically only read once and then never referenced again. For example, like this:

```
initialize by copy { readonly, readwrite }
            except { section .intvec,      /* Don't copy
                                              interrupt table */
                     section .init_array } /* Don't copy
                                              C++ init table */
```

INTERACTION BETWEEN ILINK AND THE APPLICATION

ILINK provides the command line options `--config_def` and `--define_symbol` to define symbols which can be used for controlling the application. You can also use symbols to represent the start and end of a continuous memory area that is defined in the linker configuration file. For more details, see *Interaction between the tools and your application*, page 117.

To change a reference to one symbol to another symbol, use the ILINK command line option `--redirect`. This is useful, for example, to redirect a reference from a non-implemented function to a stub function, or to choose one of several different implementations of a certain function, for example, how to choose the DLIB formatter for the standard library functions `printf` and `scanf`.

The compiler generates mangled names to represent complex C/C++ symbols. If you want to refer to these symbols from assembler source code, you must use the mangled names.

For information about the addresses and sizes of all global (statically linked) symbols, inspect the entry list in the map file (the command line option `--map`).

For more information, see *Interaction between the tools and your application*, page 117.

STANDARD LIBRARY HANDLING

By default, ILINK determines automatically which variant of the standard library to include during linking. The decision is based on the sum of the runtime attributes available in each object file and the library options passed to ILINK.

To disable the automatic inclusion of the library, use the option `--no_library_search`. In this case, you must explicitly specify every library file to be included. For information about available library files, see *Using a prebuilt library*, page 64.

PRODUCING OTHER OUTPUT FORMATS THAN ELF/DWARF

ILINK can only produce an output file in the ELF/DWARF format. To convert that format into a format suitable for programming PROM/flash, use `ielftool`.

VENEERS

The ARM cores need to use veneers on two occasions:

- When calling an ARM function from Thumb mode or vice versa; the veneer then changes the state of the microprocessor. If the core supports the `BLX` instruction, a veneer is not needed for changing modes.
- When calling a function that it cannot normally reach; the veneer introduces code which makes the call successfully reach the destination.

Code for veneers can be inserted between any caller and called function. As a result, the `R12` register must be treated as a scratch register at function calls, including functions written in assembler. This also applies to jumps.

For more information, see *--no_veneers*, page 202.

Hints for troubleshooting

ILINK has several features that can help you manage code and data placement correctly, for example:

- Messages at link time, for examples when a relocation error occurs
- The `--log` option that makes ILINK log information to `stdout`, which can be useful to understand why an executable image became the way it is, see *--log*, page 199

- The `--map` option that makes ILINK produce a memory map file, which contains the result of the linker configuration file, see *--map*, page 200.

RELOCATION ERRORS

For each instruction that cannot be relocated correctly, ILINK will generate a *relocation error*. This can occur for instructions where the target is out of reach or is of an incompatible type, or for many other reasons.

A relocation error produced by ILINK can look like this:

```
Error[Lp002]: relocation failed: out of range or illegal value
    Kind     :   R_XXX_YYY[0x1]
    Location :   0x40000448
                 "myfunc" + 0x2c
                 Module:  somecode.o
                 Section: 7 (.text)
                 Offset:  0x2c
    Destination: 0x9000000c
                 "read"
                 Module:  read.o(iolib.a)
                 Section: 6 (.text)
                 Offset:  0x0
```

The message entries are described in this table:

Message entry	Description
`Kind`	The relocation directive that failed. The directive depends on the instruction used.
`Location`	The location where the problem occurred, described with the following details: • The instruction address, expressed both as a hexadecimal value and as a label with an offset. In this example, `0x40000448` and `"myfunc" + 0x2c`. • The module, and the file. In this example, the module `somecode.o`. • The section number and section name. In this example, section number `7` with the name `.text`. • The offset, specified in number of bytes, in the section. In this example, `0x2c`.

Table 5: Description of a relocation error

Message entry	Description
`Destination`	The target of the instruction, described with the following details: • The instruction address, expressed both as a hexadecimal value and as a label with an offset. In this example, `0x9000000c` and `"read"` (thus, no offset). • The module, and when applicable the library. In this example, the module `read.o` and the library `iolib.a`. • The section number and section name. In this example, section number `6` with the name `.text`. • The offset, specified in number of bytes, in the section. In this example, `0x0`.

Table 5: Description of a relocation error (Continued)

Possible solutions

In this case, the distance from the instruction in `getchar` to `__read` is too long for the branch instruction.

Possible solutions include ensuring that the two `.text` sections are allocated closer to each other or using some other calling mechanism that can reach the required distance. It is also possible that the referring function tried to refer to the wrong target and that this caused the range error.

Different range errors have different solutions. Usually, the solution is a variant of the ones presented above, in other words modifying either the code or the section placement.

The DLIB runtime environment

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

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

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

Introduction to the runtime environment

The runtime environment is the environment in which your application executes. The runtime environment depends on the target hardware, the software environment, and the application code. The IAR DLIB runtime environment can be used as is together with the 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.

For information about AEABI compliance, see *AEABI compliance*, page 123.

RUNTIME ENVIRONMENT FUNCTIONALITY

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

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

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

- Support for hardware features:
  - Direct access to low-level processor operations by means of *intrinsic* functions, such as functions for register handling
  - Peripheral unit registers and interrupt definitions in include files
  - 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.

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

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

LIBRARY SELECTION

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

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

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

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. ILINK will automatically choose a prebuilt library suitable for the application.

SITUATIONS THAT REQUIRE LIBRARY BUILDING

Building a customized library is complex. Therefore, consider carefully whether it is really necessary.

You must build your own library when:

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

For information about how to build a customized library, see *Building and using a customized library*, page 71.

LIBRARY CONFIGURATIONS

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

These DLIB library configurations are available:

Library configuration	Description
Normal DLIB	No locale interface, C locale, no file descriptor support, no multibyte characters in `printf` and `scanf`, and no hexadecimal floating-point numbers in `strtod`.
Full DLIB	Full locale interface, C locale, file descriptor support, multibyte characters in `printf` and `scanf`, and hexadecimal floating-point numbers in `strtod`.

Table 6: Library configurations

You can also 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 71.

The prebuilt libraries are based on the default configurations, see *Using a prebuilt library*, page 64. There is also a ready-made library project template that you can use if you want to rebuild the runtime library.

LOW-LEVEL INTERFACE FOR DEBUG SUPPORT

If your application uses the DLIB low-level interface (see *C-SPY runtime interface*, page 86), you must implement support for the parts used by the application. However, if you must debug your application before this is implemented, you can temporarily use the semihosted debug support also provided as a library.

The low-level debugger runtime interface provided by DLIB is compatible with the semihosting interface provided by ARM Limited. The interface is implemented by a set of SVC (SuperVisor Call) instructions that generate exceptions from program control. The application invokes the appropriate semihosting call and the debugger then handles the exception. The debugger provides the required communication with the host computer.

If you build your application project with the ILINK option **Semihosted** (`--semihosting`) or **IAR breakpoint** (`--semihosting=iar_breakpoint`), certain functions in the library are replaced by functions that communicate with the debugger.

To set linker options for debug support in the IDE, choose **Project>Options** and select the **General Options** category. On the **Library configuration** page, select the **Semihosted** option or the **IAR breakpoint** option.

Using a prebuilt library

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

- Architecture
- CPU mode
- Interworking
- Library configuration—Normal or Full
- Floating-point implementation.

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

If you build your application from the command line, you must specify the library configuration file for the compiler, either `DLib_Config_Full.h` or `DLib_Config_Normal.h`, for example:

```
--dlib_config C:\...\DLib_Config_Normal.h
```

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

GROUPS OF LIBRARY FILES

The libraries are delivered in three groups of library functions:

- C/C++ standard library functions

  These are the functions defined by the ISO/ANSI C/C++ standard, for example functions like `printf` and `scanf`.
- Runtime support functions

  These are functions for system startup, initialization, floating-point arithmetics, ABI support, and some of the functions part of the ISO/ANSI C/C++ standard.
- Debug support functions

  These are functions for debug support for the semihosting interface.

Library filename syntax

The names of the libraries are constructed by the following constituents:

- <*architecture*> is the name of the architecture. It can be one of `4t`, `5E`, `6M`, or `7M` for the ARM architectures v4T, v5TE, v6M, or v7M, respectively. Libraries built for the v5TE architecture are also used for the v6 architecture.
- <*cpu_mode*> is one of `t` or `a`, for Thumb and ARM, respectively.
- <*endian*> is one of `l` or `b`, for little-endian and big-endian, respectively.
- <*fp_implementation*> is _ when the library is compiled without VFP support, that is, software implementation compliant to AAPCS. It is `s` when the library is compiled with VFP support and compliant to AAPCS/STD. It is `v` when compiled with VFP support and using VFP registers in function calls; this is not AEABI compliant. The supported version of VFP is v1 when architecture is 4t and v2 when architecture is 5E.
- <*interworking*> is `i` when the library contains interworking code, otherwise it is `_`.
- <*library_config*> is one of `n` or `f` for normal and full, respectively.
- <*debug_interface*> is one of `s`, `b` or `i`, for the SWI/SVC mechanism, the BKPT mechanism, and the IAR-specific breakpoint mechanism, respectively. For more information, see *--semihosting*, page 205.

Library files for C/C++ standard library functions

The names of the library files are constructed in the following way:

```
dl<architecture>_<cpu_mode><endian><fp_implementation>
<interworking><library_config>.a
```

which more specifically means

```
dl<4t|5E|6M|7M>_<a|t><l|b><_|s|v><i|_><n|f>.a
```

Library files for runtime support functions

The names of the library files are constructed in the following way:

```
rt<architecture>_<cpu_mode><endian><fp_implementation>.a
```

which more specifically means

```
rt<4t|5E|6M|7M>_<a|t><l|b><_|s|v>.a
```

Library files for debug support functions

The names of the library files are constructed in the following way:

```
sh<debug_interface>_<endian>.a
```

which more specifically means

```
sh<s|b|i>_<l|b>.a
```

CUSTOMIZING A PREBUILT LIBRARY WITHOUT REBUILDING

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

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

These items can be customized:

Items that can be customized	Described in
Formatters for printf and scanf	*Choosing formatters for printf and scanf*, page 67
Startup and termination code	*System startup and termination*, page 72
Low-level input and output	*Standard streams for input and output*, page 77
File input and output	*File input and output*, page 80
Low-level environment functions	*Environment interaction*, page 83
Low-level signal functions	*Signal and raise*, page 84
Low-level time functions	*Time*, page 85

Table 7: Customizable items

Items that can be customized	Described in
Size of heaps, stacks, and sections	*Stack considerations*, page 115 *Heap considerations*, page 117 *Placing code and data—the linker configuration file*, page 42

Table 7: Customizable items (Continued)

For a description about how to override library modules, see *Overriding library modules*, page 69.

Choosing formatters for printf and scanf

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

Note: If you rebuild the library, it is possible to optimize these functions even further, see *Configuration symbols for printf and scanf*, page 79.

CHOOSING PRINTF FORMATTER

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

This table summarizes the capabilities of the different formatters:

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

Table 8: Formatters for printf

† **Depends on the library configuration that is used.**

For information about how to fine-tune the formatting capabilities even further, see *Configuration symbols for printf and scanf*, page 79.

Specifying the print formatter in the IDE

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

Specifying printf formatter from the command line

To use any other formatter than the default (`_PrintfFull`), add one of these ILINK command line options:

```
--redirect _Printf=_PrintfLarge
--redirect _Printf=_PrintfSmall
--redirect _Printf=_PrintfTiny
```

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.

This table summarizes the capabilities of the different formatters:

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

Table 9: Formatters for scanf

† **Depends on the library configuration that is used.**

For information about how to fine-tune the formatting capabilities even further, see *Configuration symbols for printf and scanf*, page 79.

Specifying scanf formatter in the IDE

To use any other formatter than the default (Full), 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 these ILINK command line options:

```
--redirect _Scanf=_ScanfLarge
--redirect _Scanf=_PrintfSmall
```

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 IDE

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

1. Copy the appropriate *`library_module`*`.c` file to your project directory.
2. Make the required additions to the file (or create your own routine, using the default file as a model).
3. Add the customized file to your project.
4. Rebuild your project.

Overriding library modules from the command line

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

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

   ```
   iccarm library_module.c
   ```

   This creates a replacement object module file named *`library_module`*`.o`.
4. Add *`library_module`*`.o` to the ILINK command line, either directly or by using an extended linker command file, for example:

   ```
   ilinkarm library_module.o
   ```

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

   Run ILINK to rebuild your application.

   This will use your version of *`library_module`*`.o`, instead of the one in the library. For information about the ILINK options, see the chapter *Linker options*.

Building and using a customized library

In some situations, see *Situations that require library building*, page 63, it is necessary to rebuild the C/C++ standard library. In those cases you must:

- 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.

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

For information about the build process and the IAR Command Line Build Utility, see the *IAR Embedded Workbench® IDE User Guide for ARM®*.

SETTING UP A LIBRARY PROJECT

The IDE provides a library project template which can be used for customizing the runtime environment configuration. This library template has Full library configuration, see Table 6, *Library configurations*, page 63.

In the IDE, modify the generic options in the created library project to suit your application, see *Basic project configuration*, page 19.

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

MODIFYING THE LIBRARY FUNCTIONALITY

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

The library functionality is determined by a set of *configuration symbols*. The default values of these symbols are defined in the file `DLib_Defaults.h`. This read-only file describes the configuration possibilities. In addition, your library must have its own library configuration file based on either `DLib_Config_Normal.h` or `DLib_Config_Full.h`, which sets up that specific library with the required library configuration. For more information, see Table 7, *Customizable items*, page 66.

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

Modifying the library configuration file

In your library project, open the file `DLib_Config_Normal.h` or `DLib_Config_Full.h`, depending on your library, make a copy of the 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 build your library, you must make sure to use it in your application project.

In the IDE you must do these 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 **Configuration file** text box, locate your library configuration file.

4 Click the **Library** tab, also in the **Linker** category. Use the **Additional libraries** text box to locate your library file.

System startup and termination

This section describes the runtime environment actions performed during startup and termination of your application.

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

For Cortex-M, one of the following files is used instead of `cstartup.s`:

`thumb\cstartup_M.s` or `thumb\cstartup_M.c`

For information about how to customize the system startup code, see *Customizing system initialization*, page 76.

SYSTEM STARTUP

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

For the hardware initialization, it looks like this:

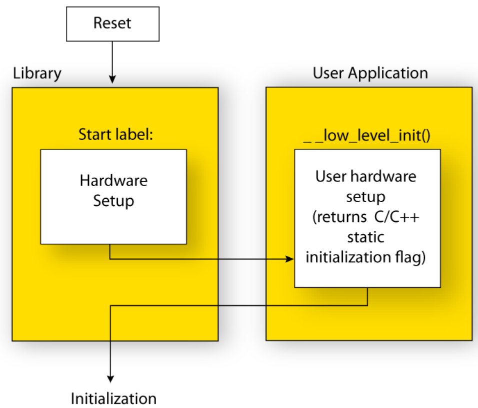


Figure 10: Target hardware initialization phase

- When the CPU is reset it will jump to the program entry label `__iar_program_start` in the system startup code.
- Exception stack pointers are initialized to the end of each corresponding section
- The stack pointer is initialized to the end of the `CSTACK` block
- The function `__low_level_init` is called if you defined it, giving the application a chance to perform early initializations.

Note: For Cortex-M devices, the second bullet in the above list is not valid. The first and the third bullets are handled slightly differently. At reset, a Cortex-M CPU initializes `PC` and `SP` from the vector table (`__vector_table`), which is defined in the `cstartup_M.c` file.

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

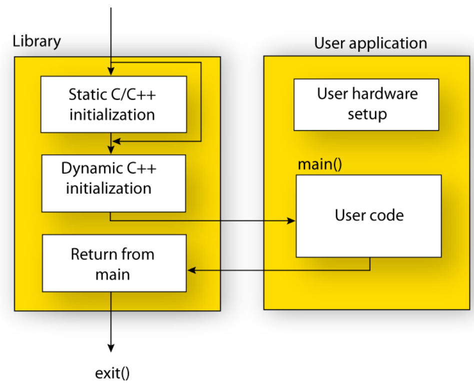


Figure 11: C/C++ initialization phase

- Static and global variables are initialized. That is, zero-initialized variables are cleared and the values of other initialized variables are copied from ROM to RAM memory. This step is skipped if `__low_level_init` returns zero. For more details, see *Initialization at system startup*, page 45
- Static C++ objects are constructed
- The `main` function is called, which starts the application.

For an overview of the initialization phase, see *Application execution—an overview*, page 14.

SYSTEM TERMINATION

This illustration shows the different ways an embedded application can terminate in a controlled way:

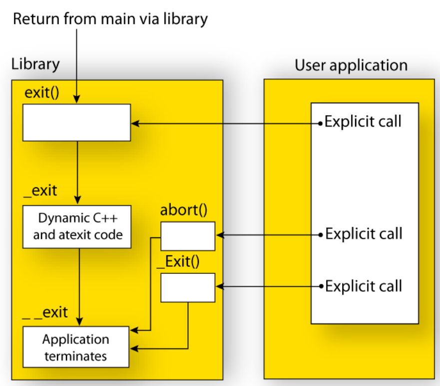


Figure 12: System termination phase

An application can terminate normally in two different ways:

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

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

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

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

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

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

C-SPY interface to system termination

If your project is linked with the semihosted interface, 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 runtime interface*, page 86.

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 sections performed by `cstartup`.

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

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

Note: Normally, you do not need to customize either of the files `cmain.s` or `cexit.s`.

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 71.

Note: Regardless of whether you modify the routine `__low_level_init` or the file `cstartup.s`, 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.s`. 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 sections should be initialized by the system startup code. If the function returns `0`, the data sections will not be initialized.

MODIFYING THE FILE CSTARTUP.S

As noted earlier, you should not modify the file `cstartup.s` if a customized version of `__low_level_init` is enough for your needs. However, if you do need to modify the file `cstartup.s`, 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 69.

Note that you must make sure that the linker uses the start label used in your version of `cstartup.s`. For information about how to change the start label used by the linker, see *--entry*, page 196.

For Cortex-M, you must create a modified copy of `cstartup_M.s` or `cstartup_M.c` to use interrupts or other exception handlers.

Standard streams for input and output

Standard communication channels (streams) are defined in `stdio.h`. If any of these streams are used by your application, for example by the functions `printf` and `scanf`, you must 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\lib` directory.

If you intend to rebuild the library, the source files are available in the template library project, see *Building and using a customized library*, page 71. 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 runtime interface*, page 86.

Example of using __write

The code in this example uses memory-mapped I/O to write to an LCD display:

```
#include <stddef.h>

__no_init volatile unsigned char lcdIO @ 0x1000;
```

```
size_t __write(int handle,
               const unsigned char *buf,
               size_t bufSize)
{
  size_t nChars = 0;

  /* Check for the command to flush all handles */
  if (handle == -1)
  {
    return 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)
  {
    lcdIO = *buf;
    ++buf;
    ++nChars;
  }

  return nChars;
}
```

Note: A call to `__write` where `buf` has the value `NULL` is a command to flush the handle. When the handle is `-1`, all streams should be flushed.

Example of using __read

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

```
#include <stddef.h>

__no_init volatile unsigned char kbIO @ 0x1000;

size_t __read(int handle,
              unsigned char *buf,
              size_t bufSize)
{
  size_t nChars = 0;

  /* Check for stdin
```

```
    (only necessary if FILE descriptors are enabled) */
  if (handle != 0)
  {
    return -1;
  }

  for (/*Empty*/; bufSize > 0; --bufSize)
  {
    unsigned char c = kbIO;
    if (c == 0)
      break;

    *buf++ = c;
    ++nChars;
  }

  return nChars;
}
```

For information about the `@` operator, see *Controlling data and function placement in memory*, page 131.

Configuration symbols for printf and scanf

When you set up your application project, you typically need to consider what `printf` and `scanf` formatting capabilities your application requires, see *Choosing formatters for printf and scanf*, page 67.

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

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

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

Printf configuration symbols	Includes support for
`_DLIB_PRINTF_MULTIBYTE`	Multibyte characters
`_DLIB_PRINTF_LONG_LONG`	Long long (`ll` qualifier)
`_DLIB_PRINTF_SPECIFIER_FLOAT`	Floating-point numbers
`_DLIB_PRINTF_SPECIFIER_A`	Hexadecimal floating-point numbers
`_DLIB_PRINTF_SPECIFIER_N`	Output count (`%n`)

Table 10: Descriptions of printf configuration symbols

Printf configuration symbols	Includes support for
_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 10: Descriptions of printf configuration symbols (Continued)

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

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

Table 11: Descriptions of scanf configuration symbols

CUSTOMIZING FORMATTING CAPABILITIES

To customize the formatting capabilities, you must set up a library project, see *Building and using a customized library*, page 71. 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 must customize them to suit your hardware. To simplify adaptation to specific hardware, all I/O functions call a small set of primitive functions, each designed to accomplish one particular task; for example, `__open` opens a file, and `__write` outputs characters.

Note that file I/O capability in the library is only supported by libraries with full library configuration, see *Library configurations*, page 63. 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 these I/O files are included in the product:

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

Table 12: Low-level I/O files

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

Note: If you link your library with I/O debugging support, C-SPY variants of the low-level I/O functions are linked for interaction with C-SPY. For more information, see *Low-level interface for debug support*, page 64.

Locale

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

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

The DLIB library can be used in two main modes:

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

LOCALE SUPPORT IN PREBUILT LIBRARIES

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

- All prebuilt libraries support the C locale only

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

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

CUSTOMIZING THE LOCALE SUPPORT

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

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

Locale configuration symbols

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

```
#define _DLIB_FULL_LOCALE_SUPPORT 1
#define _LOCALE_USE_C        /* C locale */
#define _LOCALE_USE_EN_US    /* American English */
#define _LOCALE_USE_EN_GB    /* British English */
#define _LOCALE_USE_SV_SE    /* Swedish in Sweden */
```

See `DLib_Defaults.h` for a list of supported locale and encoding settings.

If you want to customize the locale support, you simply define the locale configuration symbols required by your application. For more information, see *Building and using a customized library*, page 71.

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

Building a library without support for locale interface

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

Building a library with support for locale interface

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

CHANGING LOCALES AT RUNTIME

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

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

`lang_REGION`

or

`lang_REGION.encoding`

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

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

Example

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

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

Environment interaction

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

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

THE GETENV FUNCTION

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
```

End the string with an extra null character (if you use a C string, this is added automatically). 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 69.

THE SYSTEM FUNCTION

If you need to use the `system` function, you must 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 71.

Note: If you link your application with support for I/O debugging, the functions `getenv` and `system` are replaced by C-SPY variants. For further information, see *Low-level interface for debug support*, page 64.

Signal and raise

Default implementations of the functions `signal` and `raise` are 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\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 69.

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 71.

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 69.

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 71.

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

Note: If you link your application with support for I/O debugging, the functions `clock` and `time` are replaced by C-SPY variants that return the host clock and time respectively. For further information, see *C-SPY runtime interface*, page 86.

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 must rebuild the library, see *Building and using a customized library*, page 71. Enable the configuration symbol `_DLIB_STRTOD_HEX_FLOAT` in the library configuration file.

Assert

If you linked your application with support for runtime debugging, an assert will print a message on `stdout`. If this is not the behavior you require, you must add the source file `xreportassert.c` to your application project. 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 71. To turn off assertions, you must define the symbol `NDEBUG`.

In the IDE, this symbol `NDEBUG` is by default defined in a Release project and *not* defined in a Debug project. If you build from the command line, you must explicitly define the symbol according to your needs. See *NDEBUG*, page 287.

Atexit

The linker allocates a static memory area for `atexit` function calls. By default, the number of calls to the `atexit` function are limited to 32 bytes. To change this limit, see *Setting up the atexit limit*, page 53.

C-SPY runtime interface

To include support for runtime and I/O debugging, you must link your application with the option **Semihosted** or **IAR breakpoint**, see *Low-level interface for debug support*, page 64.

In this case, special debugger variants of these library functions are linked to the application:

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

Table 13: Functions with special meanings when linked with debug info

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 are 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 DEBUGGER TERMINAL I/O WINDOW

To make the Terminal I/O window available, the application must be linked with support for I/O debugging, see *Low-level interface for debug support*, page 64. This means that when the functions `__read` or `__write` are called to perform I/O operations on the streams `stdin`, `stdout`, or `stderr`, data will be sent to or read from the C-SPY Terminal I/O window.

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

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

Speeding up terminal output

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

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

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

```
--redirect __write=__write_buffered
```

Checking module consistency

This section introduces the concept of runtime model attributes, a mechanism that you can use to ensure that modules are built using compatible settings.

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

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

RUNTIME MODEL ATTRIBUTES

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

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

Note: For IAR predefined runtime model attributes, the linker uses several ways of checking them.

Example

In this table, the object files could (but do not have to) define the two runtime attributes `color` and `taste`:

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

Table 14: Example of runtime model attributes

In this case, `file1` cannot be linked with any of the other files, since the runtime attribute `color` does not match. Also, `file4` and `file5` 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.

USING RUNTIME MODEL ATTRIBUTES

To ensure module consistency with other object files, use the `#pragma rtmodel` directive to specify runtime model attributes in your C/C++ source code. For example:

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

For detailed syntax information, see *rtmodel*, page 255.

You can also use the `rtmodel` assembler directive to specify runtime model attributes in your assembler source code. For example:

```
rtmodel "color", "red"
```

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

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

Assembler language interface

When you develop an application for an embedded system, there might 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 and some C alternatives, with their advantages and disadvantages. It also describes how to write functions in assembler language that work together with an application written in C or C++.

Finally, the chapter covers how functions are called, 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 IAR C/C++ Compiler for ARM provides several ways to access low-level resources:

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

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

INTRINSIC FUNCTIONS

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

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

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

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

MIXING C AND ASSEMBLER MODULES

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

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

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

An important advantage is that you will have a well-defined interface between what the compiler produces and what you write in assembler. When using inline assembler, you will not have any guarantees that your inline assembler lines do not interfere with the compiler generated code.

When an application is written partly in assembler language and partly in C or C++, you are faced with several 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 94. The following two are covered in the section *Calling convention*, page 97.

The section *Inline assembler*, page 93, covers how to use inline assembler, but it also shows how data in memory is accessed.

The answer to the final question is that the call stack can be displayed when you run assembler code in the debugger. However, the debugger requires information about the *call frame*, which must be supplied as annotations in the assembler source file. For more information, see *Call frame information*, page 103.

The recommended method for mixing C or C++ and assembler modules is described in *Calling assembler routines from C*, page 94, and *Calling assembler routines from C++*, page 96, respectively.

INLINE ASSEMBLER

It is possible to insert assembler code directly into a C or C++ function. The `asm` keyword inserts the supplied assembler statement in-line, see *Inline assembler*, page 225 for reference information. The following example demonstrates the use of the `asm` keyword. This example also shows the risks of using inline assembler.

```
bool flag;

void foo()
{
  while (!flag)
  {
    asm("   ldr r2,[pc,#0]   \n" /* r2 = address of flag */
        "   b .+8            \n" /* jump over constant   */
        "   DCD flag         \n" /* address of flag      */
        "   ldr r3,[pc,#0]   \n" /* r3 = address of PIND */
        "   b .+8            \n" /* jump over constant   */
        "   DCD PIND         \n" /* address of PIND      */
        "   ldr r0,[r3]      \n" /* r0 = PIND            */
        "   str r0,[r2]");       /* flag = r0            */
  }
}
```

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

The inline assembler instruction will simply be inserted at the given location in the program flow. The consequences or side-effects the insertion might have on the surrounding code are not taken into consideration. If, for example, registers or memory locations are altered, they might 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. There are also several limitations to using inline assembler:

- The compiler's various optimizations will disregard any effects of the inline sequences, which will not be optimized at all
- In general, assembler directives will cause errors or have no meaning. Data definition directives will however work as expected
- Alignment cannot be controlled; this means, for example, that `DC32` directives might be misaligned
- Auto variables cannot be accessed
- Alternative register names, mnemonics, and operators are not supported; read more about the `-j` assembler option in the *ARM® IAR Assembler Reference Guide.*

Inline assembler is therefore often best avoided. If no suitable intrinsic function is available, we recommend that you use 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 will 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 these examples:

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

  or

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

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

CREATING SKELETON CODE

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

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

```
extern int gInt;
extern char gChar;

int Func(int arg1, char arg2)
{
  int locInt = arg1;
  gInt = arg1;
  gChar = arg2;
  return locInt;
}

int main()
{
  int locInt = gInt;
  gInt = Func(locInt, gChar);
  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 IDE, specify list options on file level. Select the file in the workspace window. Then choose **Project>Options**. In the **C/C++ Compiler** category, select **Override inherited settings**. On the **List** page, deselect **Output list file**, and instead select the **Output assembler file** option and its suboption **Include source**. Also, be sure to specify a low level of optimization.

Use these options to compile the skeleton code:

```
iccarm skeleton.c -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 `s`. Also remember to specify a low level of optimization, and `-e` for enabling language extensions.

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

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

On the command line, use the option `-lB` instead of `-lA`. Note that `CFI` information must be included in the source code to make the C-SPY Call Stack window work.

The output file

The output file contains the following important information:

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

The `CFI` directives describe the call frame information needed by the Call Stack window in the debugger. For more information, see *Call frame information*, page 103.

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 can therefore be called from C++ when declared in this manner:

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

The following example shows how to achieve the equivalent to a non-static member function, which means that the implicit `this` pointer must be made explicit. It is also possible to "wrap" the call to the assembler routine in a member function. Use an inline member function to remove the overhead of the extra call—this assumes that function inlining is enabled:

```
class MyClass;

extern "C"
{
  void DoIt(MyClass *ptr, int arg);
}

class MyClass
{
public:
  inline void DoIt(int arg)
  {
    ::DoIt(this, arg);
  }
};
```

Calling convention

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

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

This section describes the calling conventions used by the compiler. These items are examined:

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

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

Unless otherwise noted, the calling convention used by the compiler adheres to AAPCS, a part of AEABI; see *AEABI compliance*, page 123.

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

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

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

USING C LINKAGE IN C++ SOURCE CODE

In C++, a function can have either C or C++ linkage. To call assembler routines from C++, it is easiest if you make the C++ function have C linkage.

This is an example of a declaration of a function with C linkage:

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

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

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

int F(int);

#ifdef __cplusplus
}
#endif
```

PRESERVED VERSUS SCRATCH REGISTERS

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

Scratch registers

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

Any of the registers `R0` to `R3`, and `R12`, can be used as a scratch register by the function. Note that `R12` is a scratch register also when calling between assembler functions only because of automatically inserted instructions for veneers.

Preserved registers

Preserved registers, on the other hand, are preserved across function calls. The called function can use the register for other purposes, but must save the value before using the register 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

For some registers, you must consider certain prerequisites:

- 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, can be destroyed. At function entry and exit, the stack pointer must be 8-byte aligned. In the function, the stack pointer must always be word aligned. At exit, `SP` must have the same value as it had at the entry.
- 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 one of two basic methods: in registers or on the stack. It is much more efficient to use registers than to take a detour via memory, so the calling convention is designed to use registers as much as possible. Only a limited number of registers can be used for passing parameters; when no more registers are available, the remaining parameters are passed on the stack. These exceptions to the rules apply:

- Interrupt functions cannot take any parameters, except software interrupt functions that accept parameters and have return values
- Software interrupt functions cannot use the stack in the same way as ordinary functions. When an `SVC` instruction is executed, the processor switches to supervisor mode where the supervisor stack is used. Arguments can therefore not be

passed on the stack if your application is not running in supervisor mode previous to the interrupt.

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). For more information, see *Calling assembler routines from C++*, page 96.

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`
`long long` and double-precision (64-bit) values	Passed in first available register pair: `R0:R1`, or `R2:R3`

Table 15: 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`.

Stack parameters and 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.

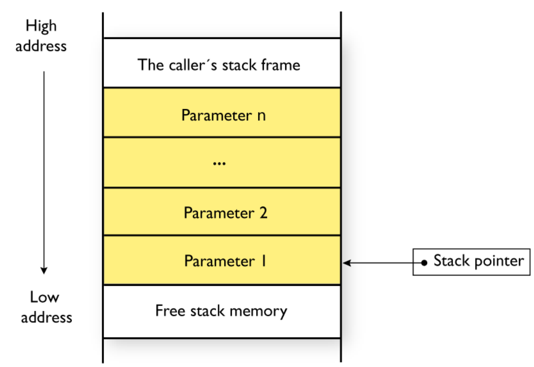


Figure 13: Storing stack parameters in memory

The stack should be aligned to 8 at function entry.

FUNCTION EXIT

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

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

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 16: Registers used for returning values

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

Stack layout at function exit

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:

```
PUSH        {R4-R6,LR}      /* Function entry. */
 .
 .
 .
POP        {R4-R6,PC}      /* Function exit. */
```

EXAMPLES

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

Example 1

Assume this 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`.

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

```
  add1:
    ADDS    R0,R0,#+0x1
    BX      BX,LR
```

Example 2

This example shows how structures are passed on the stack. Assume these declarations:

```
struct MyStruct
{
  int a,b,c,d,e;
};

int MyFunction(struct MyStruct 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`.

Call frame information

When you debug an application using C-SPY, you can view the *call stack*, that is, the chain of functions that called the current function. To make this possible, the compiler supplies 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*.

CFI DIRECTIVES

The `CFI` directives 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 might 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.

This 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

Table 17: Call frame information resources defined in a names block

CREATING ASSEMBLER SOURCE WITH CFI SUPPORT

The recommended way to create an assembler language routine that handles call frame information correctly is to start with an assembler language source file created by the compiler.

1 Start with suitable C source code, for example:

```
int F(int);
int cfiExample(int i)
{
  return i + F(i);
}
```

2 Compile the C source code, and make sure to create a list file that contains call frame information—the CFI directives.

On the command line, use the option `-lA`.

In the IDE, choose **Project>Options>C/C++ Compiler>List** and make sure the suboption **Include call frame information** is selected.

For the source code in this example, the list file looks like this:

```
        NAME cfiexample

        EXTERN F

        PUBLIC cfiExample
```

```
        CFI Names cfiNames0
        CFI StackFrame CFA R13 DATA
        CFI Resource R0:32, R1:32, R2:32, R3:32, R4:32, R5:32,
R6:32, R7:32
        CFI Resource R8:32, R9:32, R10:32, R11:32, R12:32,
R13:32, R14:32
        CFI EndNames cfiNames0

        CFI Common cfiCommon0 Using cfiNames0
        CFI CodeAlign 4
        CFI DataAlign 4
        CFI ReturnAddress R14 CODE
        CFI CFA R13+0
        CFI R0 Undefined
        CFI R1 Undefined
        CFI R2 Undefined
        CFI R3 Undefined
        CFI R4 SameValue
        CFI R5 SameValue
        CFI R6 SameValue
        CFI R7 SameValue
        CFI R8 SameValue
        CFI R9 SameValue
        CFI R10 SameValue
        CFI R11 SameValue
        CFI R12 Undefined
        CFI R14 SameValue
        CFI EndCommon cfiCommon0


        SECTION `.text`:CODE:NOROOT(2)
        CFI Block cfiBlock0 Using cfiCommon0
        CFI Function cfiExample
        ARM
cfiExample:
        PUSH     {R4,LR}
        CFI R14 Frame(CFA, -4)
        CFI R4 Frame(CFA, -8)
        CFI CFA R13+8
        MOVS     R4,R0
        MOVS     R0,R4
        BL       F
        ADDS     R0,R0,R4
        POP      {R4,LR}
        CFI R4 SameValue
        CFI R14 SameValue
        CFI CFA R13+0
```

```
BX       LR                ;; return
CFI EndBlock cfiBlock0

END
```

Note: The header file `cfiCommon.i` contains the macros `CFI_NAME_BLOCK`, `CFI_COMMON_ARM`, and `CFI_COMMON_Thumb` which declare a typical names block and two typical common blocks. These macros declare several resources, both concrete and virtual.

Using C++

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

Overview

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

STANDARD EMBEDDED C++

The following C++ features are supported:

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

C++ features that are excluded are those that introduce overhead in execution time or code size that are beyond the control of the programmer. Also excluded are late additions to the ISO/ANSI C++ standard. This is because they represent potential portability problems, due to 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 these features of C++:

- Templates
- Multiple and virtual inheritance
- Exception handling

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

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

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

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

EXTENDED EMBEDDED C++

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

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

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

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

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

ENABLING C++ SUPPORT

In the 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 169.

To take advantage of *Extended* Embedded C++ features in your source code, you must use the `--eec++` compiler option. See *--eec++*, page 169.

To set the equivalent option in the IDE, choose **Project>Options>C/C++ Compiler>Language**.

Feature descriptions

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

CLASSES

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

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

The location operator `@` can be used on static data members and on any type of function members.

For further information about attributes, see *Type qualifiers*, page 218.

Example

```
class MyClass
{
public:
  // Locate a static variable at address 60
  static __no_init int mI @ 60;

  // A static Thumb function
  static __thumb void f();

  // A Thumb function
  __thumb void g();
```

```
  // Interworking assumed
  virtual __thumb void th()

  // Interworking assumed
  virtual __arm void ah();

  // Locate a virtual function into SPECIAL
  virtual void M() const volatile @ "SPECIAL";
};
```

FUNCTION TYPES

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

Example

```
extern "C"
{
  typedef void (*FpC)(void);    // A C function typedef
}

typedef void (*FpCpp)(void);    // A C++ function typedef

FpC F1;
FpCpp F2;
void MyF(FpC);

void MyG()
{
  MyF(F1);                      // Always works
  MyF(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` must 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 must 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 108.

STL and the IAR C-SPY® Debugger

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

To read more about displaying STL containers in the C-SPY debugger, see the *IAR Embedded Workbench® IDE User Guide for ARM®.*

VARIANTS OF CAST OPERATORS

In Extended EC++ these additional variants of C++ cast operators can be used:

```
const_cast<to>(from)
static_cast<to>(from)
reinterpret_cast<to>(from)
```

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
```

You must make sure that identifiers in your application do not interfere with identifiers in the runtime library.

POINTER TO MEMBER FUNCTIONS

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

Example

```
class X
{
public:
  __interwork void aF();
};

void (__interwork   X::*ap)() = &X::af;
```

USING INTERRUPTS AND EC++ DESTRUCTORS

If interrupts are enabled and the interrupt functions use static class objects that need to be destroyed (using destructors), there might be problems if the interrupt occur during or after application exits. If an interrupt occurs after the static class object was destroyed, the application will not work properly.

To avoid this, make sure that interrupts are disabled when returning from `main` or when calling `exit` or `abort`. To do this, call the intrinsic function `__disable_interrupt`.

C++ language extensions

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

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

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

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

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

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

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

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

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

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

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

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

```
bool X;

char *P1 = X ? "abc" : "def";       //Possible when using IAR
                                    //language extensions
char const *P2 = X ? "abc" : "def"; //According to standard
```

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

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

Application-related considerations

This chapter discusses a selected range of application issues related to developing your embedded application.

Typically, this chapter highlights issues that are not specifically related to only the compiler or the linker.

Output format considerations

The linker produces an absolute executable image in the ELF/DWARF object file format.

You can use the IAR ELF Tool—`ielftool`— to convert an absolute ELF image to a format more suitable for loading directly to memory, or burning to a PROM or flash memory etc.

`ielftool` can produce these output formats:

- Plain binary
- Motorola S-records
- Intel hex.

Note: `ielftool` can also be used for other types of transformations, such as filling and calculating checksums in the absolute image.

The source code for `ielftool` is provided in the `arm/src` directory. For more information about `ielftool`, see *The IAR ELF Tool—ielftool*, page 330.

Stack considerations

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

STACK SIZE CONSIDERATIONS

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

For more information about the stack size, see *Setting up the stack*, page 52, and *Saving stack space and RAM memory*, page 140.

ALIGNING THE STACK

You must make sure that the stack of your application is 8-byte aligned, because this is expected by some parts of the runtime library.

For more information about aligning the stack, see *Calling convention*, page 97 and more specifically *Special registers*, page 99 and *Stack parameters and layout*, page 101.

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

Table 18: Exception stacks

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

Cortex-M does not have individual exception stacks. By default, all exception stacks are placed in the `CSTACK` section.

To view any of these stacks in the Stack window available in the IDE, these preconfigured section names must be used instead of user-defined section names.

Heap considerations

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

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

- Linker sections used for the heap
- Allocating the heap size, see *Setting up the heap*, page 52.

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

Heap size and standard I/O

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

Interaction between the tools and your application

The linking process and the application can interact symbolically in four ways:

- Creating a symbol by using the ILINK command line option `--define_symbol`. ILINK will create a public absolute constant symbol that the application can use as a label, as a size, as setup for a debugger, et cetera.

- Creating an exported configuration symbol by using the command line option `--config_def` or the configuration directive `define symbol`, and exporting the symbol using the `export symbol` directive. ILINK will create a public absolute constant symbol that the application can use as a label, as a size, as setup for a debugger, etc.

  One advantage of this symbol definition is that this symbol can also be used in expressions in the configuration file, for example to control the placement of sections into memory ranges.
- Using the compiler operators `__section_begin`, `__section_end`, or `__section_size`, or the assembler operators `SFB`, `SFE`, or `SIZEOF` on a named section or block. These operators provide access to the start address, end address, and size of a contiguous sequence of sections with the same name, or of a linker block specified in the linker configuration file.
- The command line option `--entry` informs ILINK about the start label of the application. It is used by ILINK as a root symbol and to inform the debugger where to start execution.

The following lines illustrate how to use these mechanisms. Add these options to your command line:

```
--define_symbol NrOfElements=10
--config_def HeapSize=1024
```

The linker configuration file can look like this:

```
define memory Mem with size = 4G;
define region ROM = Mem:[from 0x00000 size 0x10000];
define region RAM = Mem:[from 0x20000 size 0x10000];

/* Export of symbol */
export symbol HeapSize;

/* Setup a heap area witha size defined by an ILINK option */
define block MyHEAP with size = HeapSize, alignment = 8 {};

place in RAM { block MyHEAP };
```

Add these lines to your application source code:

```
#include <stdlib.h>

/* Use symbol defined by ILINK option to dynamically allocate
   an array of elements with specified size */
extern char NrOfElements;
```

```
typedef long Elements;
Elements *GetElementArray()
{
  return malloc(sizeof(Elements) * (int)& NrOfElements);
}

/* Use a symbol defined by ILINK option, a symbol that in the
   configuration file was made available to the application */
extern char HeapSize;

/* Declares the section that contains our heap */
#pragma section = "MyHEAP"

char *MyHeap()
{
  /* First get start of statically allocated section */
  char *p = __section_begin("MyHEAP");

  /* then we zero it, using the imported size */
  for (int i = 0; i < (int)& HeapSize; ++i)
  {
    p[i] = 0;
  }
  return p;
}
```

Checksum calculation

The IAR ELF Tool—`ielftool`—fills specific ranges of memory with a pattern and then calculates a checksum for those ranges. The calculated checksum replaces the value of an existing symbol in the input ELF image. The application can then verify that the ranges did not change.

To use checksumming to verify the integrity of your application, you must:

- Reserve a place, with an associated name and size, for the checksum calculated by `ielftool`
- Choose a checksum algorithm, set up `ielftool` for it, and include source code for the algorithm in your application
- Decide what memory ranges to verify and set up both `ielftool` and the source code for it in your application source code.

Note: To set up `ielftool` in the IDE, choose **Project>Options>Linker>Checksum**.

CALCULATING A CHECKSUM

In this example, a checksum is calculated for ROM memory at `0x8002` up to `0x8FFF` and the 2-byte calculated checksum is placed at `0x8000`.

Creating a place for the calculated checksum

You can create a place for the calculated checksum in two ways; by creating a global C/C++ or assembler constant symbol with a proper size, residing in a specific section (in this example `.checksum`), or by using the linker option `--place_holder`.

For example, to create a 2-byte space for the symbol `__checksum` in the section `.checksum`, with alignment 4:

```
--place_holder __checksum,2,.checksum,4
```

To place the `.checksum` section, you must modify the linker configuration file. It can look like this (note the handling of the block `CHECKSUM`):

```
define memory Mem with size = 4G;

define region ROM_region = Mem:[from 0x8000 to 0x80000000 - 1];
define region RAM_region = Mem:[from 0x80000000 to 0x100000000 -2 ];

initialize by copy { rw };
do not initialize  { section .noinit };

define block HEAP      with alignment = 8, size = 16M {};
define block CSTACK    with alignment = 8, size = 16K {};
define block IRQ_STACK with alignment = 8, size = 16K {};
define block FIQ_STACK with alignment = 8, size = 16K {};

define block CHECKSUM    { ro section .checksum };
place at address Mem:0x0 { ro section .intvec};
place in ROM_region  { ro, first block CHECKSUM };
place in RAM_region { rw, block HEAP, block CSTACK, block
                       IRQ_STACK, block FIQ_STACK };
```

Running ielftool

To calculate the checksum, run `ielftool`:

```
ielftool --fill=0x00;0x8000–0x8FFF
--checksum=__checksum:2,crc16;0x8000–0x8FFF sourceFile.out
destinationFile.out
```

To calculate a checksum you also must define a fill operation. In this example, the fill pattern `0x0` is used. The checksum algorithm used is crc16.

Note that `ielftool` needs an unstripped input ELF image. If you use the `--strip` linker option, remove it and use the `--strip` ielftool option instead.

ADDING A CHECKSUM FUNCTION TO YOUR SOURCE CODE

To check the value of the `ielftool` generated checksum, it must be compared with a checksum that your application calculated. This means that you must add a function for checksum calculation (that uses the same algorithm as `ielftool`) to your application source code. Your application must also include a call to this function.

A function for checksum calculation

This function—a slow variant but with small memory footprint—uses the crc16 algorithm:

```
unsigned short slow_crc16(unsigned short sum,
                          unsigned char *p,
                          unsigned int len)
{
  while (len--)
  {
    int i;
    unsigned char byte = *(p++);

    for (i = 0; i < 8; ++i)
    {
      unsigned long oSum = sum;
      sum <<= 1;
      if (byte & 0x80)
        sum |= 1;
      if (oSum & 0x8000)
        sum ^= 0x1021;
      byte <<= 1;
    }
  }
  return sum;
}
```

You can find the source code for the checksum algorithms in the `arm\src\linker` directory of your product installation.

Checksum calculation

This code gives an example of how the checksum can be calculated:

```
/* Start and end of the checksum range */
unsigned long ChecksumStart = 0x8000+2;
unsigned long ChecksumEnd   = 0x8FFF;
```

```
/* The checksum calculated by ielftool
 * (note that it lies on address 0x8000)
 */
extern unsigned short const __checksum;


void TestChecksum()
{
  unsigned short calc = 0;
  unsigned char zeros[2] = {0, 0};

  /* Run the checksum algorithm */
  calc = slow_crc16(0,
                    (unsigned char *) ChecksumStart,
                    (ChecksumEnd - ChecksumStart+1));

  /* Rotate out the answer */
  calc = slow_crc16(calc, zeros, 2);

  /* Test the checksum */
  if (calc != __checksum)
  {
    abort();   /* Failure */
  }
}
```

THINGS TO REMEMBER

When calculating a checksum, you must remember that:

- The checksum must be calculated from the lowest to the highest address for every memory range
- Each memory range must be verified in the same order as defined
- It is OK to have several ranges for one checksum
- If several checksums are used, you should place them in sections with unique names and use unique symbol names
- If a slow function is used, you must make a final call to the checksum calculation with as many bytes (with the value `0x00`) as there are bytes in the checksum.

For more information, see also *The IAR ELF Tool—ielftool*, page 330.

C-SPY CONSIDERATIONS

By default, a symbol that you have allocated in memory by using the linker option `--place_holder` is considered by C-SPY to be of the type `int`. If the size of the checksum is less than four bytes, you can change the display format of the checksum symbol to match its size.

In the C-SPY Watch window, select the symbol and choose **Show As** from the context menu. Choose the display format that matches the size of the checksum symbol.

AEABI compliance

The IAR build tools for ARM support the Embedded Application Binary Interface for ARM, AEABI, defined by ARM Limited. This interface is based on the Intel IA64 ABI interface. The advantage of adhering to AEABI is that any such module can be linked with any other AEABI compliant module, even modules produced by tools provided by other vendors.

The IAR build tools for ARM support the following parts of the AEABI:

AAPCS	Procedure Call Standard for the ARM architecture
CPPABI	C++ ABI for the ARM architecture (EC++ parts only)
AAELF	ELF for the ARM architecture
AADWARF	DWARF for the ARM architecture
RTABI	Runtime ABI for the ARM architecture
CLIBABI	C library ABI for the ARM architecture

The IAR build tools only support a *bare metal* platform, that is a ROM-based system that lacks an explicit operating system.

Note that:

- The AEABI is specified for C89 only
- The IAR build tools only support using the default and C locales
- The AEABI does not specify C++ library compatibility
- The IAR build tools do not support the use of exceptions and `rtti`
- Neither the size of an enum or of `wchar_t` is constant in the AEABI.

If AEABI compliance is enabled, almost all optimizations performed in the system header files are turned off, and certain preprocessor constants become real constant variables instead.

LINKING AEABI COMPLIANT MODULES USING THE IAR ILINK LINKER

When building an application using the IAR ILINK Linker, the following types of modules can be combined:

- Modules produced using IAR build tools, both AEABI compliant modules as well as modules that are not AEABI compliant
- AEABI compliant modules produced using build tools from another vendor.

Note: To link a module produced by a compiler from another vendor, extra support libraries from that vendor might be required.

The IAR ILINK Linker automatically chooses the appropriate standard C/C++ libraries to use based on attributes from the object files. Imported object files might not have all these attributes. Therefore, you might need to help ILINK choose the standard library by verifying one or more of the following details:

- The used cpu by specifying the `--cpu` linker option
- If full I/O is needed; make sure to link with a Full library configuration in the standard library
- Explicitly specify runtime library file(s), possibly in combination with the `--no_library_search` linker option.

LINKING AEABI COMPLIANT MODULES USING A LINKER FROM A DIFFERENT VENDOR

If you have a module produced using the IAR C/C++ Compiler and you plan to link that module using a linker from a different vendor, that module must be AEABI compliant, see *Enabling AEABI compliance in the compiler*, page 124.

In addition, if that module uses any of the IAR-specific compiler extensions, you must make sure that those features are also supported by the tools from the other vendor. Note specifically:

- Support for the following extensions must be verified: `#pragma pack`, `__no_init`, `__root`, and `__ramfunc`
- The following extensions are harmless to use: `#pragma location`/`@`, `__arm`, `__thumb`, `__swi`, `__irq`, `__fiq`, and `__nested`.

ENABLING AEABI COMPLIANCE IN THE COMPILER

You can enable AEABI compliance in the compiler by setting the `--aeabi` option.

In the IDE, use the **Project>Options>C/C++ Compiler>Extra Options** page to specify the `--aeabi` option.

On the command line, use the option `--aeabi` to enable AEABI support in the compiler.

Alternatively, to enable support for AEABI for a specific system header file, you must define the preprocessor symbol `_AEABI_PORTABILITY_LEVEL` to non-zero prior to including a system header file, and make sure that the symbol `AEABI_PORTABLE` is set to non-zero after the inclusion of the header file:

```
#define _AEABI_PORTABILITY_LEVEL 1
#undef _AEABI_PORTABLE
#include <header.h>
#ifndef _AEABI_PORTABLE
  #error "header.h not AEABI compatible"

#endif
```

Efficient coding for embedded applications

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

The topics discussed are:

- Selecting data types
- Controlling data and function placement in memory
- Controlling compiler optimizations
- Facilitating good code generation.

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

Selecting data types

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

USING EFFICIENT DATA TYPES

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

- 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.
- Be aware of the costs of using 64-bit data types, such as `double` and `long long`.

- Bitfields and packed structures generate large and slow code.
- 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. Thus, you should consider replacing code that uses floating-point operations with code that uses integers, because these are more efficient.

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

In the compiler, the floating-point type `float` always uses the 32-bit format, and the type `double` always uses the 64-bit format. The format used by the `double` floating-point type depends on the setting of the `--double` compiler option.

Unless the application requires the extra precision that 64-bit floating-point numbers give, we recommend using 32-bit floating-point numbers instead.

By default, 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`, the `double` constant `1.0` 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 the suffix `f` to it, for example:

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

For more information about floating-point types, see *Floating-point types*, page 213.

ALIGNMENT OF ELEMENTS IN A STRUCTURE

The ARM core requires that data in memory must be aligned. Each element in a structure must be aligned according to its specified type requirements. This means that the compiler might need to insert *pad bytes* to keep the alignment correct.

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

- Due to external demands; for example, network communication protocols are usually specified in terms of data types with no padding in between
- You need to save data memory.

For information about alignment requirements, see *Alignment*, page 209.

There are two ways to solve the problem:

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

For further details about the `#pragma pack` directive, see *pack*, page 253.

ANONYMOUS STRUCTS AND UNIONS

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

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

In the IDE, language extensions are enabled by default.

Use the `-e` compiler option to enable language extensions. See *-e*, page 169, for additional information.

Example

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

```
struct S
{
  char mTag;
  union
  {
    long mL;
    float mF;
  };
} St;

void F(void)
{
  St.mL = 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 this example:

```
__no_init volatile
union
{
  unsigned char IOPORT;
  struct
  {
    unsigned char Way: 1;
    unsigned char Out: 1;
  };
} @ 0x1000;


/* Here the variables are used*/

void Test(void)
{
  IOPORT = 0;
  Way = 1;
  Out = 1;
}
```

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

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

Controlling data and function placement in memory

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

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

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

  Use the `@` operator or the `#pragma location` directive to place groups of functions or global and static variables in named sections, without having explicit control of each object. The sections can, for example, be placed in specific areas of memory, or initialized or copied in controlled ways using the section begin and end operators. This is also useful if you want an interface between separately linked units, for example an application project and a boot loader project. Use named sections when absolute control over the placement of individual variables is not needed, or not useful.
- The `--section` option

  Use the `--section` option to place functions and/or data objects in named sections, which is useful, for example, if you want to direct them to different fast or slow memories. To read more about the `--section` option, see *--section*, page 186.

At compile time, data and functions are placed in different sections as described in *Modules and sections*, page 40. At link time, one of the most important functions of the linker is to assign load addresses to the various sections used by the application. All sections, except for the sections holding absolute located data, are automatically allocated to memory according to the specifications in the linker configuration file, as described in *Placing code and data—the linker configuration file*, page 42.

DATA PLACEMENT AT AN ABSOLUTE LOCATION

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

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

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

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

Examples

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

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

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

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

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

These examples show incorrect usage:

```
int delta @ 0x100C;                /* Error, not __no_init */

__no_init int epsilon @ 0x1011;    /* Error, misaligned. */
```

C++ considerations

In C++, module scoped `const` variables are static (module local), whereas in C they are global. This means that each module that declares a certain `const` variable will contain a separate variable with this name. If you link an application with several such modules all containing (via a header file), for instance, the declaration:

```
volatile const __no_init int x @ 0x100;        /* Bad in C++ */
```

the linker will report that more than one variable is located at address `0x100`.

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

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

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

DATA AND FUNCTION PLACEMENT IN SECTIONS

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

- The `@` operator, alternatively the `#pragma location` directive, can be used for placing individual variables or individual functions in named sections. The named section can either be a predefined section, or a user-defined section.
- The `--section` option can be used for placing variables and functions, which are parts of the whole compilation unit, in named sections.

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

If you use your own sections, in addition to the predefined sections, the sections must also be defined in the linker configuration file.

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

The location of the sections can be controlled from the linker configuration file.

For more information about sections, see the chapter *Section reference*.

Examples of placing variables in named sections

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

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

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

Examples of placing functions in named sections

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

void g(void) @ "FUNCTIONS"
{
}

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

Controlling compiler optimizations

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

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

SCOPE FOR PERFORMED OPTIMIZATIONS

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

You can also exclude individual functions from the performed optimizations. The `#pragma optimize` directive allows you to either lower the optimization level, or specify another type of optimization to be performed. Refer to *optimize*, page 252, for information about the pragma directive.

Multi-file compilation units

In addition to applying different optimizations to different source files or even functions, you can also decide what a compilation unit consists of—one or several source code files.

By default, a compilation unit consists of one source file, but you can also use multi-file compilation to make several source files in a compilation unit. The advantage is that interprocedural optimizations such as inlining and cross jump have more source code to work on. Ideally, the whole application should be compiled as one compilation unit. However, for large applications this is not practical because of resource restrictions on the host computer. For more information, see --*mfc*, page 175.

If the whole application is compiled as one compilation unit, it is very useful to make the compiler also discard unused public functions and variables before the interprocedural optimizations are performed. Doing this limits the scope of the optimizations to functions and variables that are actually used. For more information, see --*discard_unused_publics*, page 168.

OPTIMIZATION LEVELS

The compiler supports different levels of optimizations. This table lists optimizations that are typically performed on each level:

Optimization level	Description
None (Best debug support)	Variables live through their entire scope
Low	Same as above but variables only live for as long as they are needed, not necessarily through their entire scope
Medium	Same as above, and: 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

Table 19: Compiler optimization levels

Optimization level	Description
High (Maximum optimization)	Same as above, and: Instruction scheduling Cross jumping Advanced register content analysis and optimization Loop unrolling Function inlining Code motion Type-based alias analysis

Table 19: Compiler optimization levels (Continued)

Note: Some of the performed optimizations can be individually enabled or disabled. For more information about these, see *Fine-tuning enabled transformations*, page 136.

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

SPEED VERSUS SIZE

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

FINE-TUNING ENABLED TRANSFORMATIONS

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

- Common subexpression 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 levels **None** and **Low**.

To read more about the command line option, see --*no_cse*, page 177.

Loop unrolling

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

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

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

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

To read more about the command line option, see --*no_unroll*, page 180.

Function inlining

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

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

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

To read more about the command line option, see --*no_inline*, page 178.

Code motion

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

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

Type-based alias analysis

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

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

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

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

To read more about the command line option, see *--no_tbaa*, page 179.

Example

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

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

Static clustering

When static clustering is enabled, static and global variables that are defined within the same module 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.

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

Instruction scheduling

The 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 that not all cores benefit from scheduling.

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

Facilitating good code generation

This section contains hints on how to help the compiler generate good code, for example:

- Using efficient addressing modes
- Helping the compiler optimize
- Generating more useful error message.

WRITING OPTIMIZATION-FRIENDLY SOURCE CODE

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

- Local variables—auto variables and parameters—are preferred over static or global variables. The reason is that the optimizer must assume, for example, that called functions can modify non-local variables. When the life spans for local variables end, the previously occupied memory can then be reused. Globally declared variables will occupy data memory during the whole program execution.
- Avoid taking the address of local variables using the `&` operator. This is inefficient for two main reasons. 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 might enable further optimizations. The compiler often inlines small functions declared static. The use of the `#pragma inline` directive and the C++

keyword `inline` gives you fine-grained control, and it is the preferred method compared to the traditional way of using preprocessor macros. Too much inlining can decrease performance due to the limited number of registers. This feature can be disabled using the `--no_inline` command line option; see *--no_inline*, page 178.

- 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 91.

SAVING STACK SPACE AND RAM MEMORY

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

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

FUNCTION PROTOTYPES

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

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

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

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

Prototyped style

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

```
int Test(char, int); /* Declaration */

int Test(char ch, int i) /* Definition */
{
  return i + ch;
}
```

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.

For example:

```
int Test();     /* Declaration */

int Test(ch, i) /* Definition */
char ch;
int i;
{
  return i + ch;
}
```

INTEGER TYPES AND BIT NEGATION

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

In some cases there might be warnings (for example, for constant conditional or pointless comparison), in others just a different result than what is expected. Under certain circumstances the compiler might warn only at higher optimizations, for example, if the compiler relies on optimizations to identify some instances of constant conditionals. In this 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 `0xFFFFFF7F`. On the left hand side, `c1` is an 8-bit unsigned character, so it cannot be larger than 255. It also cannot be negative, which means that the integral promoted value can never have the topmost 24 bits set.

PROTECTING SIMULTANEOUSLY ACCESSED VARIABLES

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

To mark a variable properly, use the `volatile` keyword. This informs the compiler, among other things, that the variable can be changed from other threads. The compiler will then avoid optimizing on the variable (for example, keeping track of the variable in registers), will not delay writes to it, and be careful accessing the variable only the number of times given in the source code. To read more about the `volatile` type qualifier, see *Declaring objects volatile*, page 218.

ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for several ARM devices are included in the IAR product installation. The header files are named `io`*`device`*`.h` and define the processor-specific special function registers (SFRs).

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

SFRs with bitfields are declared in the header file. This 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         :6;
} __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 devices. For details about the @ operator, see *Controlling data and function placement in memory*, page 131.

PASSING VALUES BETWEEN C AND ASSEMBLER OBJECTS

The following example shows how you in your C source code can use inline assembler to set and get values from a special purpose register:

```
#pragma diag_suppress=Pe940
#pragma optimize=no_inline
static unsigned long get_APSR( void )
{
  /* On function exit,
     function return value should be present in R0 */
  asm( "MRS R0, APSR" );
}
#pragma diag_default=Pe940

#pragma optimize=no_inline
static void set_APSR( unsigned long value)
{
  /* On function entry, the first parameter is found in R0 */
  asm( "MSR APSR, R0" );
}
```

The general purpose register `R0` is used for getting and setting the value of the special purpose register `APSR`. As the functions only contain inline assembler, the compiler will not interfere with the register usage. The register `R0` is always used for return values. The first parameter is always passed in `R0` if the type is 32 bits or smaller.

The same method can be used also for accessing other special purpose registers and specific instructions.

To read more about the risks of using inline assembler, see *Inline assembler*, page 93. For reference information about using inline assembler, see *Inline assembler*, page 225.

Note: Before you use inline assembler, see if you can use an intrinsic function instead. See *Summary of intrinsic functions*, page 259.

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 a separate section

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 239. Note that to use this keyword, language extensions must be enabled; see *-e*, page 169. For information about the `#pragma object_attribute`, see page 252.

Part 2. Reference information

This part of the IAR C/C++ Development Guide for ARM® contains these chapters:

- External interface details
- Compiler options
- Linker options
- Data representation
- Compiler extensions
- Extended keywords
- Pragma directives
- Intrinsic functions
- The preprocessor
- Library functions
- The linker configuration file
- Section reference
- IAR utilities
- Implementation-defined behavior.

External interface details

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

Invocation syntax

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

COMPILER INVOCATION SYNTAX

The invocation syntax for the compiler is:

```
iccarm [options] [sourcefile] [options]
```

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

```
iccarm prog.c --debug
```

The source file can be a C or C++ file, typically with the filename extension `c` or `cpp`, respectively. If no filename extension is specified, the file to be compiled must have the extension `c`.

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

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

ILINK INVOCATION SYNTAX

The invocation syntax for ILINK is:

```
ilinkarm [arguments]
```

Each argument is either a command-line option, an object file, or a library.

For example, when linking the object file `prog.o`, use this command:

```
ilinkarm prog.o --config configfile
```

If no filename extension is specified for the linker configuration file, the configuration file must have the extension `icf`.

Generally, the order of arguments on the command line is *not* significant. There is, however, one exception: when you supply several libraries, the libraries are searched in the same order that they are specified on the command line. The default libraries are always searched last.

The output executable image will be placed in a file named `a.out`, unless the `-o` option is used.

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

PASSING OPTIONS

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

- Directly from the command line

  Specify the options on the command line after the `iccarm` or `ilinkarm` commands; see *Invocation syntax*, page 147.
- Via environment variables

  The compiler and linker automatically append the value of the environment variables to every command line; see *Environment variables*, page 148.
- Via a text file, using the `-f` option; see *-f*, page 171.

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

ENVIRONMENT VARIABLES

These environment variables can be used with the compiler:

Environment variable	Description
C_INCLUDE	Specifies directories to search for include files; for example: `C_INCLUDE=c:\program files\iar systems\embedded workbench 5.`*n*`\arm\inc;c:\headers`
QCCARM	Specifies command line options; for example: `QCCARM=-lA asm.lst`

Table 20: Compiler environment variables

This environment variable can be used with ILINK:

Environment variable	Description
ILINKARM_CMD_LINE	Specifies command line options; for example: `ILINKARM_CMD_LINE=--config full.icf --silent`

Table 21: ILINK environment variables

Include file search procedure

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

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

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

  it searches these directories for the file to include:

  1 The directories specified with the `-I` option, in the order that they were specified, see *-I*, page 173.

  2 The directories specified using the `C_INCLUDE` environment variable, if any, see *Environment variables*, page 148.

- 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 this command for compilation:

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

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

`dir\include`	Current file is `src.h`.

`dir\src`	File including current file (`src.c`).
`dir\include`	As specified with the first `-I` option.
`dir\debugconfig`	As specified with the second `-I` option.

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

Compiler output

The compiler can produce the following output:

- A linkable object file

  The object files produced by the compiler use the industry-standard format ELF. By default, the object file has the filename extension `o`.
- Optional list files

  Various kinds of list files can be specified using the compiler option `-l`, see *-l*, page 173. By default, these files will have the filename extension `lst`.
- Optional preprocessor output files

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

  Diagnostic messages are directed to the standard error stream and displayed on the screen, and printed in an optional list file. To read more about diagnostic messages, see *Diagnostics*, page 152.
- Error return codes

  These codes provide status information to the operating system which can be tested in a batch file, see *Error return codes*, page 151.
- Size information

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

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

Error return codes

The compiler and linker return status information to the operating system that can be tested in a batch file.

These command line error codes are supported:

Code	Description
0	Compilation or linking successful, but there might have been warnings.
1	Warnings were produced and the option `--warnings_affect_exit_code` was used.
2	Errors occurred.
3	Fatal errors occurred, making the tool abort.
4	Internal errors occurred, making the tool abort.

Table 22: Error return codes

ILINK output

ILINK can produce the following output:

- An absolute executable image

  The final output produced by the IAR ILINK Linker is an absolute object file containing the executable image that can be put into an EPROM, downloaded to a hardware emulator, or executed on your PC using the IAR C-SPY Debugger Simulator. By default, the file has the filename extension `out`. The output format is always in ELF, which optionally includes debug information in the DWARF format.
- Optional logging information

  During operation, ILINK logs its decisions on `stdout`, and optionally to a file. For example, if a library is searched, whether a required symbol is found in a library module, or whether a module will be part of the output. Timing information for each ILINK subsystem is also logged.
- Optional map files

  A linker map file—containing summaries of linkage, runtime attributes, memory, and placement, as well as an entry list— can be generated by the ILINK option `--map`, see *--map*, page 200. By default, the map file has the filename extension `map`.
- Diagnostic messages

  Diagnostic messages are directed to `stderr` and displayed on the screen, as well as printed in the optional map file. To read more about diagnostic messages, see *Diagnostics*, page 152.

- Error return codes

  ILINK returns status information to the operating system which can be tested in a batch file, see *Error return codes*, page 151.

- Size information about used memory and amount of time

  Information about the generated amount of bytes for functions and data for each memory is directed to `stdout` and displayed on the screen.

Diagnostics

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

MESSAGE FORMAT FOR THE COMPILER

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

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

with these elements:

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

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

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

MESSAGE FORMAT FOR THE LINKER

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

level[*tag*]: *message*

with these elements:

`level`	The level of seriousness of the issue
`tag`	A unique tag that identifies the diagnostic message
`message`	An explanation, possibly several lines long

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

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

SEVERITY LEVELS

The diagnostic messages are divided into different levels of severity:

Remark

A diagnostic message that is produced when the compiler or linker finds a 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 185.

Warning

A diagnostic message that is produced when the compiler or linker finds a problem which is of concern, but not so severe as to prevent the completion of compilation or linking. Warnings can be disabled by use of the command line option `--no_warnings`, see page 181.

Error

A diagnostic message that is produced when the compiler or linker finds a serious error. An error will produce a non-zero exit code.

Fatal error

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

SETTING THE SEVERITY LEVEL

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

See *Summary of compiler options*, page 158, for a description of the compiler options that are available for setting severity levels.

For the compiler see also 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 was a serious and unexpected failure due to a fault in the compiler or linker. It is produced using this form:

```
Internal error: message
```

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

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

Compiler options

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

Options syntax

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

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

TYPES OF OPTIONS

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

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

For information about the different methods for passing options, see *Passing options*, page 148.

RULES FOR SPECIFYING PARAMETERS

There are some general syntax rules for specifying option parameters. First, the rules depending on whether the parameter is *optional* or *mandatory*, and whether the option has a short or a long name, are described. Then, the rules for specifying filenames and directories are listed. Finally, the remaining rules are listed.

Rules for optional parameters

For options with a short name and an optional parameter, any parameter should be specified without a preceding space, for example:

`-O` or `-Oh`

For options with a long name and an optional parameter, any parameter should be specified with a preceding equal sign (=), for example:

`--misrac2004=n`

Rules for mandatory parameters

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

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

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

```
--diagnostics_tables=MyDiagnostics.lst
```

or

```
--diagnostics_tables MyDiagnostics.lst
```

Rules for options with both optional and mandatory parameters

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

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

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

```
-lA MyList.lst
```

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

```
--preprocess=n PreprocOutput.lst
```

Rules for specifying a filename or directory as parameters

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

- Options that take a filename as a parameter can optionally also take a path. The path can be relative or absolute. For example, to generate a listing to the file `List.lst` in the directory `..\listings\`:

  ```
  iccarm prog.c -l ..\listings\List.lst
  ```

- For options that take a filename as the destination for output, the parameter can be specified as a path without a specified filename. The compiler stores the output in that directory, in a file with an extension according to the option. The filename will be the same as the name of the compiled source file, unless a different name was specified with the option `-o`, in which case that name is used. For example:

  ```
  iccarm prog.c -l ..\listings\
  ```

  The produced list file will have the default name `..\listings\prog.lst`
- The *current directory* is specified with a period (`.`). For example:

  ```
  iccarm prog.c -l .
  ```

- `/` can be used instead of `\` as the directory delimiter.
- By specifying `-`, input files and output files can be redirected to the standard input and output stream, respectively. For example:

  ```
  iccarm prog.c -l -
  ```

Additional rules

These rules also apply:

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

  ```
  iccarm prog.c -l ---r
  ```

- For options that accept multiple arguments of the same type, the arguments can be provided as a comma-separated list (without a space), for example:

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

  Alternatively, the option can be repeated for each argument, for example:

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

Summary of compiler options

This table summarizes the compiler command line options:

Command line option	Description
--aapcs	Specifies the calling convention
--aeabi	Enables AEABI-compliant code generation
--align_sp_on_irq	Generates code to align SP on entry to __irq functions
--arm	Sets the default function mode to ARM
--char_is_signed	Treats char as signed
--cpu	Specifies a processor variant
--cpu_mode	Selects the default mode for functions
-D	Defines preprocessor symbols
--debug	Generates debug information
--dependencies	Lists file dependencies
--diag_error	Treats these as errors
--diag_remark	Treats these as remarks
--diag_suppress	Suppresses these diagnostics
--diag_warning	Treats these as warnings
--diagnostics_tables	Lists all diagnostic messages
--discard_unused_publics	Discards unused public symbols
--dlib_config	Determines the library configuration file
-e	Enables language extensions
--ec++	Enables Embedded C++ syntax
--eec++	Enables Extended Embedded C++ syntax
--enable_hardware_workaround	Enables a specific hardware workaround
--enable_multibytes	Enables support for multibyte characters in source files
--endian	Specifies the byte order of the generated code and data
--enum_is_int	Sets the minimum size on enumeration types
--error_limit	Specifies the allowed number of errors before compilation stops
-f	Extends the command line

Table 23: Compiler options summary

Command line option	Description
--fpu	Selects the type of floating-point unit
--header_context	Lists all referred source files and header files
-I	Specifies include file path
--interwork	Generates interworking code
-l	Creates a list file
--legacy	Generates object code linkable with older tool chains
--mfc	Enables multi file compilation
--migration_preprocessor_extensions	Extends the preprocessor
--misrac1998	Enables error messages specific to MISRA-C:1998. See the *IAR Embedded Workbench® MISRA C:1998 Reference Guide*.
--misrac2004	Enables error messages specific to MISRA-C:2004. See the *IAR Embedded Workbench® MISRA C:2004 Reference Guide*.
--misrac_verbose	Enables verbose logging of MISRA C checking. See the *IAR Embedded Workbench® MISRA C:1998 Reference Guide* or the *IAR Embedded Workbench® MISRA C:2004 Reference Guide*.
--no_clustering	Disables static clustering optimizations
--no_code_motion	Disables code motion optimization
--no_const_align	Disables the alignment optimization for constants.
--no_cse	Disables common subexpression elimination
--no_fragments	Disables section fragment handling
--no_guard_calls	Disables guard calls for static initializers
--no_inline	Disables function inlining
--no_path_in_file_macros	Removes the path from the return value of the symbols `__FILE__` and `__BASE_FILE__`
--no_scheduling	Disables the instruction scheduler
--no_tbaa	Disables type-based alias analysis
--no_typedefs_in_diagnostics	Disables the use of typedef names in diagnostics
--no_unaligned_access	Avoids unaligned accesses
--no_unroll	Disables loop unrolling

Table 23: Compiler options summary (Continued)

Command line option	Description
--no_warnings	Disables all warnings
--no_wrap_diagnostics	Disables wrapping of diagnostic messages
-O	Sets the optimization level
-o	Sets the object filename
--only_stdout	Uses standard output only
--output	Sets the object filename
--predef_macros	Lists the predefined symbols.
--preinclude	Includes an include file before reading the source file
--preprocess	Generates preprocessor output
--public_equ	Defines a global named assembler label
-r	Generates debug information
--remarks	Enables remarks
--require_prototypes	Verifies that functions are declared before they are defined
--section	Changes a section name
--separate_cluster_for_ initialized_variables	Separates initialized and non-initialized variables
--silent	Sets silent operation
--strict_ansi	Checks for strict compliance with ISO/ANSI C
--thumb	Sets default function mode to Thumb
--use_unix_directory_ separators	Uses / as directory separator in paths
--warnings_affect_exit_code	Warnings affects exit code
--warnings_are_errors	Warnings are treated as errors

Table 23: Compiler options summary (Continued)

Descriptions of options

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

Note that if you use the options page **Extra Options** to specify specific command line options, the IDE does not perform an instant check for consistency problems like conflicting options, duplication of options, or use of irrelevant options.

--aapcs

Syntax `--aapcs={std|vfp}`

Parameters

`std`	Processor registers are used for floating-point parameters and return values in function calls according to standard AAPCS. `std` is the default when the `--aeabi` compiler option is used or the software FPU is selected. Note that this calling convention enables guard calls.
`vfp`	VFP registers are used for floating-point parameters and return values. The generated code is not compatible with AEABI code. `vfp` is the default when a VFP is selected and `--aeabi` is not used.

Description Use this option to specify the calling convention.

Project>Options>C/C++ Compiler>Extra Options.

--aeabi

Syntax `--aeabi`

Description Use this option to generate AEABI compliant object code.

See also *AEABI compliance*, page 123 and *--no_guard_calls*, page 177.

Project>Options>C/C++ Compiler>Extra Options.

--align_sp_on_irq

Syntax `--align_sp_on_irq`

Description Use this option to align the stack pointer (`SP`) on entry to `__irq` declared functions.

See also *__irq*, page 238.

Project>Options>C/C++ Compiler>Extra Options.

--arm

Syntax `--arm`

Description Use this option to set default function mode to ARM. This setting must be the same for all files included in a program, unless they are interworking.

Note: This option has the same effect as the `--cpu_mode=arm` option.

See also *--interwork*, page 173 and *__interwork*, page 238.

Project>Options>General Options>Target>Processor mode>Arm

--char_is_signed

Syntax `--char_is_signed`

Description By default, the compiler interprets the `char` type as unsigned. Use this option to make the compiler interpret the `char` type as signed instead. This can be useful when you, for example, want to maintain compatibility with another compiler.

Note: The runtime library is compiled without the `--char_is_signed` option. If you use this option, you might get type mismatch warnings from the linker, because the library uses `unsigned char`.

Project>Options>C/C++ Compiler>Language>Plain 'char' is

--cpu

Syntax `--cpu=`*`core`*

Parameters

`core`	Specifies a specific processor variant

Description Use this option to select the processor variant for which the code is to be generated. The default is ARM7TDMI. The following cores and processor macrocells are recognized:

- `ARM7TDMI`
- `ARM7TDMI-S`
- `ARM710T`
- `ARM720T`
- `ARM740T`
- `ARM7EJ-S`
- `ARM9TDMI`

- ARM920T
- ARM922T
- ARM940T
- ARM9E
- ARM9E-S
- ARM926EJ-S
- ARM946E-S
- ARM966E-S
- ARM968E-S
- ARM10E
- ARM1020E
- ARM1022E
- ARM1026EJ-S
- ARM1136J
- ARM1136J-S
- ARM1136JF
- ARM1136JF-S
- ARM1176J
- ARM1176J-S
- ARM1176JF
- ARM1176JF-S
- Cortex-M0
- Cortex-M1
- Cortex-Ms1[*]
- Cortex-M3
- Cortex-R4
- XScale
- XScale-IR7.

[*] Cortex-M1 with Operating System extension.

See also

Processor variant, page 20.

Project>Options>General Options>Target>Processor configuration

--cpu_mode

Syntax

`--cpu_mode={arm|a|thumb|t}`

Parameters

`arm, a` (default)	Selects the arm mode as the default mode for functions
`thumb, t`	Selects the thumb mode as the default mode for functions

Description

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 interworking.

See also — *--interwork*, page 173 and *__interwork*, page 238.

Project>Options>General Options>Target>Processor mode

-D

Syntax

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

Parameters

symbol	The name of the preprocessor symbol
value	The value of the preprocessor symbol

Description

Use this option to define a preprocessor symbol. If no value is specified, `1` is used. This option can be used one or more times on the command line.

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

```
-Dsymbol
```

is equivalent to:

```
#define symbol 1
```

To get the equivalence of:

```
#define FOO
```

specify the `=` sign but nothing after, for example:

```
-DFOO=
```

Project>Options>C/C++ Compiler>Preprocessor>Defined symbols

--debug, -r

Syntax

```
--debug
-r
```

Description

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

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

Project>Options>C/C++ Compiler>Output>Generate debug information

--dependencies

Syntax

```
--dependencies[=[i|m]] {filename|directory}
```

Parameters

`i` (default)	Lists only the names of files
`m`	Lists in makefile style

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 156.

Description

Use this option to make the compiler list all source and header files opened by the compilation into a file with the default filename extension `i`.

Example

If `--dependencies` or `--dependencies=i` is used, the name of each opened source file, including the full path, if available, is output on a separate line. For example:

```
c:\iar\product\include\stdio.h
d:\myproject\include\foo.h
```

If `--dependencies=m` is used, the output uses makefile style. For each source file, one line containing a makefile dependency rule is produced. Each line consists of the name of the object file, a colon, a space, and the name of a source file. For example:

```
foo.o: c:\iar\product\include\stdio.h
foo.o: d:\myproject\include\foo.h
```

An example of using `--dependencies` with a popular make utility, such as gmake (GNU make):

1 Set up the rule for compiling files to be something like:

```
%.o : %.c
        $(ICC) $(ICCFLAGS) $< --dependencies=m $*.d
```

That is, in addition to producing an object file, the command also produces a dependency file in makefile style (in this example, using the extension `.d`).

2 Include all the dependency files in the makefile using, for example:

```
-include $(sources:.c=.d)
```

Because of the dash (`-`) it works the first time, when the `.d` files do not yet exist.

This option is not available in the IDE.

--diag_error

Syntax `--diag_error=`*`tag`*`[,`*`tag`*`,...]`

Parameters

tag	The number of a diagnostic message, for example the message number `Pe117`

Description Use this option to reclassify certain diagnostic messages as errors. An error indicates a violation of the C or C++ language rules, of such severity that object code will not be generated. The exit code will be non-zero. This option may be used more than once on the command line.

Project>Options>C/C++ Compiler>Diagnostics>Treat these as errors

--diag_remark

Syntax `--diag_remark=`*`tag`*`[,`*`tag`*`,...]`

Parameters

tag	The number of a diagnostic message, for example the message number `Pe177`

Description Use this option to reclassify certain diagnostic messages as remarks. A remark is the least severe type of diagnostic message and indicates a source code construction that may cause strange behavior in the generated code. This option may be used more than once on the command line.

Note: By default, remarks are not displayed; use the `--remarks` option to display them.

Project>Options>C/C++ Compiler>Diagnostics>Treat these as remarks

--diag_suppress

Syntax

`--diag_suppress=tag[,tag,...]`

Parameters

tag	The number of a diagnostic message, for example the message number `Pe117`

Description

Use this option to suppress certain diagnostic messages. These messages will not be displayed. This option may be used more than once on the command line.

Project>Options>C/C++ Compiler>Diagnostics>Suppress these diagnostics

--diag_warning

Syntax

`--diag_warning=tag[,tag,...]`

Parameters

tag	The number of a diagnostic message, for example the message number `Pe826`

Description

Use this option to reclassify certain diagnostic messages as warnings. A warning indicates an error or omission that is of concern, but which will not cause the compiler to stop before compilation is completed. This option may be used more than once on the command line.

Project>Options>C/C++ Compiler>Diagnostics>Treat these as warnings

--diagnostics_tables

Syntax

`--diagnostics_tables {filename|directory}`

Parameters

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 156.

Description

Use this option to list all possible diagnostic messages in a named file. This can be convenient, for example, if you have used a pragma directive to suppress or change the severity level of any diagnostic messages, but forgot to document why.

This option cannot be given together with other options.

This option is not available in the IDE.

--discard_unused_publics

Syntax

`--discard_unused_publics`

Description

Use this option to discard unused public functions and variables from the compilation unit. This enhances interprocedural optimizations such as inlining, cross call, and cross jump by limiting their scope to public functions and variables that are actually used.

This option is only useful when *all* source files are compiled as one unit, which means that the `--mfc` compiler option is used.

Note: Do not use this option only on parts of the application, as necessary symbols might be removed from the generated output.

See also

--*mfc*, page 175 and *Multi-file compilation units*, page 135.

Project>Options>C/C++ Compiler>Discard unused publics

--dlib_config

Syntax

`--dlib_config` *`filename`*

Parameters

For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 156.

Description

Each runtime library has a corresponding library configuration file. Use this option to specify the library configuration file for the compiler. Make sure that you specify a configuration file that corresponds to the library you are using.

All prebuilt runtime libraries are delivered with corresponding configuration files. You can find the library object files and the library configuration files in the directory `arm\lib`. For examples and a list of prebuilt runtime libraries, see *Using a prebuilt library*, page 64.

If you build your own customized runtime library, you should also create a corresponding customized library configuration file, which must be specified to the compiler. For more information, see *Building and using a customized library*, page 71.

To set related options, choose:

Project>Options>General Options>Library Configuration

-e

Syntax `-e`

Description In the command line version of the compiler, language extensions are disabled by default. If you use language extensions such as extended keywords and anonymous structs and unions in your source code, you must use this option to enable them.

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

See also The chapter *Compiler extensions.*

Project>Options>C/C++ Compiler>Language>Allow IAR extensions

Note: By default, this option is enabled in the IDE.

--ec++

Syntax `--ec++`

Description In the compiler, the default language is C. If you use Embedded C++, you must use this option to set the language the compiler uses to Embedded C++.

Project>Options>C/C++ Compiler>Language>Embedded C++

--eec++

Syntax `--eec++`

Description In the compiler, the default language is C. If you take advantage of Extended Embedded C++ features like namespaces or the standard template library in your source code, you must use this option to set the language the compiler uses to Extended Embedded C++.

See also

Extended Embedded C++, page 108.

Project>Options>C/C++ Compiler>Language>Extended Embedded C++

--enable_hardware_workaround

Syntax

`--enable_hardware_workaround=`*`waid`*`[,`*`waid`*`[...]]`

Parameters

waid	The ID number of the workaround to enable. For a list of available workarounds to enable, see the release notes.

Description

Use this option to make the compiler generate a workaround for a specific hardware problem.

See also

The release notes for a list of available parameters.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--enable_multibytes

Syntax

`--enable_multibytes`

Description

By default, multibyte characters cannot be used in C or C++ source code. Use this option to make multibyte characters in the source code be interpreted according to the host computer's default setting for multibyte support.

Multibyte characters are allowed in C and C++ style comments, in string literals, and in character constants. They are transferred untouched to the generated code.

Project>Options>C/C++ Compiler>Language>Enable multibyte support

--endian

Syntax

`--endian={big|b|little|l}`

Parameters

`big, b`	Specifies big endian as the default byte order
`little, l` (default)	Specifies little endian as the default byte order

Description

Use this option to specify 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 21, *Byte order*, page 210, *--BE8*, page 191, and *--BE32*, page 191.

Project>Options>General Options>Target>Endian mode

--enum_is_int

Syntax

`--enum_is_int`

Description

Use this option to force the size of all enumeration types to be at least 4 bytes.

Note: This option will not consider the fact that an `enum` type can be larger than an integer type.

See also

The enum type, page 211.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--error_limit

Syntax

`--error_limit=n`

Parameters

n	The number of errors before the compiler stops the compilation. *n* must be a positive integer; `0` indicates no limit.

Description

Use the `--error_limit` option to specify the number of errors allowed before the compiler stops the compilation. By default, 100 errors are allowed.

This option is not available in the IDE.

-f

Syntax

`-f filename`

Parameters

For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 156.

Descriptions

Use this option to make the compiler read command line options from the named file, with the default filename extension `xcl`.

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

Both C and C++ style comments are allowed in the file. Double quotes behave in the same way as in the Microsoft Windows command line environment.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--fpu

Syntax

`--fpu={VFPv1|VFPv2|VFP9-S|none}`

Parameters

`VFPv1`	For a vector floating-point unit conforming to the architecture VFPv1.
`VFPv2`	For a system that implements a VFP unit conforming to the architecture VFPv2.
`VFP9-S`	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.
`none` (default)	The software floating-point library is used.

Description

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.

See also

VFP and floating-point arithmetic, page 21.

Project>Options>General Options>Target>FPU

--header_context

Syntax

`--header_context`

Description

Occasionally, to find the cause of a problem it is necessary to know which header file that was included from which source line. Use this option to list, for each diagnostic message, not only the source position of the problem, but also the entire include stack at that point.

This option is not available in the IDE.

-I

Syntax

`-I path`

Parameters

path	The search path for `#include` files

Description

Use this option to specify the search paths for `#include` files. This option can be used more than once on the command line.

See also

Include file search procedure, page 149.

Project>Options>C/C++ Compiler>Preprocessor>Additional include directories

--interwork

Syntax

`--interwork`

Description

Use this option to generate interworking code.

In code compiled with this option, functions will by default be of the type interwork. It will be possible to mix files compiled as `arm` and `thumb` (using the `--cpu_mode` option) as long as they are all compiled with the `--interwork` option.

Note: Source code compiled for an ARM architecture v5 or higher, or AEABI compliance is interworking by default.

Project>Options>General Options>Target>Generate interwork code

-l

Syntax

`-l[a|A|b|B|c|C|D][N][H] {filename|directory}`

Parameters

`a` (default)	Assembler list file
`A`	Assembler list file with C or C++ source as comments

`b`	Basic assembler list file. This file has the same contents as a list file produced with `-la`, except that no extra compiler-generated information (runtime model attributes, call frame information, frame size information) is included *
`B`	Basic assembler list file. This file has the same contents as a list file produced with `-lA`, except that no extra compiler generated information (runtime model attributes, call frame information, frame size information) is included *
`c`	C or C++ list file
`C` (default)	C or C++ list file with assembler source as comments
`D`	C or C++ list file with assembler source as comments, but without instruction offsets and hexadecimal byte values
`N`	No diagnostics in file
`H`	Include source lines from header files in output. Without this option, only source lines from the primary source file are included

*** This makes the list file less useful as input to the assembler, but more useful for reading by a human.**

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 156.

Description

Use this option to generate an assembler or C/C++ listing to a file. Note that this option can be used one or more times on the command line.

To set related options, choose:

Project>Options>C/C++ Compiler>List

--legacy

Syntax

`--legacy={mode}`

Parameters

`RVCT3.0`	Generates object code linkable with the linker in RVCT3.0. Use this mode together with the `--aeabi` option to export code that should be linked with the linker in RVCT3.0.

Description

Use this option to generate code compatible with older tool chains.

Project>Options>C/C++ Compiler>Extra Options.

--mfc

Syntax

```
--mfc
```

Description

Use this option to enable *multi-file compilation*. This means that the compiler compiles one or several source files specified on the command line as one unit, which makes interprocedural optimizations such as inlining, cross call, and cross jump possible.

Note: The compiler will generate one object file per input source code file, where the first object file contains all relevant data and the other ones are empty. If you want only the first file to be produced, use the `-o` compiler option and specify a certain output file.

Example

```
iccarm myfile1.c myfile2.c myfile3.c --mfc
```

See also

--discard_unused_publics, page 168, *-o, --output*, page 182, and *Multi-file compilation units*, page 135.

Project>Options>C/C++ Compiler>Multi-file compilation

--migration_preprocessor_extensions

Syntax

```
--migration_preprocessor_extensions
```

Description

If you need to migrate code from an earlier IAR Systems C or C/C++ compiler, you might want to use this option. Use this option to use the following in preprocessor expressions:

- Floating-point expressions
- Basic type names and `sizeof`
- All symbol names (including typedefs and variables).

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

Important! Do not depend on these extensions in newly written code, because support for them might be removed in future compiler versions.

Project>Options>C/C++ Compiler>Language>Enable IAR migration preprocessor extensions

--no_clustering

Syntax `--no_clustering`

Description Use this option to disable static clustering optimizations. 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. These optimizations, which are performed at optimization levels Medium and High, normally reduce code size and execution time.

Note: This option has no effect at optimization levels below Medium.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Static clustering

--no_code_motion

Syntax `--no_code_motion`

Description Use this option to disable code motion optimizations. These optimizations, which are performed at the optimization levels Medium and High, normally reduce code size and execution time. However, the resulting code might be difficult to debug.

Note: This option has no effect at optimization levels below Medium.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Code motion

--no_const_align

Syntax `--no_const_align`

Description By default, the compiler uses alignment 4 for objects with a size of 4 bytes or more. Use this option to make the compiler align `const` objects based on the alignment of their type.

For example, a string literal will get alignment 1, because it is an array with elements of the type `const char` which has alignment 1. Using this option might save ROM space, possibly at the expense of processing speed.

See also — *Alignment*, page 209.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--no_cse

Syntax

```
--no_cse
```

Description

Use this option to disable common subexpression elimination. At the optimization levels Medium and High, the compiler avoids calculating the same expression more than once. This optimization normally reduces both code size and execution time. However, the resulting code might be difficult to debug.

Note: This option has no effect at optimization levels below Medium.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Common subexpression elimination

--no_fragments

Syntax

```
--no_fragments
```

Description

Use this option to disable section fragment handling. Normally, the toolset uses IAR proprietary information for transferring section fragment information to the linker. The linker uses this information to remove unused code and data, and thus further minimize the size of the executable image. The effect of using this option in the compiler is smaller object size.

See also — *Keeping symbols and sections*, page 52.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**

--no_guard_calls

Syntax

```
--no_guard_calls
```

Description

If the `--aeabi` compiler option is used, the compiler produces extra library calls that guard the initialization of static variables in file scope. These library calls are only meaningful in an OS environment where you must make sure that these variables are not initialized by another concurrent process at the same time.

Use this option to remove these library calls.

Note: To be AEABI compliant, this option must not be used.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--no_inline

Syntax

`--no_inline`

Description

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

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

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

Note: This option has no effect at optimization levels below High.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Function inlining

--no_path_in_file_macros

Syntax

`--no_path_in_file_macros`

Description

Use this option to exclude the path from the return value of the predefined preprocessor symbols `__FILE__` and `__BASE_FILE__`.

See also

Descriptions of predefined preprocessor symbols, page 284.

This option is not available in the IDE.

--no_scheduling

Syntax

`--no_scheduling`

Description

Use this option to disable the instruction scheduler. The 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. This optimization, which is performed at optimization level High, normally reduce execution time. However, the resulting code might be difficult to debug.

Note: This option has no effect at optimization levels below High.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Instruction scheduling

--no_tbaa

Syntax

```
--no_tbaa
```

Description

Use this option to disable type-based alias analysis. When this options is not used, the compiler is free to assume that objects are only accessed through the declared type or through `unsigned char`.

See also

Type-based alias analysis, page 138.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Type-based alias analysis

--no_typedefs_in_diagnostics

Syntax

```
--no_typedefs_in_diagnostics
```

Description

Use this option to disable the use of typedef names in diagnostics. Normally, when a type is mentioned in a message from the compiler, most commonly in a diagnostic message of some kind, the typedef names that were used in the original declaration are used whenever they make the resulting text shorter.

Example

```
typedef int (*MyPtr)(char const *);
MyPtr p = "foo";
```

will give an error message like this:

```
Error[Pe144]: a value of type "char *" cannot be used to
initialize an entity of type "MyPtr"
```

If the `--no_typedefs_in_diagnostics` option is used, the error message will be like this:

```
Error[Pe144]: a value of type "char *" cannot be used to
initialize an entity of type "int (*)(char const *)"
```

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--no_unaligned_access

Syntax

`--no_unaligned_access`

Description

Use this option to make the compiler avoid unaligned accesses. Data accesses are usually performed aligned for improved performance. However, some accesses, most notably when reading from or writing to packed data structures, may be unaligned. When using this option, all such accesses will be performed using a smaller data size to avoid any unaligned accesses. This option is only useful for ARMv6 architectures and higher.

See also

--*interwork*, page 173 and __*interwork*, page 238.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--no_unroll

Syntax

`--no_unroll`

Description

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

For small loops, the overhead required to perform the looping can be large compared with the work performed in the loop body.

The loop unrolling optimization duplicates the body several times, reducing the loop overhead. The unrolled body also opens up for other optimization opportunities.

This optimization, which is performed at optimization level High, normally reduces execution time, but increases code size. The resulting code 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 below High.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Loop unrolling

--no_warnings

Syntax

`--no_warnings`

Description

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

This option is not available in the IDE.

--no_wrap_diagnostics

Syntax

`--no_wrap_diagnostics`

Description

By default, long lines in diagnostic messages are broken into several lines to make the message easier to read. Use this option to disable line wrapping of diagnostic messages.

This option is not available in the IDE.

-O

Syntax

`-O[n|l|m|h|hs|hz]`

Parameters

`n`	None* (Best debug support)
`l` (default)	Low*
`m`	Medium
`h`	High, balanced
`hs`	High, favoring speed
`hz`	High, favoring size

***The most important difference between None and Low is that at None, all non-static variables will live during their entire scope.**

Description

Use this option to set the optimization level to be used by the compiler when optimizing the code. If no optimization option is specified, the optimization level Low is used by default. If only `-O` is used without any parameter, the optimization level High balanced is used.

A low level of optimization makes it relatively easy to follow the program flow in the debugger, and, conversely, a high level of optimization makes it relatively hard.

See also

Controlling compiler optimizations, page 134.

Project>Options>C/C++ Compiler>Optimizations

-o, --output

Syntax

```
-o {filename|directory}
--output {filename|directory}
```

Parameters

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 156.

Description

By default, the object code output produced by the compiler is located in a file with the same name as the source file, but with the extension `o`. Use this option to explicitly specify a different output filename for the object code output.

This option is not available in the IDE.

--only_stdout

Syntax

```
--only_stdout
```

Description

Use this option to make the compiler use the standard output stream (`stdout`) also for messages that are normally directed to the error output stream (`stderr`).

This option is not available in the IDE.

--output, -o

Syntax

```
--output {filename|directory}
-o {filename|directory}
```

Parameters

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 156.

Description

By default, the object code output produced by the compiler is located in a file with the same name as the source file, but with the extension `o`. Use this option to explicitly specify a different output filename for the object code output.

This option is not available in the IDE.

--predef_macros

Syntax

```
--predef_macros {filename|directory}
```

Parameters

For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 156.

Description

Use this option to list the predefined symbols. When using this option, make sure to also use the same options as for the rest of your project.

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 `predef` filename extension.

Note that this option requires that you specify a source file on the command line.

This option is not available in the IDE.

--preinclude

Syntax

```
--preinclude includefile
```

Parameters

For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 156.

Description

Use this option to make the compiler include the specified include file before it starts to read the source file. This is useful if you want to change something in the source code for the entire application, for instance if you want to define a new symbol.

Project>Options>C/C++ Compiler>Preprocessor>Preinclude file

--preprocess

Syntax

`--preprocess[=[c][n][l]] {filename|directory}`

Parameters

`c`	Preserve comments
`n`	Preprocess only
`l`	Generate `#line` directives

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 156.

Description

Use this option to generate preprocessed output to a named file.

Project>Options>C/C++ Compiler>Preprocessor>Preprocessor output to file

--public_equ

Syntax

`--public_equ symbol[=value]`

Parameters

symbol	The name of the assembler symbol to be defined
value	An optional value of the defined assembler symbol

Description

This option is equivalent to defining a label in assembler language using the `EQU` directive and exporting it using the `PUBLIC` directive. This option can be used more than once on the command line.

This option is not available in the IDE.

-r, --debug

Syntax

```
-r
--debug
```

Description

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

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

Project>Options>C/C++ Compiler>Output>Generate debug information

--remarks

Syntax

```
--remarks
```

Description

The least severe diagnostic messages are called remarks. A remark indicates a source code construct that may cause strange behavior in the generated code. By default, the compiler does not generate remarks. Use this option to make the compiler generate remarks.

See also

Severity levels, page 153.

Project>Options>C/C++ Compiler>Diagnostics>Enable remarks

--require_prototypes

Syntax

```
--require_prototypes
```

Description

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

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

Note: This option only applies to functions in the C standard library.

Project>Options>C/C++ Compiler>Language>Require prototypes

--section

Syntax

```
--section OldName=NewName
```

Description

The compiler places functions and data objects into named sections which are referred to by the IAR ILINK Linker. Use this option to change the name of the section `OldName` to `NewName`.

This is useful if you want to place your code or data in different address ranges and you find the `@` notation, alternatively the `#pragma location` directive, insufficient. Note that any changes to the section names require corresponding modifications in the linker configuration file.

Example

To place functions in the section `MyText`, use:

```
--section .text=MyText
```

See also

For information about the different methods for controlling placement of data and code, see *Controlling data and function placement in memory*, page 131.

Project>Options>C/C++ Compiler>Output>Code section name

--separate_cluster_for_initialized_variables

Syntax

```
--separate_cluster_for_initialized_variables
```

Description

Use this option to separate initialized and non-initialized variables when using variable clustering. This might reduce the number of bytes in the ROM area which are needed for data initialization, but it might lead to larger code.

This option can be useful if you want to have your own data initialization routine, but want the IAR tools to arrange for the zero-initialized variables.

See also

Manual initialization, page 54 and *Initialize directive*, page 305.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--silent

Syntax `--silent`

Description

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

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

This option is not available in the IDE.

--strict_ansi

Syntax `--strict_ansi`

Description

By default, the compiler accepts a relaxed superset of ISO/ANSI C/C++, see *Minor language extensions*, page 228. Use this option to ensure that the program conforms to the ISO/ANSI C/C++ standard.

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

Project>Options>C/C++ Compiler>Language>Language conformances>Strict ISO/ANSI

--thumb

Syntax `--thumb`

Description

Use this option to set default function mode to Thumb. This setting must be the same for all files included in a program, unless they are interworking.

Note: This option has the same effect as the `--cpu_mode=thumb` option.

See also

--*interwork*, page 173 and __*interwork*, page 238.

Project>Options>General Options>Target>Processor mode>Arm

--use_unix_directory_separators

Syntax `--use_unix_directory_separators`

Description Use this option to make DWARF debug information use `/` (instead of `\`) as directory separators in paths.

This option can be useful if you have a debugger that requires directory separators in UNIX style.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--warnings_affect_exit_code

Syntax `--warnings_affect_exit_code`

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

This option is not available in the IDE.

--warnings_are_errors

Syntax `--warnings_are_errors`

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

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

See also --*diag_warning*, page 225.

Project>Options>C/C++ Compiler>Diagnostics>Treat all warnings as errors

Linker options

This chapter gives detailed reference information about each linker option.

For general syntax rules, see *Options syntax*, page 155.

Summary of linker options

This table summarizes the linker options:

Command line option	Description
`--BE8`	Uses the big-endian format BE8
`--BE32`	Uses the big-endian format BE32
`--config`	Specifies the linker configuration file to be used by the linker
`--config_def`	Defines symbols for the configuration file
`--cpp_init_routine`	Specifies a user-defined C++ dynamic initialization routine
`--cpu`	Specifies a processor variant
`--define_symbol`	Defines symbols that can be used by the application
`--diag_error`	Treats these message tags as errors
`--diag_remark`	Treats these message tags as remarks
`--diag_suppress`	Suppresses these diagnostic messages
`--diag_warning`	Treats these message tags as warnings
`--diagnostics_tables`	Lists all diagnostic messages
`--entry`	Treats the symbol as a root symbol and as the start of the application
`--error_limit`	Specifies the allowed number of errors before compilation stops
`--export_builtin_config`	Produces an `icf` file for the default configuration
`-f`	Extends the command line
`--force_output`	Produces an output file even if errors occurred
`--image_input`	Puts an image file in a section
`--keep`	Forces a symbol to be included in the application
`--log`	Enables log output for selected topics

Table 24: Linker options summary

Command line option	Description
`--log_file`	Directs the log to a file
`--mangled_names_in_messages`	Adds mangled names in messages
`--map`	Produces a map file
`--misrac1998`	Enables error messages specific to MISRA-C:1998. See the *IAR Embedded Workbench® MISRA C:1998 Reference Guide*.
`--misrac2004`	Enables error messages specific to MISRA-C:2004. See the *IAR Embedded Workbench® MISRA C:2004 Reference Guide*.
`--misrac_verbose`	Enables verbose logging of MISRA C checking. See the *IAR Embedded Workbench® MISRA C:1998 Reference Guide* and the *IAR Embedded Workbench® MISRA C:2004 Reference Guide*.
`--no_fragments`	Disables section fragment handling
`--no_library_search`	Disables automatic runtime library search
`--no_locals`	Removes local symbols from the ELF executable image.
`--no_remove`	Disables removal of unused sections
`--no_veneers`	Disables generation of veneers
`--no_warnings`	Disables generation of warnings
`--no_wrap_diagnostics`	Does not wrap long lines in diagnostic messages
`-o`	Sets the object filename
`--only_stdout`	Uses standard output only
`--ose_load_module`	Produces an OSE load module image
`--output`	Sets the object filename
`--pi_veneers`	Generates position independent veneers.
`--place_holder`	Reserve a place in ROM to be filled by some other tool, for example a checksum calculated by `ielftool`.
`--redirect`	Redirects a reference to a symbol to another symbol
`--remarks`	Enables remarks
`--semihosting`	Links with debug interface
`--silent`	Sets silent operation

Table 24: Linker options summary (Continued)

Command line option	Description
`--strip`	Removes debug information from the executable image
`--warnings_are_errors`	Warnings are treated as errors
`--warnings_affect_exit_code`	Warnings affects exit code

Table 24: Linker options summary (Continued)

Descriptions of options

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

Note that if you use the options page **Extra Options** to specify specific command line options, the IDE does not perform an instant check for consistency problems like conflicting options, duplication of options, or use of irrelevant options.

--BE8

Syntax

`--BE8`

Description

Use this option to specify the Byte Invariant Addressing mode.

This means that the linker reverses the byte order of the instructions, resulting in little-endian code and big-endian data. This is the default byte addressing mode for ARMv6 big-endian images. This is the only mode available for ARM v6M and ARM v7 with big-endian images.

Byte Invariant Addressing mode is only available on ARM processors that support ARMv6, ARM v6M, and ARM v7.

See also

Byte order, page 21, *Byte order*, page 210, *--BE32*, page 191, and *--endian*, page 170.

Project>Options>General Options>Target>Endian mode

--BE32

Syntax

`--BE32`

Description

Use this option to specify the legacy big-endian mode.

This produces big-endian code and data. This is the only byte-addressing mode for all big-endian images prior to ARMv6. This mode is also available for ARM v6 with big-endian, but not for ARM v6M or ARM v7.

See also

Byte order, page 21, *Byte order*, page 210, *--BE8*, page 191, and *--endian*, page 170.

Project>Options>General Options>Target>Endian mode

--config

Syntax

`--config filename`

Parameters

For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 156.

Description

Use this option to specify the configuration file to be used by the linker (the default filename extension is `icf`). If no configuration file is specified, a default configuration is used. This option can only be used once on the command line.

See also

The chapter *The linker configuration file*.

Project>Options>Linker>Config>Linker configuration file

--config_def

Syntax

`--config_def symbol[=constant_value]`

Parameters

symbol	The name of the symbol to be used in the configuration file. By default, the value 0 (zero) is used.
constant_value	The constant value of the configuration symbol.

Description

Use this option to define a constant configuration symbol to be used in the configuration file. This option has the same effect as the `define symbol` directive in the linker configuration file. This option can be used more that once on the command line.

See also

--define_symbol, page 193 and *Interaction between ILINK and the application*, page 56.

Project>Options>Linker>Config>Defined symbols for configuration file

--cpp_init_routine

Syntax `--cpp_init_routine routine`

Parameters

routine	A user-defined C++ dynamic initialization routine.

Description When using the IAR C/C++ compiler and the standard library, C++ dynamic initialization is handled automatically. In other cases you might need to use this option.

If any sections with the section type `INIT_ARRAY` or `PREINIT_ARRAY` are included in your application, the C++ dynamic initialization routine is considered to be needed. By default, this routine is named `__iar_cstart_call_ctors` and is called by the startup code in the standard library. Use this option if you are not using the standard library and require another routine to handle these section types.

To set this option, use **Project>Options>Linker>Extra Options**.

--cpu

Syntax `--cpu=core`

Parameters

core	Specifies a specific processor variant

Description Use this option to select the processor variant for which the code is to be generated. The default is ARM7TDMI.

See also --*cpu*, page 162 for a list of recognized cores and processor macrocells.

Project>Options>General Options>Target>Processor configuration

--define_symbol

Syntax `--define_symbol symbol[=constant_value]`

Parameters

symbol	The name of the constant symbol that can be used by the application. By default, the value 0 (zero) is used.
constant_value	The constant value of the symbol.

Description

Use this option to define a constant symbol that can be used by your application. If no value is specified, `0` is used. This option can be used more than once on the command line. Note that his option is different from the `define symbol` directive.

See also

--config_def, page 192 and *Interaction between ILINK and the application*, page 56.

Project>Options>Linker>#define>Defined symbols

--diag_error

Syntax

`--diag_error=tag[,tag,...]`

Parameters

tag	The number of a diagnostic message, for example the message number `Pe117`

Description

Use this option to reclassify certain diagnostic messages as errors. An error indicates a violation of the C or C++ language rules, of such severity that a violation of the linking rules of such severity that an executable image will not be generated. The exit code will be non-zero. This option may be used more than once on the command line.

Project>Options>Linker>Diagnostics>Treat these as errors

--diag_remark

Syntax

`--diag_remark=tag[,tag,...]`

Parameters

tag	The number of a diagnostic message, for example the message number `Pe177`

Description

Use this option to reclassify certain diagnostic messages as remarks. A remark is the least severe type of diagnostic message and indicates a construction that may cause strange behavior in the executable image. This option may be used more than once on the command line.

Note: By default, remarks are not displayed; use the `--remarks` option to display them.

Project>Options>Linker>Diagnostics>Treat these as remarks

--diag_suppress

Syntax `--diag_suppress=`*`tag`*`[,`*`tag,...`*`]`

Parameters

tag	The number of a diagnostic message, for example the message number `Pe117`

Description Use this option to suppress certain diagnostic messages. These messages will not be displayed. This option may be used more than once on the command line.

Project>Options>Linker>Diagnostics>Suppress these diagnostics

--diag_warning

Syntax `--diag_warning=`*`tag`*`[,`*`tag,...`*`]`

Parameters

tag	The number of a diagnostic message, for example the message number `Pe826`

Description Use this option to reclassify certain diagnostic messages as warnings. A warning indicates an error or omission that is of concern, but which will not cause the linker to stop before linking is completed. This option may be used more than once on the command line.

Project>Options>Linker>Diagnostics>Treat these as warnings

--diagnostics_tables

Syntax `--diagnostics_tables {`*`filename`*`|`*`directory`*`}`

Parameters For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 156.

Description Use this option to list all possible diagnostic messages in a named file.

This option cannot be given together with other options.

This option is not available in the IDE.

--entry

Syntax `--entry` *`symbol`*

Parameters

`symbol`	The name of the symbol to be treated as a root symbol and start label

Description Use this option to make a symbol be treated as a root symbol and the start label of the application. This is useful for loaders. If this option is not used, the default start symbol is `__iar_program_start`. A root symbol is kept whether or not it is referenced from the rest of the application, provided its module is included. A module in an object file is always included and a module part of a library is only included if needed.

Project>Options>Linker>Library>Override default program entry

--error_limit

Syntax `--error_limit=`*`n`*

Parameters

`n`	The number of errors before the linker stops linking. *`n`* must be a positive integer; `0` indicates no limit.

Description Use the `--error_limit` option to specify the number of errors allowed before the linker stops the linking. By default, 100 errors are allowed.

This option is not available in the IDE.

--export_builtin_config

Syntax `--export_builtin_config filename`

Parameters For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 156.

Description Exports the configuration used by default to a file.

This option is not available in the IDE.

-f

Syntax `-f filename`

Parameters For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 156.

Descriptions Use this option to make the linker read command line options from the named file, with the default filename extension `xcl`.

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

Both C and C++ style comments are allowed in the file. Double quotes behave in the same way as in the Microsoft Windows command line environment.

To set this option, use **Project>Options>Linker>Extra Options**.

--force_output

Syntax `--force_output`

Description Use this option to produce an output executable image regardless of any linking errors.

To set this option, use **Project>Options>Linker>Extra Options**

--image_input

Syntax `--image_input filename [symbol,[section[,alignment]]]`

Parameters

filename	The pure binary file containing the raw image you want to link
symbol	The symbol which the binary data can be referenced with.
section	The section where the binary data will be placed; default is `.text`.
alignment	The alignment of the section; default is `1`.

Description

Use this option to link pure binary files in addition to the ordinary input files. The file's entire contents are placed in the section, which means it can only contain pure binary data.

The section where the contents of the *filename* file are placed, is only included if the symbol *symbol* is required by your application. Use the `--keep` option if you want to force a reference to the section.

Example

```
--image_input bootstrap.abs,Bootstrap,CSTARTUPCODE,4
```

The contents of the pure binary file `bootstrap.abs` are placed in the section `CSTARTUPCODE`. The section where the contents are placed is 4-byte aligned and will only be included if your application (or the command line option `--keep`) includes a reference to the symbol `Bootstrap`.

See also

--*keep*, page 198.

Project>Options>Linker>Input>Raw binary image

--keep

Syntax `--keep symbol`

Parameters

symbol	The name of the symbol to be treated as a root symbol

Description

Normally, the linker keeps a symbol only if it is needed by your application. Use this option to make a symbol always be included in the final application.

Project>Options>Linker>Input>Keep symbols

--log

Syntax `--log topic,topic,...`

Parameters

`initialization`	Log initialization decisions
`modules`	Log module selections
`sections`	Log section selections

Description Use this option to make the linker log information to `stdout`. The log information can be useful for understanding why an executable image became the way it is.

See also *--log_file*, page 199.

Project>Options>Linker>List>Generate log

--log_file

Syntax `--log_file filename`

Parameters For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 156.

Description Use this option to direct the log output to the specified file.

See also *--log*, page 199.

Project>Options>Linker>List>Generate log

--mangled_names_in_messages

Syntax `--mangled_names_in_messages`

Descriptions Use this option to produce both mangled and unmangled names for C/C++ symbols in messages. Mangling is a technique used for mapping a complex C name or a C++ name (for example, for overloading) into a simple name. For example, `void h(int, char)` becomes `_Z1hic`.

This option is not available in the IDE.

--map

Syntax `--map {filename|directory}`

Description Use this option to produce a linker memory map file. The map file has the default filename extension `map`. The map file contains:

- Linking summary in the map file header which lists the version of the linker, the current date and time, and the command line that was used.
- Runtime attribute summary which lists AEABI attributes and IAR-specific runtime attributes.
- Placement summary which lists each section/block in address order, sorted by placement directives.
- Initialization table layout which lists the data ranges, packing methods, and compression ratios.
- Module summary which lists contributions from each module to the image, sorted by directory and library.
- Entry list which lists all public and some local symbols in alphabetical order, indicating which module they came from.
- Some of the bytes might be reported as *shared*.

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

This option can only be used once on the command line.

Project>Options>Linker>List>Generate linker map file

--no_fragments

Syntax `--no_fragments`

Description Use this option to disable section fragment handling. Normally, the toolset uses IAR proprietary information for transferring section fragment information to the linker. The linker uses this information to remove unused code and data, and thus further minimize the size of the executable image.

See also — *Keeping symbols and sections*, page 52.

To set this option, use **Project>Options>Linker>Extra Options**

--no_library_search

Syntax
`--no_library_search`

Description
Use this option to disable the automatic runtime library search. This option turns off the automatic inclusion of the correct standard libraries. This is useful, for example, if the application needs a user-built standard library, etc.

Project>Options>Linker>Library>Automatic runtime library selection

--no_locals

Syntax
`--no_locals`

Description
Use this option to remove local symbols from the ELF executable image.

Note: This option does not remove any local symbols from the DWARF information in the executable image.

Project>Options>Linker>Output

--no_remove

Syntax
`--no_remove`

Description
When this option is used, unused sections are not removed. In other words, each module that is included in the executable image contains all its original sections.

See also — *Keeping symbols and sections*, page 52.

To set this option, use **Project>Options>Linker>Extra Options**

--no_veneers

Syntax
```
--no_veneers
```

Description

Use this option to disable the insertion of veneers even though the executable image needs it. In this case, the linker will generate a relocation error for each reference that needs a veneer.

See also

Veneers, page 57.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--no_warnings

Syntax
```
--no_warnings
```

Description

By default, the linker issues warning messages. Use this option to disable all warning messages.

This option is not available in the IDE.

--no_wrap_diagnostics

Syntax
```
--no_wrap_diagnostics
```

Description

By default, long lines in diagnostic messages are broken into several lines to make the message easier to read. Use this option to disable line wrapping of diagnostic messages.

This option is not available in the IDE.

-o, --output

Syntax
```
-o {filename|directory}
--output {filename|directory}
```

Parameters

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 156.

Description

By default, the object executable image produced by the linker is located in a file with the name `a.out`. Use this option to explicitly specify a different output filename, which by default will have the filename extension `out`.

Project>Options>Linker>Output>Output file

--only_stdout

Syntax

```
--only_stdout
```

Description

Use this option to make the linker use the standard output stream (`stdout`) also for messages that are normally directed to the error output stream (`stderr`).

This option is not available in the IDE.

--ose_load_module

Syntax

```
--ose_load_module
```

Description

By default, the linker generates a ROMable executable image. Use this option to generate an executable image in the OSE load module image format instead.

Project>Options>Linker>Output

--output, -o

Syntax

```
--output {filename|directory}
-o {filename|directory}
```

Parameters

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 156.

Description

By default, the object executable image produced by the linker is located in a file with the name `a.out`. Use this option to explicitly specify a different output filename, which by default will have the filename extension `out`.

Project>Options>Linker>Output>Output file

--pi_veneers

Syntax
`--pi_veneers`

Description
Use this option to make the linker generate position-independent veneers. Note that this type of veneers is bigger and slower than normal veneers.

See also
Veneers, page 57.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--place_holder

Syntax
`--place_holder symbol[,size[,section[,alignment]]]`

Parameters

symbol	The name of the symbol to create
size	Size in ROM; by default 4 bytes
section	Section name to use; by default `.text`
alignment	Alignment of section; by default `1`

Description
Use this option to reserve a place in ROM to be filled by some other tool, for example a checksum calculated by `ielftool`. Each use of this linker option results in a section with the specified name, size, and alignment. The symbol can be used by your application to refer to the section.

Note: Like any other section, sections created by the `--place_holder` option will only be included in your application if the section appears to be needed. The `--keep` linker option, or the `keep` linker directive can be used for forcing such section to be included.

See also
IAR utilities, page 323.

To set this option, use **Project>Options>Linker>Extra Options**

--redirect

Syntax `--redirect from_symbol=to_symbol`

Parameters

from_symbol	The name of the source symbol
to_symbol	The name of the destination symbol

Description Use this option to change a reference from one symbol to another symbol.

To set this option, use **Project>Options>Linker>Extra Options**

--remarks

Syntax `--remarks`

Description The least severe diagnostic messages are called remarks. A remark indicates a source code construct that may cause strange behavior in the generated code. By default, the linker does not generate remarks. Use this option to make the linker generate remarks.

See also *Severity levels*, page 153.

Project>Options>Linker>Diagnostics>Enable remarks

--semihosting

Syntax `--semihosting[=iar_breakpoint]`

Parameters

iar_breakpoint	The IAR-specific mechanism can be used when debugging applications that use SWI/SVC extensively.

Description Use this option to include the debug interface—breakpoint mechanism—in the output image. If no parameter is specified, the SWI/SVC mechanism is included for ARM7/9/11, and the BKPT mechanism is included for Cortex-M.

See also *Low-level interface for debug support*, page 64.

Project>Options>General Options>Library Configuration>Semihosted

--silent

Syntax `--silent`

Description By default, the linker issues introductory messages and a final statistics report. Use this option to make the linker operate without sending these messages to the standard output stream (normally the screen).

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

This option is not available in the IDE.

--strip

Syntax `--strip`

Description By default, the linker retains the debug information from the input object files in the output executable image. Use this option to remove that information.

To set related options, choose:

Project>Options>Linker>Output>Include debug information in output

--warnings_affect_exit_code

Syntax `--warnings_affect_exit_code`

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

This option is not available in the IDE.

--warnings_are_errors

Syntax `--warnings_are_errors`

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

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

See also

--*diag_warning*, page 195 and --*diag_warning*, page 167.

Project>Options>Linker>Diagnostics>Treat all warnings as errors

Data representation

This chapter describes the data types, pointers, and structure types supported by the compiler.

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

Alignment

Every C data object has an alignment that controls how the object can be stored in memory. Should an object have an alignment of, for example, 4, it must be stored on an address that is divisible by 4.

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

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

Another processor might only be able to read 2 bytes at a time; in that environment, the alignment for a 4-byte `long` integer might be 2.

A structure type will have the same alignment as the structure member with the most strict alignment. To decrease the alignment requirements on the structure and its members, use `#pragma pack` or the `__packed` data type attribute.

All data types must have a size that is a multiple of their alignment. Otherwise, only the first element of an array would be guaranteed to be placed in accordance with the alignment requirements. This means that the compiler might add pad bytes at the end of the structure. For more information about pad bytes, see *Packed structure types*, page 217.

Note that with the `#pragma data_alignment` directive you can increase the alignment demands on specific variables.

ALIGNMENT ON THE ARM CORE

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 more efficiently only when the object is stored at an address divisible by 4.

Objects with alignment 4 must be stored at an address divisible by 4, while objects with alignment 2 must be stored at addresses divisible by 2.

The 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` compiler option; see *--endian*, page 170.

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. If you use the big-endian byte order, it might be necessary to use the `#pragma bitfields=reversed` directive to be compatible with code for other compilers and I/O register definitions of some devices, see *Bitfields*, page 212.

Note: There are two variants of the big-endian mode, BE8 and BE32, which you specify at link time. In BE8 data is big-endian and code is little-endian. In BE32 both data and code are big-endian. In architectures before v6, the BE32 endian mode is used, and after v6 the BE8 mode is used. In the v6 (ARM11) architecture, both big-endian modes are supported.

Basic data types

The compiler supports both all ISO/ANSI C basic data types and some additional types.

INTEGER TYPES

This table gives the size and range of each integer data type:

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

Table 25: Integer types

Data type	Size	Range	Alignment
signed int	32 bits	-2^{31} to $2^{31}-1$	4
unsigned int	32 bits	0 to $2^{32}-1$	4
signed long	32 bits	-2^{31} to $2^{31}-1$	4
unsigned long	32 bits	0 to $2^{32}-1$	4
signed long long	64 bits	-2^{63} to $2^{63}-1$	8
unsigned long long	64 bits	0 to $2^{64}-1$	8

Table 25: Integer types (Continued)

Signed variables are represented using the two's complement form.

Bool

The `bool` data type is supported by default in the C++ language. If you have enabled language extensions, the `bool` type can also be used in C source code if you include the file `stdbool.h`. This will also enable the boolean values `false` and `true`.

The enum type

The compiler will use the smallest type required to hold `enum` constants, preferring `signed` rather than `unsigned`.

When IAR Systems language extensions are enabled, and in C++, the `enum` constants and types can also be of the type `long`, `unsigned long`, `long long`, or `unsigned long long`.

To make the compiler use a larger type than it would automatically use, define an `enum` constant with a large enough value. For example:

```
/* Disables usage of the char type for enum */
enum Cards{Spade1, Spade2,
           DontUseChar=257};
```

Read also about the compiler option *--enum_is_int*, page 171.

The char type

The `char` type is by default unsigned in the compiler, but the `--char_is_signed` compiler option allows you to make it signed. Note, however, that the library is compiled with the `char` type as unsigned.

The wchar_t type

The `wchar_t` data type is an integer type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locals.

The wchar_t data type is supported by default in the C++ language. To use the wchar_t type also in C source code, you must include the file stddef.h from the runtime library.

Bitfields

In ISO/ANSI C, int, signed int, and unsigned int can be used as the base type for integer bitfields. It is implementation defined whether the type specified by int is the same as signed int or unsigned int. In the IAR C/C++ Compiler for ARM, bitfields specified as int are treated as unsigned int. Furthermore, any integer type can be used as the base type when language extensions are enabled. Bitfields in expressions will have the same data type as the integer base type.

The compiler places bitfield members based on the byte order mode that is used. By default in little-endian mode, the compiler places bitfield members from the least significant to the most significant bit in the container type. And by default in big-endian mode, the compiler places bitfield members from the most significant to the least significant bit in the container type. A bitfield is assigned to the last available container of its base type which has enough unassigned bits to contain the entire bitfield. This means that bitfield containers can overlap other structure members as long as the order of the fields in the structure is preserved, for example in big-endian mode:

```
struct example
{
  char  a;
  short b : 10;
  int   c : 6;
};
```

Here the first declaration creates an unsigned character which is allocated to bits 24 through 31. The second declaration creates a signed short integer member of size 10 bits. This member is allocated to bits 15 through 6 as it will not fit in the remaining 8 bits of the first short integer container. The last bitfield member declared is placed in the bits 0 through 5. If seen as a 32-bit value, the structure looks like this in memory:

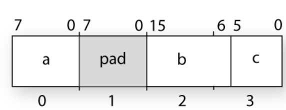


Figure 14: Layout of bitfield members in big-endian mode

Use the directive `#pragma bitfields=disjoint_types` to force the bitfield containers to be disjoint, or in other words, not to overlap. The layout of the above example structure would then become:

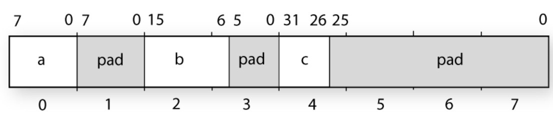


Figure 15: Layout of bitfield members forced to be disjoint in big-endian mode

Use the directive `#pragma bitfields=reversed_disjoint_types` to place the bitfield members from the least significant bit to the most significant bit in non-overlapping storage containers.

FLOATING-POINT TYPES

In the IAR C/C++ Compiler for ARM, floating-point values are represented in standard IEEE 754 format. The sizes for the different floating-point types are:

Type	Size	Range (+/-)	Decimals	Exponent	Mantissa
float	32 bits	±1.18E-38 to ±3.40E+38	7	8 bits	23 bits
double	64 bits	±2.23E-308 to ±1.79E+308	15	11 bits	52 bits
long double	64 bits	±2.23E-308 to ±1.79E+308	15	11 bits	52 bits

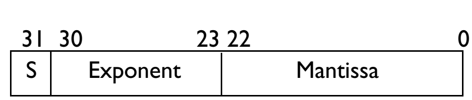
Table 26: Floating-point types

For Cortex-M0 and Cortex-M1, the compiler does not support subnormal numbers. All operations that should produce subnormal numbers will instead generate zero. For information about the representation of subnormal numbers for other cores, see *Representation of special floating-point numbers*, page 214.

Exception flags according to the IEEE 754 standard are not supported. The alignment for the `float` type is 4, and for the `long double` type it is 8.

32-bit floating-point format

The representation of a 32-bit floating-point number as an integer is:



The exponent is 8 bits, and the mantissa is 23 bits.

The value of the number is:

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

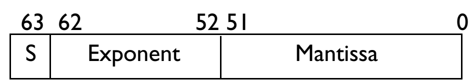
The range of the number is:

$\pm 1.18E-38$ to $\pm 3.40E+38$

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

64-bit floating-point format

The representation of a 64-bit floating-point number as an integer is:



The exponent is 11 bits, and the mantissa is 52 bits.

The value of the number is:

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

The range of the number is:

$\pm 2.23E-308$ to $\pm 1.79E+308$

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

Representation of special floating-point numbers

This list describes the representation of special floating-point numbers:

- Zero is represented by zero mantissa and exponent. The sign bit signifies positive or negative zero.
- Infinity is represented by setting the exponent to the highest value and the mantissa to zero. The sign bit signifies positive or negative infinity.
- Not a number (NaN) is represented by setting the exponent to the highest positive value and at least one bit set in the 20 most significant bits of the mantissa. Remaining bits are zero.
- Subnormal numbers are used for representing 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 was 1. Unlike normal numbers, denormalized numbers do not have an implicit 1 as the most significant bit (the 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 compiler has two basic types of pointers: function pointers and data pointers.

FUNCTION POINTERS

The size of function pointers is always 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 these characteristics:

- Casting a *value* of an integer type to a pointer of a smaller type is performed by truncation
- Casting a *value* of an unsigned integer type to a pointer of a larger type is performed by zero extension
- Casting a *value* of a signed integer type to a pointer of a larger type is performed by sign 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 IAR C/C++ Compiler for ARM, 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 IAR C/C++ Compiler for ARM, 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 IAR C/C++ Compiler for ARM, the size of `intptr_t` is 32 bits.

uintptr_t

`uintptr_t` is equivalent to `intptr_t`, with the exception that it is unsigned.

Structure types

The members of a `struct` are stored sequentially in the order in which they are declared: the first member has the lowest memory address.

ALIGNMENT

The `struct` and `union` types have the same alignment as the member with the highest alignment requirement. The size of a `struct` is also adjusted to allow arrays of aligned structure objects.

GENERAL LAYOUT

Members of a `struct` are always allocated in the order specified in the declaration. Each member is placed in the `struct` according to the specified alignment (offsets).

Example

```
struct First
{
  char c;
  short s;
} s;
```

This diagram shows the layout in memory:

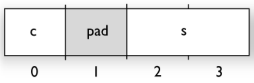


Figure 16: Structure layout

The alignment of the structure is 2 bytes, and a pad byte must be inserted to give `short s` the correct alignment.

PACKED STRUCTURE TYPES

The `__packed` data type attribute or the `#pragma pack` directive is used for relaxing the alignment requirements of the members of a structure. This changes the layout of the structure. The members are placed in the same order as when declared, but there might be less pad space between members.

Note that accessing an object that is not correctly aligned requires code that is both larger and slower. If such structure members are accessed many times, it is usually better to construct the correct values in a `struct` that is not packed, and access this `struct` instead.

Special care is also needed when creating and using pointers to misaligned members. For direct access to misaligned members in a packed `struct`, the compiler will emit the correct (but slower and larger) code when needed. However, when a misaligned member is accessed through a pointer to the member, the normal (smaller and faster) code is used. In the general case, this will not work.

Example

This example declares a packed structure:

```
#pragma pack(1)
struct S
{
  char c;
  short s;
};

#pragma pack()
```

In this example, the structure `S` has this memory layout:

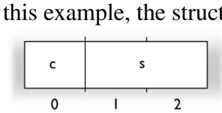


Figure 17: Packed structure layout

This example declares a new non-packed structure, `S2`, that contains the structure `s` declared in the previous example:

```
struct S2
{
  struct S s;
  long l;
};
```

S2 has this memory layout

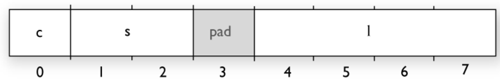


Figure 18: Packed structure layout

The structure S will use the memory layout, size, and alignment described in the previous example. The alignment of the member l is 4, which means that alignment of the structure S2 will become 2.

For more information, see *Alignment of elements in a structure*, page 129.

Type qualifiers

According to the ISO/ANSI C standard, `volatile` and `const` are type qualifiers.

DECLARING OBJECTS VOLATILE

There are three main reasons for declaring an object `volatile`:

- Shared access; the object is shared between several tasks in a multitasking environment
- Trigger access; as for a memory-mapped SFR where the fact that an access occurs has an effect
- Modified access; where the contents of the object can change in ways not known to the compiler.

Definition of access to volatile objects

The ISO/ANSI standard defines an abstract machine, which governs the behavior of accesses to `volatile` declared objects. In general and in accordance to the abstract machine, the compiler:

- Considers each read and write access to an object declared `volatile` as an access
- The unit for the access is either the entire object or, for accesses to an element in a composite object—such as an array, struct, class, or union—the element. For example:

```
char volatile a;
a = 5;   /* A write access */
a += 6;  /* First a read then a write access */
```

- An access to a bitfield is treated as an access to the underlying type.

However, these rules are not detailed enough to handle the hardware-related requirements. The rules specific to the IAR C/C++ Compiler for ARM are described below.

Rules for accesses

In the IAR C/C++ Compiler for ARM, accesses to `volatile` declared objects are subject to these rules:

- All accesses are preserved
- All accesses are complete, that is, the whole object is accessed
- All accesses are performed in the same order as given in the abstract machine
- All accesses are atomic, that is, they cannot be interrupted.

The compiler adheres to these rules for accesses to all 8-, 16-, and 32-bit scalar types, except for accesses to unaligned 16- and 32-bit fields in packed structures.

For all other object types, only the rule that states that all accesses are preserved applies.

DECLARING OBJECTS CONST

The `const` type qualifier is used for indicating that a data object, accessed directly or via a pointer, is non-writable. A pointer to `const` declared data can point to both constant and non-constant objects. It is good programming practice to use `const` declared pointers whenever possible because this improves the compiler's possibilities to optimize the generated code and reduces the risk of application failure due to erroneously modified data.

Static and global objects declared `const` are allocated in ROM.

In C++, objects that require runtime initialization cannot be placed in ROM.

Data types in C++

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

Compiler extensions

This chapter gives a brief overview of the compiler extensions to the ISO/ANSI C standard. All extensions can also be used for the C++ programming language. More specifically the chapter describes the available C language extensions.

Compiler extensions overview

The compiler offers the standard features of ISO/ANSI C and a wide set of extensions, ranging from features specifically tailored for efficient programming in the embedded industry to the relaxation of some minor standards issues.

You can find the extensions available as:

- C/C++ language extensions

  For a summary of available language extensions, see *C language extensions*, page 222. For reference information about the extended keywords, see the chapter *Extended keywords*. For information about C++, the two levels of support for the language, and C++ language extensions; see the chapter *Using C++*.

- Pragma directives

  The `#pragma` directive is defined by the ISO/ANSI C standard and is a mechanism for using vendor-specific extensions in a controlled way to make sure that the source code is still portable.

  The compiler provides a set of predefined pragma directives, which can be used for controlling the behavior of the compiler, for example how it allocates memory, whether it allows extended keywords, and whether it outputs warning messages. Most pragma directives are preprocessed, which means that macros are substituted in a pragma directive. The pragma directives are always enabled in the compiler. For several of them there is also a corresponding C/C++ language extension. For a list of available pragma directives, see the chapter *Pragma directives*.

- Preprocessor extensions

  The preprocessor of the compiler adheres to the ISO/ANSI standard. The compiler also makes several preprocessor-related extensions available to you. For more information, see the chapter *The preprocessor*.

- Intrinsic functions

  The intrinsic functions provide direct access to low-level processor operations and can be very useful in, for example, time-critical routines. The intrinsic functions compile into inline code, either as a single instruction or as a short sequence of instructions. To read more about using intrinsic functions, see *Mixing C and assembler*, page 91. For a list of available functions, see the chapter *Intrinsic functions*.

- Library functions

  The IAR DLIB Library provides most of the important C and C++ library definitions that apply to embedded systems. The library also provides some extensions, partly taken from the C99 standard. For more information, see *IAR DLIB Library*, page 290.

Note: Any use of these extensions, except for the pragma directives, makes your application inconsistent with the ISO/ANSI C standard.

ENABLING LANGUAGE EXTENSIONS

In the IDE, language extensions are enabled by default.

For information about how to enable and disable language extensions from the command line, see the compiler options *-e*, page 169, and *--strict_ansi*, page 187.

C language extensions

This section gives a brief overview of the C language extensions available in the compiler. The compiler provides a wide set of extensions, so to help you to find the extensions required by your application, the extensions are grouped according to their expected usefulness. In short, this means:

- Important language extensions—extensions specifically tailored for efficient embedded programming, typically to meet memory restrictions
- Useful language extensions—features considered useful and typically taken from related standards, such as C99 and C++
- Minor language extensions, that is, the relaxation of some minor standards issues and also some useful but minor syntax extensions.

IMPORTANT LANGUAGE EXTENSIONS

The following language extensions available both in the C and the C++ programming languages are well suited for embedded systems programming:

- Type attributes, and object attributes

  For information about the related concepts, the general syntax rules, and for reference information, see the chapter *Extended keywords*.

- Placement at an absolute address or in a named section

  The `@` operator or the directive `#pragma location` can be used for placing global and static variables at absolute addresses, or placing a variable or function in a named section. For more information about using these primitives, see *Controlling data and function placement in memory*, page 131, and *location*, page 251.

- Alignment

  Each data type has its own alignment, for more details, see *Alignment*, page 209. If you want to change the alignment, the `__packed` data type attribute, and the `#pragma pack` and `#pragma data_alignment` directives are available. If you want to use the alignment of an object, use the `__ALIGNOF__()` operator.

  The `__ALIGNOF__` operator is used for accessing the alignment of an object. It takes one of two forms:

  - `__ALIGNOF__ (`*`type`*`)`
  - `__ALIGNOF__ (`*`expression`*`)`

  In the second form, the expression is not evaluated.

- Anonymous structs and unions

  C++ includes a feature named anonymous unions. The compiler allows a similar feature for both structs and unions in the C programming language. For more information, see *Anonymous structs and unions*, page 129.

- Bitfields and non-standard types

  In ISO/ANSI C, a bitfield must be of type `int` or `unsigned int`. Using IAR Systems language extensions, any integer type or enumeration can be used. The advantage is that the struct will sometimes be smaller. This matches G.5.8 in the appendix of the ISO standard, *ISO Portability Issues*. For more information, see *Bitfields*, page 212.

Dedicated section operators

The compiler supports for these built-in section operators: `__section_begin`, `__section_end`, and `__section_size`.

These operators behave syntactically as if declared like this:

```
void * __section_begin(char const * section)
void * __section_end(char const * section)
size_t * __section_size(char const * section)
```

These operators can be used on named sections or on named blocks defined in the linker configuration file.

The `__section_begin` operator returns the address of the first byte of the named section or block.

The `__section_end` operator returns the address of the first byte *after* the named *`section`* or block.

The `__section_size` operator returns the size of the named section or block in bytes.

Note: The aliases `__segment_begin`/`__sfb`, `__segment_end`/`__sfe`, and `__segment_size`/`__sfs` can also be used.

When using the `@` operator or the `#pragma location` directive to place a data object or a function in a user-defined section, or when using named blocks in the linker configuration file, the section operators can be used for getting the start and end of the memory range where the sections or blocks were placed.

The named *section* must be a string literal and *section* must have been declared earlier with the `#pragma section` directive. The type of the `__section_begin` operator is a pointer to `void`. Note that you must enable language extensions to use these operators.

The operators are implemented in terms of *symbols* with dedicated names, and will appear in the linker map file under these names:

Operator	Symbol
`__section_begin(sec)`	`sec$$Base`
`__section_end(sec)`	`sec$$Limit`
`__section_size(sec)`	`sec$$Length`

Table 27: Section operators and their symbols

Note that the linker will not necessarily place sections with the same name contiguously when these operators are not used. Using one of these operators (or the equivalent symbols) will cause the linker to behave as if the sections were in a named block. This is to assure that the sections are placed contiguously, so that the operators can be assigned meaningful values. If this is in conflict with the section placement as specified in the linker configuration file, the linker will issue an error.

Example

In this example, the type of the `__section_begin` operator is `void *`.

```
#pragma section="MYSECTION"
...
section_start_address = __section_begin("MYSECTION");
```

See also *section*, page 256, and *location*, page 251.

USEFUL LANGUAGE EXTENSIONS

This section lists and briefly describes useful extensions, that is, useful features typically taken from related standards, such as C99 and C++:

- Inline functions

  The `#pragma inline` directive, alternatively the `inline` keyword, advises the compiler that the function whose declaration follows immediately after the directive should be inlined. This is similar to the C++ keyword `inline`. For more information, see *inline*, page 250.

- Mixing declarations and statements

  It is possible to mix declarations and statements within the same scope. This feature is part of the C99 standard and C++.

- Declaration in `for` loops

  It is possible to have a declaration in the initialization expression of a `for` loop, for example:

  ```
  for (int i = 0; i < 10; ++i)
  {
    /* Do something here. */
  }
  ```

  This feature is part of the C99 standard and C++.

- The `bool` data type

  To use the `bool` type in C source code, you must include the file `stdbool.h`. This feature is part of the C99 standard and C++. (The `bool` data type is supported by default in C++.)

- C++ style comments

  C++ style comments are accepted. A C++ style comment starts with the character sequence `//` and continues to the end of the line. For example:

  ```
  // The length of the bar, in centimeters.
  int length;
  ```

  This feature is copied from the C99 standard and C++.

Inline assembler

Inline assembler can be used for inserting assembler instructions in the generated function. This feature is part of the C99 standard and C++.

The `asm` and `__asm` extended keywords both insert an assembler instruction. However, when compiling C source code, the `asm` keyword is not available when the option `--strict_ansi` is used. The `__asm` keyword is always available.

Note: Not all assembler directives or operators can be inserted using this keyword.

The syntax is:

```
asm ("string");
```

The string can be a valid assembler instruction or a data definition assembler directive, but not a comment. You can write several consecutive inline assembler instructions, for example:

```
asm ("Label:      nop\n"
     "            b Label");
```

where `\n` (new line) separates each new assembler instruction. Note that you can define and use local labels in inline assembler instructions.

For more information about inline assembler, see *Mixing C and assembler*, page 91.

Compound literals

To create compound literals you can use this syntax:

```
/* Create a pointer to an anonymous array */
int *p = (int []) {1, 2, 3};

/* Create a pointer to an anonymous structX */
structX *px = &(structX) {5, 6, 7};
```

Note:

- A compound literal can be modified unless it is declared `const`
- Compound literals are not supported in Embedded C++ and Extended EC++.
- This feature is part of the C99 standard.

Incomplete arrays at end of structs

The last element of a `struct` can be an incomplete array. This is useful for allocating a chunk of memory that contains both the structure and a fixed number of elements of the array. The number of elements can vary between allocations.

This feature is part of the C99 standard.

Note: The array cannot be the only member of the `struct`. If that was the case, then the size of the `struct` would be zero, which is not allowed in ISO/ANSI C.

Example

```
struct str
{
  char a;
  unsigned long b[];
};
```

```
struct str * GetAStr(int size)
{
  return malloc(sizeof(struct str) +
                sizeof(unsigned long) * size);
}

void UseStr(struct str * s)
{
  s->b[10] = 0;
}
```

The incomplete array will be aligned in the structure just like any other member of the structure. For more information about structure alignment, see *Structure types*, page 216.

Hexadecimal floating-point constants

Floating-point constants can be given in hexadecimal style. The syntax is `0x`*`MANT`*`p{+|-}`*`EXP`*, where *`MANT`* is the mantissa in hexadecimal digits, including an optional `.` (decimal point), and *`EXP`* is the exponent with decimal digits, representing an exponent of 2. This feature is part of the C99 standard.

Examples

`0x1p0` is `1`

`0xA.8p2` is `10.5*2`^{2}

Designated initializers in structures and arrays

Any initialization of either a structure (`struct` or `union`) or an array can have a designation. A designation consists of one or more designators followed by an initializer. A designator for a structure is specified as `.`*`elementname`* and for an array `[`*`constant index expression`*`]`. Using designated initializers is not supported in C++.

Examples

This definition shows a `struct` and its initialization using designators:

```
struct
{
  int i;
  int j;
  int k;
  int l;
  short array[10];
} u =
{
  .l = 6,          /* initialize l to 6 */
  .j = 6,          /* initialize j to 6 */
  8,               /* initialize k to 8 */
  .array[7] = 2,   /* initialize element 7 to 2 */
  .array[3] = 2,   /* initialize element 3 to 2 */
  5,               /* array[4] = 5 */
  .k = 4           /* reinitialize k to 4 */
};
```

Note that a designator specifies the destination element of the initialization. Note also that if one element is initialized more than once, it is the last initialization that will be used.

To initialize an element in a union other than the first, do like this:

```
union
{
  int i;
  float f;
} y = {.f = 5.0};
```

To set the size of an array by initializing the last element, do like this:

```
char array[] = {[10] = 'a'};
```

MINOR LANGUAGE EXTENSIONS

This section lists and briefly describes minor extensions, that is, the relaxation of some standards issues and also some useful but minor syntax extensions:

- Arrays of incomplete types

  An array can have an incomplete `struct`, `union`, or `enum` type as its element type. The types must be completed before the array is used (if it is), or by the end of the compilation unit (if it is not).

- Forward declaration of `enum` types

  The IAR Systems language extensions allow that you first declare the name of an `enum` and later resolve it by specifying the brace-enclosed list.

- Missing semicolon at end of `struct` or `union` specifier

  A warning is issued if the semicolon at the end of a `struct` or `union` specifier is missing.

- Null and `void`

  In operations on pointers, a pointer to `void` is always implicitly converted to another type if necessary, and a null pointer constant is always implicitly converted to a null pointer of the right type if necessary. In ISO/ANSI C, some operators allow such things, while others do not allow them.

- Casting pointers to integers in static initializers

  In an initializer, a pointer constant value can be cast to an integral type if the integral type is large enough to contain it. For more information about casting pointers, see *Casting*, page 215.

- Taking the address of a register variable

  In ISO/ANSI C, it is illegal to take the address of a variable specified as a register variable. The compiler allows this, but a warning is issued.

- Duplicated size and sign specifiers

  Should the size or sign specifiers be duplicated (for example, `short short` or `unsigned unsigned`), an error is issued.

- `long float` means `double`

  The type `long float` is accepted as a synonym for `double`.

- Repeated `typedef` declarations

  Redeclarations of `typedef` that occur in the same scope are allowed, but a warning is issued.

- Mixing pointer types

  Assignment and pointer difference is allowed between pointers to types that are interchangeable but not identical; for example, `unsigned char *` and `char *`. This includes pointers to integral types of the same size. A warning is issued.

  Assignment of a string constant to a pointer to any kind of character is allowed, and no warning is issued.

- Non-top level `const`

  Assignment of pointers is allowed in cases where the destination type has added type qualifiers that are not at the top level (for example, `int **` to `int const **`). Comparing and taking the difference of such pointers is also allowed.

- Non-`lvalue` arrays

  A non-`lvalue` array expression is converted to a pointer to the first element of the array when it is used.

- Comments at the end of preprocessor directives

  This extension, which makes it legal to place text after preprocessor directives, is enabled, unless strict ISO/ANSI mode is used. The purpose of this language extension is to support compilation of legacy code; we do *not* recommend that you write new code in this fashion.

- An extra comma at the end of `enum` lists

  Placing an extra comma is allowed at the end of an `enum` list. In strict ISO/ANSI mode, a warning is issued.

- A label preceding a `}`

  In ISO/ANSI C, a label must be followed by at least one statement. Therefore, it is illegal to place the label at the end of a block. The compiler issues a warning.

Note: This also applies to the labels of `switch` statements.

- Empty declarations

  An empty declaration (a semicolon by itself) is allowed, but a remark is issued (provided that remarks are enabled).

- Single-value initialization

  ISO/ANSI C requires that all initializer expressions of static arrays, `structs`, and `unions` are enclosed in braces.

  Single-value initializers are allowed to appear without braces, but a warning is issued. The compiler accepts this expression:

```
struct str
{
  int a;
} x = 10;
```

- Declarations in other scopes

  External and static declarations in other scopes are visible. In the following example, the variable `y` can be used at the end of the function, even though it should only be visible in the body of the `if` statement. A warning is issued.

```
int test(int x)
{
  if (x)
  {
    extern int y;
    y = 1;
  }

  return y;
}
```

- Expanding function names into strings with the function as context

  Use any of the symbols `__func__` or `__FUNCTION__` inside a function body to make the symbol expand into a string, with the function name as context. Use the symbol `__PRETTY_FUNCTION__` to also include the parameter types and return type. The result might, for example, look like this if you use the `__PRETTY_FUNCTION__` symbol:

```
"void func(char)"
```

  These symbols are useful for assertions and other trace utilities and they require that language extensions are enabled, see *-e*, page 169.

Extended keywords

This chapter describes the extended keywords that support specific features of the ARM core and the general syntax rules for the keywords. Finally the chapter gives a detailed description of each keyword.

General syntax rules for extended keywords

To understand the syntax rules for the extended keywords, it is important to be familiar with some related concepts.

The compiler provides a set of attributes that can be used on functions or data objects to 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
- Object attributes affect the *internal functionality* of the data object or function.

The syntax for the keywords differs slightly depending on whether it is a type attribute or an object attribute, and whether it is applied to a data object or a function.

For detailed information about each attribute, see *Descriptions of extended keywords*, page 237.

Note: The extended keywords are only available when language extensions are enabled in the compiler.

In the IDE, language extensions are enabled by default.

Use the `-e` compiler option to enable language extensions. See *-e*, page 169 for additional information.

TYPE ATTRIBUTES

Type attributes define how a function is called, or how a data object is accessed. This means that if you use a type attribute, it must be specified both when a function or data object is defined and when it is declared.

You can either place the type attributes directly in your source code, or use the pragma directive `#pragma type_attribute`.

These general type attributes are available:

- *Function type attributes* affect how the function should be called: `__arm`, `__fiq`, `__interwork`, `__irq`, `__swi`, `__task`, and `__thumb`

- *Data type attributes*: `__big_endian`, `const`, `__little_endian`, `__packed`, and `volatile`

You can specify as many type attributes as required for each level of pointer indirection.

To read more about the type qualifiers `const` and `volatile`, see *Type qualifiers*, page 218.

Syntax for type attributes used on data objects

In general, type attributes for data objects follow the same syntax as the type qualifiers `const` and `volatile`.

The following declaration assigns the `__little_endian` type attribute to the variables `i` and `j`; in other words, the variable `i` and `j` will be accessed with little endian byte order. The variables `k` and `l` behave in the same way:

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

Note that the attribute affects both identifiers.

This declaration of `i` and `j` is equivalent with the previous one:

```
#pragma type_attribute=__little_endian
int i, j;
```

The advantage of using pragma directives for specifying keywords is that it offers you a method to make sure that the source code is portable.

Syntax for type attributes on data pointers

The syntax for declaring pointers using type attributes follows the same syntax as the type qualifiers `const` and `volatile`:

`int __little_endian * p;`	The `int` object will be accessed in little endian byte order.
`int * __little_endian p;`	The pointer will be accessed in little endian byte order.
`__little_endian int * p;`	The pointer will be accessed in little endian byte order.

Syntax for type attributes on functions

The syntax for using type attributes on functions differs slightly from the syntax of type attributes on data objects. For functions, the attribute must be placed either in front of the return type, or in parentheses, for example:

```
__irq __arm void my_handler(void);
```

or

```
void (__irq __arm my_handler)(void);
```

This declaration of my_handler is equivalent with the previous one:

```
#pragma type_attribute=__irq __arm
void my_handler(void);
```

OBJECT ATTRIBUTES

Object attributes affect the internal functionality of functions and data objects, but not how the function is called or how the data is accessed. This means that an object attribute does not need to be present in the declaration of an object.

These 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, @, __root, and __weak,
- Object attributes that can be used for functions: __intrinsic, __nested, __noreturn, and __ramfunc.

You can specify as many object attributes as required for a specific function or data object.

For more information about location and @, see *Controlling data and function placement in memory*, page 131.

Syntax for object attributes

The object attribute must be placed in front of the type. For example, to place myarray in memory that is not initialized at startup:

```
__no_init int myarray[10];
```

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

```
#pragma object_attribute=__no_init
int myarray[10];
```

Note: Object attributes cannot be used in combination with the typedef keyword.

Summary of extended keywords

This table summarizes the extended keywords:

Extended keyword	Description
`__arm`	Makes a function execute in ARM mode
`__big_endian`	Declares a variable to use the big endian byte order
`__fiq`	Declares a fast interrupt function
`__interwork`	Declares a function to be callable from both ARM and Thumb mode
`__intrinsic`	Reserved for compiler internal use only
`__irq`	Declares an interrupt function
`__little_endian`	Declares a variable to use the little endian byte order
`__nested`	Allows an `__irq` declared interrupt function to be nested, that is, interruptible by the same type of interrupt
`__no_init`	Supports non-volatile memory
`__noreturn`	Informs the compiler that the function will not return
`__packed`	Decreases data type alignment to 1
`__ramfunc`	Makes a function execute in RAM
`__root`	Ensures that a function or variable is included in the object code even if unused
`__swi`	Declares a software interrupt function
`__task`	Relaxes the rules for preserving registers
`__thumb`	Makes a function execute in Thumb mode
`__weak`	Declares a symbol to be externally weakly linked

Table 28: Extended keywords summary

Descriptions of extended keywords

These sections give detailed information about each extended keyword.

__arm

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 233.

Description

The `__arm` 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`.

Note: Non-interwork ARM functions cannot be called from Thumb mode.

Example

```
__arm int func1(void);
```

See also

__interwork, page 238.

__big_endian

Syntax

Follows the generic syntax rules for type attributes that can be used on data objects, see *Type attributes*, page 233.

Description

The `__big_endian` keyword is used for accessing a variable that is stored in the big-endian byte order regardless of what byte order the rest of the application uses. The `__big_endian` keyword is available when you compile for ARMv6 or higher.

Example

```
__big_endian long my_variable;
```

See also

__little_endian, page 238.

__fiq

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 233.

Description

The `__fiq` keyword declares a fast interrupt function. All interrupt functions must be compiled in ARM mode. A function declared `__fiq` does not accept parameters and does not have a return value.

Example

```
__fiq __arm void interrupt_function(void);
```

__interwork

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 233.

Description

A function declared `__interwork` can be called from functions executing in either ARM or Thumb mode.

Note: By default, functions are interwork when the `--interwork` compiler option is used, and when the `--cpu` option is used and it specifies a core where interwork is default.

Example

```
typedef void (__thumb __interwork *IntHandler)(void);
```

__intrinsic

Description

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

__irq

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 233.

Description

The `__irq` keyword declares an interrupt function. All interrupt functions must be compiled in ARM mode. A function declared `__irq` does not accept parameters and does not have a return value.

Example

```
__irq __arm void interrupt_function(void);
```

__little_endian

Syntax

Follows the generic syntax rules for type attributes that can be used on data objects, see *Type attributes*, page 233.

Description

The `__little_endian` keyword is used for accessing a variable that is stored in the little-endian byte order regardless of what byte order the rest of the application uses. The `__little_endian` keyword is available when you compile for ARMv6 or higher.

Example

```
__little_endian long my_variable;
```

See also *__big_endian*, page 237.

__nested

Syntax

Follows the generic syntax rules for object attributes that can be used on functions, see *Object attributes*, page 235.

Description

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.

Note: The `__nested` keyword requires the processor mode to be in either User or System mode.

Example

```
__irq __nested __arm void interrupt_handler(void);
```

See also *Nested interrupts*, page 33.

__no_init

Syntax

Follows the generic syntax rules for object attributes, see *Object attributes*, page 235.

Description

Use the `__no_init` keyword to place a data object in non-volatile memory. This means that the initialization of the variable, for example at system startup, is suppressed.

Example

```
__no_init int myarray[10];
```

See also *Do not initialize directive*, page 308.

__noreturn

Syntax

Follows the generic syntax rules for object attributes, see *Object attributes*, page 235.

Description

The `__noreturn` keyword can be used on a function to inform the compiler that the function will not return. If you use this keyword on such functions, the compiler can optimize more efficiently. Examples of functions that do not return are `abort` and `exit`.

Example

```
__noreturn void terminate(void);
```

__packed

Syntax

Follows the generic syntax rules for type attributes that can be used on data, see *Type attributes*, page 233.

Description

Use the `__packed` keyword to decrease the data type alignment to 1. `__packed` can be used for two purposes:

- When used with a `struct` or `union` type definition, the maximum alignment of members of that `struct` or `union` is set to 1, to eliminate any gaps between the members. The type of each members also receives the `__packed` type attribute.
- When used with any other type, the resulting type is the same as the type without the `__packed` type attribute, but with an alignment of 1. Types that already have an alignment of 1 are not affected by the `__packed` type attribute.

A normal pointer can be implicitly converted to a pointer to `__packed`, but the reverse conversion requires a cast.

Note: Accessing data types at other alignments than their natural alignment can result in code that is significantly larger and slower.

Example

```
__packed struct X {char ch; int i;};           /* No pad bytes */
void foo (struct X * xp)         /* No need for __packed here */
{
  int * p1         = &xp->1;/* Error:"int *">"int __packed *" */
  int __packed * p2 = &xp->i;                          /* OK */
  char * p2         = &xp->ch;       /* OK, char not affected */
}
```

See also

pack, page 253.

__ramfunc

Syntax

Follows the generic syntax rules for object attributes, see *Object attributes*, page 235.

Description

The `__ramfunc` keyword makes a function execute in RAM. Two code sections 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 section named `CODE_I`.

Example

```
__ramfunc int FlashPage(char * data, char * page);
```

See also

To read more about `__ramfunc` declared functions in relation to breakpoints, see the *IAR Embedded Workbench® IDE User Guide for ARM®*.

__root

Syntax

Follows the generic syntax rules for object attributes, see *Object attributes*, page 235.

Description

A function or variable with the `__root` attribute is kept whether or not it is referenced from the rest of the application, provided its module is included. Program modules are always included and library modules are only included if needed.

Example

```
__root int myarray[10];
```

See also

To read more about root symbols and how they are kept, see the .

__swi

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 233.

Description

The `__swi` keyword declares a software interrupt function. It inserts an `SVC` (formerly `SWI`) instruction and the specified software interrupt number to make a proper function call. A function declared `__swi` accepts arguments and returns values. The `__swi` keyword makes the compiler generate the correct return sequence for a specific software interrupt function. Software interrupt functions follow the same calling convention regarding parameters and return values as an ordinary function, except for the stack usage.

The `__swi` keyword also expects a software interrupt number which is specified with the `#pragma swi_number=`*`number`* directive. The `swi_number` is used as an argument to the generated assembler `SWC` instruction, and can be used by the SVC interrupt handler, for example `SWI_Handler`, to select one software interrupt function in a system containing several such functions. Note that the software interrupt number should only be specified in the function declaration—typically, in a header file that you include in the source code file that calls the interrupt function—not in the function definition.

Note: All interrupt functions must be compiled in ARM mode, except for Cortex-M. Use either the `__arm` keyword or the `#pragma type_attribute=__arm` directive to alter the default behavior if needed.

Example

To declare your software interrupt function, typically in a header file, write for example like this:

```
#pragma swi_number=0x23
__swi int swi0x23_function(int a, int b);
...
```

To call the function:

```
...
int x = swi0x23_function(1, 2);  /* Will be replaced by SVC 0x23,
                                    hence the linker will
                                    never try to locate
                                        swi0x23_function */
...
```

Somewhere in your application source code, you define your software interrupt function:

```
...
__swi __arm int the_actual_swi0x23_function(int a, int b)
{
  ...
  return 42;
}
```

See also

Software interrupts, page 34 and *Calling convention*, page 97.

__task

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 233.

Description

This keyword allows functions to relax the rules for preserving registers. Typically, the keyword is used on the start function for a task in an RTOS.

By default, functions save the contents of used preserved registers on the stack upon entry, and restore them at exit. Functions that are declared `__task` do not save all registers, and therefore require less stack space.

Because a function declared `__task` can corrupt registers that are needed by the calling function, you should only use `__task` on functions that do not return or call such a function from assembler code.

The function `main` can be declared `__task`, unless it is explicitly called from the application. In real-time applications with more than one task, the root function of each task can be declared `__task`.

Example

```
__task void my_handler(void);
```

__thumb

Syntax

Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 233.

Description

The `__thumb` 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`.

Note: Non-interwork Thumb functions cannot be called from ARM mode.

Example

```
__thumb int func2(void);
```

See also

__interwork, page 238.

__weak

Syntax

Follows the generic syntax rules for object attributes, see *Object attributes*, page 235.

Description

Using the `__weak` object attribute on an external declaration of a symbol makes all references to that symbol in the module weak.

Using the `__weak` object attribute on a public definition of a symbol makes that definition a weak definition.

The linker will not include a module from a library solely to satisfy weak references to a symbol, nor will the lack of a definition for a weak reference result in an error. If no definition is included, the address of the object will be zero.

When linking, a symbol can have any number of weak definitions, and at most one non-weak definition. If the symbol is needed, and there is a non-weak definition, this definition will be used. If there is no non-weak definition, one of the weak definitions will be used.

Example

```
extern __weak int foo; /* A weak reference */

__weak void bar(void); /* A weak definition */
{
  /* Increment foo if it was included */
  if (&foo != 0)
    ++foo;
```

```
}
```

Pragma directives

This chapter describes the pragma directives of the compiler.

The #pragma directive is defined by the ISO/ANSI C standard and is a mechanism for using vendor-specific extensions in a controlled way to make sure that the source code is still portable.

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

The pragma directives are always enabled in the compiler.

Summary of pragma directives

This table lists the pragma directives of the compiler that can be used either with the `#pragma` preprocessor directive or the `_Pragma()` preprocessor operator:

Pragma directive	Description
`bitfields`	Controls the order of bitfield members
`data_alignment`	Gives a variable a higher (more strict) alignment
`diag_default`	Changes the severity level of diagnostic messages
`diag_error`	Changes the severity level of diagnostic messages
`diag_remark`	Changes the severity level of diagnostic messages
`diag_suppress`	Suppresses diagnostic messages
`diag_warning`	Changes the severity level of diagnostic messages
`include_alias`	Specifies an alias for an include file
`inline`	Inlines a function
`language`	Controls the IAR Systems language extensions
`location`	Specifies the absolute address of a variable, or places groups of functions or variables in named sections
`message`	Prints a message
`object_attribute`	Changes the definition of a variable or a function
`optimize`	Specifies the type and level of an optimization

Table 29: Pragma directives summary

Pragma directive	Description
pack	Specifies the alignment of structures and union members
__printf_args	Verifies that a function with a printf-style format string is called with the correct arguments
required	Ensures that a symbol that is needed by another symbol is included in the linked output
rtmodel	Adds a runtime model attribute to the module
__scanf_args	Verifies that a function with a scanf-style format string is called with the correct arguments
section	Declares a section name to be used by intrinsic functions
swi_number	Sets the interrupt number of a software interrupt function
type_attribute	Changes the declaration and definitions of a variable or function
weak	Makes a definition a weak definition, or creates a weak alias for a function or a variable

Table 29: 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. See also *Recognized pragma directives (6.8.6)*, page 355.

Descriptions of pragma directives

This section gives detailed information about each pragma directive.

bitfields

Syntax

```
#pragma bitfields=disjoint_types|joined_types|
                  reversed_disjoint_types|reversed|default}
```

Parameters

disjoint_types	Bitfield members are placed from the least significant bit to the most significant bit in the container type. Storage containers of bitfields with different base types will not overlap.

`joined_types`	Bitfield members are placed depending on the byte order. Storage containers of bitfields will overlap other structure members. For more details, see *Bitfields*, page 212.
`reversed_disjoint_types`	Bitfield members are placed from the most significant bit to the least significant bit in the container type. Storage containers of bitfields with different base types will not overlap.
`reversed`	This is an alias for `reversed_disjoint_types`.
`default`	Restores to default layout of bitfield members. The default behavior for the compiler is `joined_types`.

Description

Use this pragma directive to control the layout of bitfield members.

Example

```
#pragma bitfields=disjoint_types
/* Structure that uses disjoint types. */
{
  unsigned char error :1;
  unsigned char size  :4;
  unsigned short code  :10;
}
#pragma bitfields=default /* Restores to default setting. */
```

See also

Bitfields, page 212.

data_alignment

Syntax

`#pragma data_alignment=`*`expression`*

Parameters

`expression`	A constant which must be a power of two (1, 2, 4, etc.).

Description

Use this pragma directive to give a variable a higher (more strict) alignment of the start address than it would otherwise have. This directive can be used on variables with static and automatic storage duration.

When you use this directive on variables with automatic storage duration, there is an upper limit on the allowed alignment for each function, determined by the calling convention used.

Note: Normally, the size of a variable is a multiple of its alignment. The `data_alignment` directive only affects the alignment of the variable's start address, and not its size, and can thus be used for creating situations where the size is not a multiple of the alignment.

diag_default

Syntax

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

Parameters

tag	The number of a diagnostic message, for example the message number `Pe117`.

Description

Use this pragma directive to change the severity level back to the default, or to the severity level defined on the command line by any of the options `--diag_error`, `--diag_remark`, `--diag_suppress`, or `--diag_warnings`, for the diagnostic messages specified with the tags.

See also

Diagnostics, page 152.

diag_error

Syntax

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

Parameters

tag	The number of a diagnostic message, for example the message number `Pe117`.

Description

Use this pragma directive to change the severity level to `error` for the specified diagnostics.

See also

Diagnostics, page 152.

diag_remark

Syntax

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

Parameters

tag	The number of a diagnostic message, for example the message number `Pe177`.

Description
Use this pragma directive to change the severity level to `remark` for the specified diagnostic messages.

See also
Diagnostics, page 152.

diag_suppress

Syntax
`#pragma diag_suppress=tag[,tag,...]`

Parameters

tag	The number of a diagnostic message, for example the message number `Pe117`.

Description
Use this pragma directive to suppress the specified diagnostic messages.

See also
Diagnostics, page 152.

diag_warning

Syntax
`#pragma diag_warning=tag[,tag,...]`

Parameters

tag	The number of a diagnostic message, for example the message number `Pe826`.

Description
Use this pragma directive to change the severity level to `warning` for the specified diagnostic messages.

See also
Diagnostics, page 152.

include_alias

Syntax
```
#pragma include_alias ("orig_header" , "subst_header")
#pragma include_alias (<orig_header> , <subst_header>)
```

Parameters

orig_header	The name of a header file for which you want to create an alias.
subst_header	The alias for the original header file.

Description

Use this pragma directive to provide an alias for a header file. This is useful for substituting one header file with another, and for specifying an absolute path to a relative file.

This pragma directive must appear before the corresponding `#include` directives and `subst_header` must match its corresponding `#include` directive exactly.

Example

```
#pragma include_alias (<stdio.h> , <C:\MyHeaders\stdio.h>)
#include <stdio.h>
```

This example will substitute the relative file `stdio.h` with a counterpart located according to the specified path.

See also

Include file search procedure, page 149.

inline

Syntax

```
#pragma inline[=forced]
```

Parameters

`forced`	Disables the compiler's heuristics and forces inlining.

Description

Use this pragma directive to advise the compiler that the function whose declaration follows immediately after the directive should be inlined—that is, expanded into the body of the calling function. Whether the inlining actually occurs is subject to the compiler's heuristics.

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

Specifying `#pragma inline=forced` disables the compiler's heuristics and forces inlining. If the inlining fails for some reason, for example if it cannot be used with the function type in question (like `printf`), an error message is emitted.

Note: Because specifying `#pragma inline=forced` disables the compiler's heuristics, including the inlining heuristics, the function declared immediately after the directive will not be inlined on optimization levels None or Low. No error or warning message will be emitted.

language

Syntax `#pragma language={extended|default}`

Parameters

`extended`	Turns on the IAR Systems language extensions and turns off the `--strict_ansi` command line option.
`default`	Uses the language settings specified by compiler options.

Description Use this pragma directive to enable the compiler language extensions or for using the language settings specified on the command line.

location

Syntax `#pragma location={address|NAME}`

Parameters

address	The absolute address of the global or static variable for which you want an absolute location.
NAME	A user-defined section name; cannot be a section name predefined for use by the compiler and linker.

Description Use this pragma directive to specify the location—the absolute address—of the global or static variable whose declaration follows the pragma directive. The variable must be declared either `__no_init` or `const`. Alternatively, the directive can take a string specifying a section for placing either a variable or a function whose declaration follows the pragma directive.

Example

```
#pragma location=0xFFFF0400
__no_init volatile char PORT1; /* PORT1 is located at address
                                  0xFFFF0400 */

#pragma location="foo"
char PORT1; /* PORT1 is located in section foo */

/* A better way is to use a corresponding mechanism */
#define FLASH _Pragma("location=\"FLASH\"")
...
FLASH int i; /* i is placed in the FLASH section */
```

See also *Controlling data and function placement in memory*, page 131.

message

Syntax

```
#pragma message(message)
```

Parameters

message	The message that you want to direct to the standard output stream.

Description

Use this pragma directive to make the compiler print a message to the standard output stream when the file is compiled.

Example:

```
#ifdef TESTING
#pragma message("Testing")
#endif
```

object_attribute

Syntax

```
#pragma object_attribute=object_attribute[,object_attribute,...]
```

Parameters

For a list of object attributes that can be used with this pragma directive, see *Object attributes*, page 235.

Description

Use this pragma directive to declare a variable or a function with an object attribute. This directive affects the definition of the identifier that follows immediately after the directive. The object is modified, not its type. Unlike the directive `#pragma type_attribute` that specifies the storing and accessing of a variable or function, it is not necessary to specify an object attribute in declarations.

Example

```
#pragma object_attribute=__no_init
char bar;
```

See also

General syntax rules for extended keywords, page 233.

optimize

Syntax

```
#pragma optimize=param[ param...]
```

Parameters

`balanced|size|speed`	Optimizes balanced between speed and size, optimizes for size, or optimizes for speed
`none|low|medium|high`	Specifies the level of optimization
`no_code_motion`	Turns off code motion

`no_cse`	Turns off common subexpression elimination
`no_inline`	Turns off function inlining
`no_tbaa`	Turns off type-based alias analysis
`no_unroll`	Turns off loop unrolling
`no_scheduling`	Turns off instruction scheduling

Description

Use this pragma directive to decrease the optimization level, or to turn off some specific optimizations. This pragma directive only affects the function that follows immediately after the directive.

The parameters `speed`, `size`, and `balanced` only have effect on the `high` optimization level and only one of them can be used as it is not possible to optimize for speed and size at the same time. It is also not possible to use preprocessor macros embedded in this pragma directive. Any such macro will not be expanded by the preprocessor.

Note: If you use the `#pragma optimize` directive to specify an optimization level that is higher than the optimization level you specify using a compiler option, the pragma directive is ignored.

Example

```
#pragma optimize=speed
int small_and_used_often()
{
    ...
}

#pragma optimize=size no_inline
int big_and_seldom_used()
{
    ...
}
```

pack

Syntax

```
#pragma pack(n)
#pragma pack()
#pragma pack({push|pop}[,name] [,n])
```

Parameters

n	Sets an optional structure alignment; one of: `1`, `2`, `4`, `8`, or `16`
Empty list	Restores the structure alignment to default
`push`	Sets a temporary structure alignment

`pop`	Restores the structure alignment from a temporarily pushed alignment
`name`	An optional pushed or popped alignment label

Description

Use this pragma directive to specify the maximum alignment of `struct` and `union` members.

The `#pragma pack` directive affects declarations of structures following the pragma directive to the next `#pragma pack` or end of file.

Note: This can result in significantly larger and slower code when accessing members of the structure.

See also

Structure types, page 216 and *__packed*, page 240.

__printf_args

Syntax

```
#pragma __printf_args
```

Description

Use this pragma directive on a function with a printf-style format string. For any call to that function, the compiler verifies that the argument to each conversion specifier (for example `%d`) is syntactically correct.

Example

```
#pragma __printf_args
int printf(char const *,...);


/* Function call */
printf("%d",x);  /* Compiler checks that x is an integer */
```

required

Syntax

```
#pragma required=symbol
```

Parameters

`symbol`	Any statically linked function or variable.

Description

Use this pragma directive to ensure that a symbol which is needed by a second symbol is included in the linked output. The directive must be placed immediately before the second symbol.

Use the directive if the requirement for a symbol is not otherwise visible in the application, for example if a variable is only referenced indirectly through the section it resides in.

Example

```
const char copyright[] = "Copyright by me";

#pragma required=copyright
int main()
{
  /* Do something here. */
}
```

Even if the copyright string is not used by the application, it will still be included by the linker and available in the output.

rtmodel

Syntax

```
#pragma rtmodel="key","value"
```

Parameters

`"key"`	A text string that specifies the runtime model attribute.
`"value"`	A text string that specifies the value of the runtime model attribute. Using the special value * is equivalent to not defining the attribute at all.

Description

Use this pragma directive to add a runtime model attribute to a module, which can be used by the linker to check consistency between modules.

This pragma directive is useful for enforcing consistency between modules. All modules that are linked together and define the same runtime attribute key must have the same value for the corresponding key, or the special value `*`. It can, however, be useful to state explicitly that the module can handle any runtime model.

A module can have several runtime model definitions.

Note: The predefined compiler runtime model attributes start with a double underscore. To avoid confusion, this style must not be used in the user-defined attributes.

Example

```
#pragma rtmodel="I2C","ENABLED"
```

The linker will generate an error if a module that contains this definition is linked with a module that does not have the corresponding runtime model attributes defined.

See also

Checking module consistency, page 87.

__scanf_args

Syntax

```
#pragma __scanf_args
```

Description

Use this pragma directive on a function with a scanf-style format string. For any call to that function, the compiler verifies that the argument to each conversion specifier (for example `%d`) is syntactically correct.

Example

```
#pragma __scanf_args
int printf(char const *,...);


/* Function call */
scanf("%d",x);  /* Compiler checks that x is an integer */
```

section

Syntax

`#pragma section="`*`NAME`*`" [`*`align`*`]`

alias

`#pragma segment="`*`NAME`*`" [`*`align`*`]`

Parameters

NAME	The name of the section or segment
align	Specifies an alignment for the section. The value must be a constant integer expression to the power of two.

Description

Use this pragma directive to define a section name that can be used by the section operators `__section_begin`, `__section_end`, and `__section_size`. All section declarations for a specific section must have the same memory type attribute and alignment.

Example

```
#pragma section="MYSECTION" 4
```

See also

Dedicated section operators, page 223. For more information about sections, see the chapter *Linking your application*.

swi_number

Syntax
```
#pragma swi_number=number
```

Parameters

number	The software interrupt number

Description

Use this pragma directive together with the `__swi` extended keyword. It is used as an argument to the generated `SWC` assembler instruction, and is used for selecting one software interrupt function in a system containing several such functions.

Example
```
#pragma swi_number=17
```

See also

Software interrupts, page 34.

type_attribute

Syntax
```
#pragma type_attribute=type_attribute[,type_attribute,...]
```

Parameters

For a list of type attributes that can be used with this pragma directive, see *Type attributes*, page 233.

Description

Use this pragma directive to specify IAR-specific *type attributes*, which are not part of the ISO/ANSI C language standard. Note however, that a given type attribute might not be applicable to all kind of objects.

This directive affects the declaration of the identifier, the next variable, or the next function that follows immediately after the pragma directive.

Example

In this example, thumb-mode code is generated for the function `foo`:

```
#pragma type_attribute=__thumb
void foo(void)
{
}
```

This declaration, which uses extended keywords, is equivalent:

```
__thumb void foo(void);
{
}
```

See also

See the chapter *Extended keywords* for more details.

weak

Syntax

```
#pragma weak symbol1={symbol2}
```

Parameters

symbol1	A function or variable with external linkage.
symbol2	A defined function or variable.

Description

This pragma directive can be used in one of two ways:

- To make the definition of a function or variable with external linkage a weak definition. The `__weak` attribute can also be used for this purpose.
- To create a weak alias for another function or variable. You can make more than one alias for the same function or variable.

Example

To make the definition of `foo` a weak definition, write:

```
#pragma weak foo
```

To make `NMI_Handler` a weak alias for `Default_Handler`, write:

```
#pragma weak NMI_Handler=Default_Handler
```

If `NMI_Handler` is not defined elsewhere in the program, all references to `NMI_Handler` will refer to `Default_Handler`.

See also

__weak, page 243.

Intrinsic functions

This chapter gives reference information about the intrinsic functions, a predefined set of functions available in the compiler.

The intrinsic functions provide direct access to low-level processor operations and can be very useful in, for example, time-critical routines. The intrinsic functions compile into inline code, either as a single instruction or as a short sequence of instructions.

Summary of intrinsic functions

To use intrinsic functions in an application, include the header file `intrinsics.h`.

Note that the intrinsic function names start with double underscores, for example:

`__disable_interrupt`

This table summarizes the intrinsic functions:

Intrinsic function	Description
`__CLZ`	Inserts a `CLZ` instruction
`__disable_fiq`	Disables fast interrupt requests (fiq)
`__disable_interrupt`	Disables interrupts
`__disable_irq`	Disables interrupt requests (irq)
`__DMB`	Inserts a `DMB` instruction
`__DSB`	Inserts a `DSB` instruction
`__enable_fiq`	Enables fast interrupt requests (fiq)
`__enable_interrupt`	Enables interrupts
`__enable_irq`	Enables interrupt requests (irq)
`__get_BASEPRI`	Returns the value of the Cortex-M3 `BASEPRI` register
`__get_CONTROL`	Returns the value of the Cortex-M `CONTROL` register
`__get_CPSR`	Returns the value of the ARM `CPSR` (Current Program Status Register)
`__get_FAULTMASK`	Returns the value of the Cortex-M3 `FAULTMASK` register
`__get_interrupt_state`	Returns the interrupt state

Table 30: Intrinsic functions summary

Intrinsic function	Description
__get_PRIMASK	Returns the value of the Cortex-M PRIMASK register
__ISB	Inserts a ISB instruction
__LDC	Inserts the coprocessor load instruction LDC
__LDCL	Inserts the coprocessor load instruction LDCL
__LDC2	Inserts the coprocessor load instruction LDC2
__LDC2L	Inserts the coprocessor load instruction LDC2L
__LDC_noidx	Inserts the coprocessor load instruction LDC
__LDCL_noidx	Inserts the coprocessor load instruction LDCL
__LDC2_noidx	Inserts the coprocessor load instruction LDC2
__LDC2L_noidx	Inserts the coprocessor load instruction LDC2L
__LDREX	Inserts an LDREX instruction
__MCR	Inserts the coprocessor write instruction MCR
__MRC	Inserts the coprocessor read instruction MRC
__no_operation	Inserts a NOP instruction
__QADD	Inserts a QADD instruction
__QADD8	Inserts a QADD8 instruction
__QADD16	Inserts a QADD16 instruction
__QASX	Inserts a QASX instruction
__QDADD	Inserts a QDADD instruction
__QDOUBLE	Inserts a QADD instruction
__QDSUB	Inserts a QDSUB instruction
__QFlag	Returns the Q flag that indicates if overflow/saturation has occurred
__QSUB	Inserts a QSUB instruction
__QSUB8	Inserts a QSUB8 instruction
__QSUB16	Inserts a QSUB16 instruction
__QSAX	Inserts a QSAX instruction
__REV	Inserts a REV instruction
__REVSH	Inserts a REVSH instruction
__SADD8	Inserts a SADD8 instruction
__SADD16	Inserts a SADD16 instruction
__SASX	Inserts a SASX instruction

Table 30: Intrinsic functions summary (Continued)

Intrinsic function	Description
__SEL	Inserts a SEL instruction
__set_BASEPRI	Sets the value of the Cortex-M3 BASEPRI register
__set_CONTROL	Sets the value of the Cortex-M CONTROL register
__set_CPSR	Sets the value of the ARM CPSR (Current Program Status Register)
__set_FAULTMASK	Sets the value of the Cortex-M3 FAULTMASK register
__set_interrupt_state	Restores the interrupt state
__set_PRIMASK	Sets the value of the Cortex-M PRIMASK register
__SHADD8	Inserts a SHADD8 instruction
__SHADD16	Inserts a SHADD16 instruction
__SHASX	Inserts a SHASX instruction
__SHSUB8	Inserts a SHSUB8 instruction
__SHSUB16	Inserts a SHSUB16 instruction
__SHSAX	Inserts a SHSAX instruction
__SMUL	Inserts a signed 16-bit multiplication
__SSUB8	Inserts a SSUB8 instruction
__SSUB16	Inserts a SSUB16 instruction
__SSAX	Inserts a SSAX instruction
__STC	Inserts the coprocessor store instruction STC
__STCL	Inserts the coprocessor store instruction STCL
__STC2	Inserts the coprocessor store instruction STC2
__STC2L	Inserts the coprocessor store instruction STC2L
__STC_noidx	Inserts the coprocessor store instruction STC
__STCL_noidx	Inserts the coprocessor store instruction STCL
__STC2_noidx	Inserts the coprocessor store instruction STC2
__STC2L_noidx	Inserts the coprocessor store instruction STC2L
__STREX	Inserts a STREX instruction
__SWP	Inserts an SWP instruction
__SWPB	Inserts an SWPB instruction
__UADD8	Inserts a UADD8 instruction
__UADD16	Inserts a UADD16 instruction
__UASX	Inserts a UASX instruction

Table 30: Intrinsic functions summary (Continued)

Intrinsic function	Description
`__UHADD8`	Inserts a `UHADD8` instruction
`__UHADD16`	Inserts a `UHADD16` instruction
`__UHASX`	Inserts a `UHASX` instruction
`__UHSAX`	Inserts a `UHSAX` instruction
`__UHSUB8`	Inserts a `UHSUB8` instruction
`__UHSUB16`	Inserts a `UHSUB16` instruction
`__UQADD8`	Inserts a `UQADD8` instruction
`__UQADD16`	Inserts a `UQADD16` instruction
`__UQASX`	Inserts a `UQASX` instruction
`__UQSUB8`	Inserts a `UQSUB8` instruction
`__UQSUB16`	Inserts a `UQSUB16` instruction
`__UQSAX`	Inserts a `UQSAX` instruction
`__USAX`	Inserts a `USAX` instruction
`__USUB8`	Inserts a `USUB8` instruction
`__USUB16`	Inserts a `USUB16` instruction

Table 30: Intrinsic functions summary (Continued)

Descriptions of intrinsic functions

This section gives reference information about each intrinsic function.

__CLZ

Syntax

```
unsigned char __CLZ(unsigned long);
```

Description

Inserts a `CLZ` instruction.

This intrinsic function requires an ARM v5 architecture or higher for ARM mode, and ARM v6T2 or higher for Thumb mode.

__disable_fiq

Syntax

```
void __disable_fiq(void);
```

Description

Disables fast interrupt requests (fiq).

This intrinsic function can only be used in privileged mode and is not available for Cortex-M devices.

__disable_interrupt

Syntax

```
void __disable_interrupt(void);
```

Description

Disables interrupts. For Cortex-M devices, it raises the execution priority level to 0 by setting the priority mask bit, `PRIMASK`. For other devices, it disables interrupt requests (irq) and fast interrupt requests (fiq).

This intrinsic function can only be used in privileged mode.

__disable_irq

Syntax

```
void __disable_irq(void);
```

Description

Disables interrupt requests (irq).

This intrinsic function can only be used in privileged mode and is not available for Cortex-M devices.

__DMB

Syntax

```
void __DMB(void);
```

Description

Inserts a `DMB` instruction. This intrinsic function requires an ARM v7 architecture or higher.

__DSB

Syntax

```
void __DSB(void);
```

Description

Inserts a `DSB` instruction. This intrinsic function requires an ARM v7 architecture or higher.

__enable_fiq

Syntax

```
void __enable_fiq(void);
```

Description

Enables fast interrupt requests (fiq).

This intrinsic function can only be used in privileged mode, and it is not available for Cortex-M devices.

__enable_interrupt

Syntax

```
void __enable_interrupt(void);
```

Description

Enables interrupts. For Cortex-M devices, it resets the execution priority level to default by clearing the priority mask bit, `PRIMASK`. For other devices, it enables interrupt requests (irq) and fast interrupt requests (fiq).

This intrinsic function can only be used in privileged mode.

__enable_irq

Syntax

```
void __enable_irq(void);
```

Description

Enables interrupt requests (irq).

This intrinsic function can only be used in privileged mode, and it is not available for Cortex-M devices.

__get_BASEPRI

Syntax

```
unsigned long __get_BASEPRI(void);
```

Description

Returns the value of the `BASEPRI` register. This intrinsic function can only be used in privileged mode and it requires a Cortex-M3 device.

__get_CONTROL

Syntax

```
unsigned long __get_CONTROL(void);
```

Description

Returns the value of the `CONTROL` register. This intrinsic function can only be used in privileged mode and it requires a Cortex-M device.

__get_CPSR

Syntax

```
unsigned long __get_CPSR(void);
```

Description

Returns the value of the ARM CPSR (Current Program Status Register). This intrinsic function can only be used in privileged mode, is not available for Cortex-M devices, and it requires ARM mode.

__get_FAULTMASK

Syntax

```
unsigned long __get_FAULTMASK(void);
```

Description

Returns the value of the FAULTMASK register. This intrinsic function can only be used in privileged mode and it requires a Cortex-M3 device.

__get_interrupt_state

Syntax

```
__istate_t __get_interrupt_state(void);
```

Description

Returns the global interrupt state. The return value can be used as an argument to the `__set_interrupt_state` intrinsic function, which will restore the interrupt state.

This intrinsic function can only be used in privileged mode, and cannot be used when using the `--aeabi` compiler option.

Example

```
__istate_t s = __get_interrupt_state();
__disable_interrupt();

/* Do something here. */

__set_interrupt_state(s);
```

The advantage of using this sequence of code compared to using `__disable_interrupt` and `__enable_interrupt` is that the code in this example will not enable any interrupts disabled before the call of `__get_interrupt_state`.

__get_PRIMASK

Syntax

```
unsigned long __get_PRIMASK(void);
```

Description

Returns the value of the PRIMASK register. This intrinsic function can only be used in privileged mode and it requires a Cortex-M device.

__ISB

Syntax

```
void __ISB(void);
```

Description

Inserts a `ISB` instruction. This intrinsic function requires an ARM v7 architecture or higher.

__LDC
__LDCL
__LDC2
__LDC2L

Syntax

```
void __nnn(__ul coproc, __ul CRn, __ul const *src);
```

where *nnn* can be one of `LDC`, `LDCL`, `LDC2`, or `LDC2L`.

Parameters

coproc	The coprocessor number `0..15`.
CRn	The coprocessor register to load.
src	A pointer to the data to load.

Description

Inserts the coprocessor load instruction `LDC`—or one of its variants—which means that a value will be loaded into a coprocessor register. The parameters *coproc* and *CRn* will be encoded in the instruction and must therefore be constants.

__LDC_noidx
__LDCL_noidx
__LDC2_noidx
__LDC2L_noidx

Syntax

```
void __nnn_noidx(__ul coproc, __ul CRn, __ul const *src, __ul
option);
```

where *nnn* can be one of `LDC`, `LDCL`, `LDC2`, or `LDC2L`.

Parameters

coproc	The coprocessor number `0..15`.
CRn	The coprocessor register to load.
src	A pointer to the data to load.

option	Additional coprocessor option `0..255`.

Description

Inserts the coprocessor load instruction `LDC`, or one of its variants. A value will be loaded into a coprocessor register. The parameters *coproc*, *CRn*, and *option* will be encoded in the instruction and must therefore be constants.

__LDREX

Syntax

```
unsigned long __LDREX(unsigned long *);
```

Description

Inserts an `LDREX` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v6T2 or higher for Thumb mode.

__MCR

Syntax

```
void __MCR(__ul coproc, __ul opcode_1, __ul src, __ul CRn, __ul
CRm, __ul opcode_2);
```

Parameters

`coproc`	The coprocessor number `0..15`.
`opcode_1`	Coprocessor-specific operation code.
`src`	The value to be written to the coprocessor.
`CRn`	The coprocessor register to write to.
`CRm`	Additional coprocessor register; set to zero if not used.
`opcode_2`	Additional coprocessor-specific operation code; set to zero if not used.

Description

Inserts 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.

This intrinsic function requires either ARM mode, or an ARM v6T2 or higher for Thumb mode.

__MRC

Syntax

```
unsigned long __MRC(__ul coproc, __ul opcode_1, __ul CRn, __ul
CRm, __ul opcode_2);
```

Parameters

coproc	The coprocessor number 0..15.
opcode_1	Coprocessor-specific operation code.
CRn	The coprocessor register to write to.
CRm	Additional coprocessor register; set to zero if not used.
opcode_2	Additional coprocessor-specific operation code; set to zero if not used.

Description

Inserts 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.

This intrinsic function requires either ARM mode, or an ARM v6T2 or higher for Thumb mode.

__no_operation

Syntax

```
void __no_operation(void);
```

Description

Inserts a NOP instruction.

__QADD

Syntax

```
signed long __QADD(signed long, signed long);
```

Description

Inserts a QADD instruction.

This intrinsic function requires an ARM v5E architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__QADD8

Syntax

```
unsigned long __QADD8(unsigned long, unsigned long);
```

Description

Inserts a QADD8 instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__QADD16

Syntax

```
unsigned long __QADD16(unsigned long, unsigned long);
```

Description

Inserts a `QADD16` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__QASX

Syntax

```
unsigned long __QASX(unsigned long, unsigned long);
```

Description

Inserts a `QASX` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__QDADD

Syntax

```
signed long __QDADD(signed long, signed long);
```

Description

Inserts a `QDADD` instruction.

This intrinsic function requires an ARM v5E architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__QDOUBLE

Syntax

```
signed long __QDOUBLE(signed long);
```

Description

Inserts an instruction `QADD Rd,Rs,Rs` for a source register `Rs`, and a destination register `Rd`.

This intrinsic function requires an ARM v5E architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__QDSUB

Syntax `signed long __QDSUB(signed long, signed long);`

Description Inserts a `QDSUB` instruction.

This intrinsic function requires an ARM v5E architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__QFlag

Syntax `int __QFlag(void);`

Description Returns the `Q` flag that indicates if overflow/saturation has occurred.

This intrinsic function requires an ARM v5E architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__QSUB

Syntax `signed long __QSUB(signed long, signed long);`

Description Inserts a `QSUB` instruction.

This intrinsic function requires an ARM v5E architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__QSUB8

Syntax `unsigned long __QSUB8(unsigned long, unsigned long);`

Description Inserts a `QSUB8` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__QSUB16

Syntax `unsigned long __QSUB16(unsigned long, unsigned long);`

Description Inserts a `QSUB16` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__QSAX

Syntax

```
unsigned long __QSAX(unsigned long, unsigned long);
```

Description

Inserts a `QSAX` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__REV

Syntax

```
unsigned long __REV(unsigned long);
```

Description

Inserts a `REV` instruction. This intrinsic function requires an ARM v6 architecture or higher.

__REVSH

Syntax

```
signed long __REVSH(short);
```

Description

Inserts a `REVSH` instruction. This intrinsic function requires an ARM v6 architecture or higher.

__SADD8

Syntax

```
unsigned long __SADD8(unsigned long, unsigned long);
```

Description

Inserts a `SADD8` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__SADD16

Syntax

```
unsigned long __SADD16(unsigned long, unsigned long);
```

Description

Inserts a `SADD16` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__SASX

Syntax

```
unsigned long __SASX(unsigned long, unsigned long);
```

Description

Inserts a `SASX` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__SEL

Syntax

```
unsigned long __SEL(unsigned long, unsigned long);
```

Description

Inserts a `SEL` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__set_BASEPRI

Syntax

```
void __set_BASEPRI(unsigned long);
```

Description

Sets the value of the `BASEPRI` register. This intrinsic function can only be used in privileged mode and it requires a Cortex-M3 device.

__set_CONTROL

Syntax

```
void __set_CONTROL(unsigned long);
```

Description

Sets the value of the `CONTROL` register. This intrinsic function can only be used in privileged mode and it requires a Cortex-M device.

__set_CPSR

Syntax
```
void __set_CPSR(unsigned long);
```

Description

Sets the value of the ARM `CPSR` (Current Program Status Register). Only the control field is changed (bits 0-7). This intrinsic function can only be used in privileged mode, is not available for Cortex-M devices, and it requires ARM mode.

__set_FAULTMASK

Syntax
```
void __set_FAULTMASK(unsigned long);
```

Description

Sets the value of the `FAULTMASK` register. This intrinsic function can only be used in privileged mode and it requires a Cortex-M3 device.

__set_interrupt_state

Syntax
```
void __set_interrupt_state(__istate_t);
```

Descriptions

Restores the interrupt state to a value previously returned by the `__get_interrupt_state` function.

For information about the `__istate_t` type, see *__get_interrupt_state*, page 265.

__set_PRIMASK

Syntax
```
void __set_PRIMASK(unsigned long);
```

Description

Sets the value of the `PRIMASK` register. This intrinsic function can only be used in privileged mode and it requires a Cortex-M device.

__SHADD8

Syntax
```
unsigned long __SHADD8(unsigned long, unsigned long);
```

Description

Inserts a `SHADD8` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__SHADD16

Syntax

```
unsigned long __SHADD16(unsigned long, unsigned long);
```

Description

Inserts a `SHADD16` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__SHASX

Syntax

```
unsigned long __SHASX(unsigned long, unsigned long);
```

Description

Inserts a `SHASX` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__SHSUB8

Syntax

```
unsigned long __SHSUB8(unsigned long, unsigned long);
```

Description

Inserts a `SHSUB8` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__SHSUB16

Syntax

```
unsigned long __SHSUB16(unsigned long, unsigned long);
```

Description

Inserts a `SHSUB16` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__SHSAX

Syntax

```
unsigned long __SHSAX(unsigned long, unsigned long);
```

Description

Inserts a `SHSAX` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__SMUL

Syntax

```
signed long __SMUL(signed short, signed short);
```

Description

Inserts a signed 16-bit multiplication.

This intrinsic function requires an ARM v5E architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__SSUB8

Syntax

```
unsigned long __SSUB8(unsigned long, unsigned long);
```

Description

Inserts a `SSUB8` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__SSUB16

Syntax

```
unsigned long __SSUB16(unsigned long, unsigned long);
```

Description

Inserts a `SSUB16` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__SSAX

Syntax

```
unsigned long __SSAX(unsigned long, unsigned long);
```

Description

Inserts a `SSAX` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__STC
__STCL
__STC2
__STC2L

Syntax

`void __nnn(__ul coproc, __ul CRn, __ul const *dst);`

where *nnn* can be one of `STC`, `STCL`, `STC2`, or `STC2L`.

Parameters

coproc	The coprocessor number `0..15`.
CRn	The coprocessor register to load.
dst	A pointer to the destination.

Description

Inserts the coprocessor store instruction `STC`—or one of its variants—which means that the value of the specified coprocessor register will be written to a memory location. The parameters *coproc* and *CRn* will be encoded in the instruction and must therefore be constants.

__STC_noidx
__STCL_noidx
__STC2_noidx
__STC2L_noidx

Syntax

`void __nnn_noidx(__ul coproc, __ul CRn, __ul const *dst, __ul option);`

where *nnn* can be one of `STC`, `STCL`, `STC2`, or `STC2L`.

Parameters

coproc	The coprocessor number `0..15`.
CRn	The coprocessor register to load.
dst	A pointer to the destination.
option	Additional coprocessor option `0..255`.

Description

Inserts the coprocessor store instruction `STC`—or one of its variants—which means that the value of the specified coprocessor register will be written to a memory location. The parameters *coproc*, *CRn*, and *option* will be encoded in the instruction and must therefore be constants.

__STREX

Syntax

```
unsigned long __STREX(unsigned long, unsigned long *);
```

Description

Inserts a `STREX` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v6T2 or higher for Thumb mode.

__SWP

Syntax

```
unsigned long __SWP(unsigned long, unsigned long *);
```

Description

Inserts an `SWP` instruction. This intrinsic function requires ARM mode.

__SWPB

Syntax

```
char __SWPB(unsigned char, unsigned char *);
```

Description

Inserts an `SWPB` instruction. This intrinsic function requires ARM mode.

__UADD8

Syntax

```
unsigned long __UADD8(unsigned long, unsigned long);
```

Description

Inserts a `UADD8` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__UADD16

Syntax

```
unsigned long __UADD16(unsigned long, unsigned long);
```

Description

Inserts a `UADD16` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__UASX

Syntax

```
unsigned long __UASX(unsigned long, unsigned long);
```

Description

Inserts a `UASX` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__UHADD8

Syntax

```
unsigned long __UHADD8(unsigned long, unsigned long);
```

Description

Inserts a `UHADD8` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__UHADD16

Syntax

```
unsigned long __UHADD16(unsigned long, unsigned long);
```

Description

Inserts a `UHADD16` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__UHASX

Syntax

```
unsigned long __UHASX(unsigned long, unsigned long);
```

Description

Inserts a `UHASX` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__UHSAX

Syntax

```
unsigned long __UHSAX(unsigned long, unsigned long);
```

Description

Inserts a `UHSAX` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__UHSUB8

Syntax

```
unsigned long __UHSUB8(unsigned long, unsigned long);
```

Description

Inserts a `UHSUB8` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__UHSUB16

Syntax

```
unsigned long __UHSUB16(unsigned long, unsigned long);
```

Description

Inserts a `UHSUB16` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__UQADD8

Syntax

```
unsigned long __UQADD8(unsigned long, unsigned long);
```

Description

Inserts a `UQADD8` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__UQADD16

Syntax

```
unsigned long __UQADD16(unsigned long, unsigned long);
```

Description

Inserts a `UQADD16` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__UQASX

Syntax
```
unsigned long __UQASX(unsigned long, unsigned long);
```

Description

Inserts a `UQASX` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__UQSUB8

Syntax
```
unsigned long __UQSUB8(unsigned long, unsigned long);
```

Description

Inserts a `UQSUB8` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__UQSUB16

Syntax
```
unsigned long __UQSUB16(unsigned long, unsigned long);
```

Description

Inserts a `UQSUB16` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__UQSAX

Syntax
```
unsigned long __UQSAX(unsigned long, unsigned long);
```

Description

Inserts a `UQSAX` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__USAX

Syntax
```
unsigned long __USAX(unsigned long, unsigned long);
```

Description

Inserts a `USAX` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__USUB8

Syntax

```
unsigned long __USUB8(unsigned long, unsigned long);
```

Description

Inserts a `USUB8` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

__USUB16

Syntax

```
unsigned long __USUB16(unsigned long, unsigned long);
```

Description

Inserts a `USUB16` instruction.

This intrinsic function requires an ARM v6 architecture or higher for ARM mode, and ARM v7 with profile A or R for Thumb mode.

The preprocessor

This chapter gives a brief overview of the preprocessor, including reference information about the different preprocessor directives, symbols, and other related information.

Overview of the preprocessor

The preprocessor of the IAR C/C++ Compiler for ARM adheres to the ISO/ANSI standard. The compiler also makes these preprocessor-related features available to you:

- Predefined preprocessor symbols

  These symbols allow you to inspect the compile-time environment, for example the time and date of compilation. For details, see *Descriptions of predefined preprocessor symbols*, page 284.
- User-defined preprocessor symbols defined using a compiler option

  In addition to defining your own preprocessor symbols using the `#define` directive, you can also use the option `-D`, see *-D*, page 164.
- Preprocessor extensions

  There are several preprocessor extensions, for example many pragma directives; for more information, see the chapter *Pragma directives* in this guide. Read also about the corresponding `_Pragma` operator and the other extensions related to the preprocessor, see *Descriptions of miscellaneous preprocessor extensions*, page 287.
- Preprocessor output

  Use the option `--preprocess` to direct preprocessor output to a named file, see *--preprocess*, page 184.

Some parts listed by the ISO/ANSI standard are implementation-defined, for example the character set used in the preprocessor directives and inclusion of bracketed and quoted filenames. To read more about this, see *Preprocessing directives*, page 354.

Descriptions of predefined preprocessor symbols

This table describes the predefined preprocessor symbols:

Predefined symbol	Identifies
`__BASE_FILE__`	A string that identifies the name of the base source file (that is, not the header file), being compiled. See also *__FILE__, page 284*, and *--no_path_in_file_macros*, page 178.
`__BUILD_NUMBER__`	A unique integer that identifies the build number of the compiler currently in use.
`__CORE__`	An integer that identifies the processor architecture in use. The symbol reflects the `--cpu` option and is defined to `__ARM4M__`, `__ARM4TM__`, `__ARM5__`, `__ARM5E__`, `__ARM6__`, `__ARM6M__`, `__ARM6SM__`, `__ARM7M__`, or `__ARM7R__`. These symbolic names can be used when testing the `__CORE__` symbol.
`__ARMVFP__`	An integer that reflects the `--fpu` option and is defined to `1` for VFPv1 and `2` for VFPv2. If VFP code generation is disabled (default), the symbol will be undefined.
`__cplusplus`	An integer which is defined when the compiler runs in any of the C++ modes, otherwise it is undefined. When defined, its value is `199711L`. This symbol can be used with `#ifdef` to detect whether the compiler accepts C++ code. It is particularly useful when creating header files that are to be shared by C and C++ code.*
`__CPU_MODE__`	An integer that reflects the selected CPU mode and is defined to `1` for Thumb and `2` for ARM.
`__DATE__`	A string that identifies the date of compilation, which is returned in the form "`Mmm dd yyyy`", for example "`Oct 30 2008`". *
`__embedded_cplusplus`	An integer which is defined to `1` when the compiler runs in any of the C++ modes, otherwise the symbol is undefined. This symbol can be used with `#ifdef` to detect whether the compiler accepts C++ code. It is particularly useful when creating header files that are to be shared by C and C++ code.*
`__FILE__`	A string that identifies the name of the file being compiled, which can be both the base source file and any included header file. See also *__BASE_FILE__*, page 284, and *--no_path_in_file_macros*, page 178.*

Table 31: Predefined symbols

Predefined symbol	Identifies
`__func__`	A string that identifies the name of the function in which the symbol is used. This is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see -e, page 169. See also *__PRETTY_FUNCTION__*, page 285.
`__FUNCTION__`	A string that identifies the name of the function in which the symbol is used. This is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see -e, page 169. See also *__PRETTY_FUNCTION__*, page 285.
`__IAR_SYSTEMS_ICC__`	An integer that identifies the IAR compiler platform. The current value is 7. 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__`	An integer that is set to `1` when the code is compiled with the IAR C/C++ Compiler for ARM, and otherwise to `0`.
`__LINE__`	An integer that identifies the current source line number of the file being compiled, which can be both the base source file and any included header file.*
`__LITTLE_ENDIAN__`	An integer that reflects the `--endian` option and is defined to `1` when the byte order is little-endian. The symbol is defined to `0` when the byte order is big-endian.
`__PRETTY_FUNCTION__`	A string that identifies the function name, including parameter types and return type, of the function in which the symbol is used, for example `"void func(char)"`. This symbol is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled, see -e, page 169. See also *__func__*, page 285.
`__STDC__`	An integer that is set to `1`, which means the compiler adheres to the ISO/ANSI C standard. This symbol can be tested with `#ifdef` to detect whether the compiler in use adheres to ISO/ANSI C.*
`__STDC_VERSION__`	An integer that identifies the version of ISO/ANSI C standard in use. The symbols expands to `199409L`. This symbol does not apply in EC++ mode.*
`__TIME__`	A string that identifies the time of compilation in the form `"hh:mm:ss"`.*

Table 31: Predefined symbols (Continued)

Predefined symbol	Identifies
`__VER__`	An integer that identifies the version number of the IAR compiler in use. For example, version 5.11.3 is returned as 5011003.

Table 31: Predefined symbols (Continued)

* **This symbol is required by the ISO/ANSI standard.**

__TID__

Description

Target identifier for the IAR C/C++ Compiler for ARM. Expands to the target identifier which contains the following parts:

- A one-bit intrinsic flag (`i`) which is reserved for use by IAR
- A target identifier (`t`) unique for each IAR compiler. For the ARM compiler, the target identifier is `79`
- A value (`c`) reserved for specifying different CPU core families. The value is derived from the setting of the `--cpu` option:

Value	CPU core family
0	Unspecified
1	ARM7TDMI
2	ARM9TDMI
3	ARM9E
4	ARM10E
5	ARM11
6	Cortex–M3
7	Cortex–M0 or Cortex–M1
8	Cortex–R4

Table 32: Values for specifying different CPU core families in __TID__

The `__TID__`value is constructed as:

```
((i << 15) | (t << 8) | (c << 4))
```

You can extract the values as follows:

```
i = (__TID__ >> 15) & 0x01;    /* intrinsic flag */
t = (__TID__ >> 8) & 0x7F;     /* target identifier */
c = (__TID__ >> 4) & 0x0F;     /* cpu core family */
```

To find the value of the target identifier for the current compiler, execute:

```
printf("%ld",(__TID__ >> 8) & 0x7F)
```

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.

Descriptions of miscellaneous preprocessor extensions

This section gives reference information about the preprocessor extensions that are available in addition to the predefined symbols, pragma directives, and ISO/ANSI directives.

NDEBUG

Description

This preprocessor symbol determines whether any assert macros you have written in your application shall be included or not in the built application.

If this symbol is not defined, all assert macros are evaluated. If the symbol is defined, all assert macros are excluded from the compilation. In other words, if the symbol is:

- **defined**, the assert code will *not* be included
- **not defined**, the assert code will be included

This means that if you write any assert code and build your application, you should define this symbol to exclude the assert code from the final application.

Note that the assert macro is defined in the `assert.h` standard include file.

See also

Assert, page 85.

In the IDE, the `NDEBUG` symbol is automatically defined if you build your application in the Release build configuration.

_Pragma()

Syntax

`_Pragma("`*`string`*`")`

where *`string`* follows the syntax of the corresponding pragma directive.

Description

This preprocessor operator is part of the C99 standard and can be used, for example, in defines and is equivalent to the `#pragma` directive.

Note: The `-e` option—enable language extensions—does not have to be specified.

Example

```
#if NO_OPTIMIZE
  #define NOOPT _Pragma("optimize=none")
#else
  #define NOOPT
#endif
```

See also

See the chapter *Pragma directives*.

#warning message

Syntax

```
#warning message
```

where *message* can be any string.

Description

Use this preprocessor directive to produce messages. Typically, this is useful for assertions and other trace utilities, similar to the way the ISO/ANSI standard `#error` directive is used.

__VA_ARGS__

Syntax

```
#define P(...)        __VA_ARGS__
#define P(x, y, ...)    x + y + __VA_ARGS__
```

`__VA_ARGS__` will contain all variadic arguments concatenated, including the separating commas.

Description

Variadic macros are the preprocessor macro equivalents of `printf` style functions. `__VA_ARGS__` is part of the C99 standard.

Example

```
#if DEBUG
  #define DEBUG_TRACE(S, ...) printf(S, __VA_ARGS__)
#else
  #define DEBUG_TRACE(S, ...)
#endif
/* Place your own code here */
DEBUG_TRACE("The value is:%d\n",value);
```

will result in:

```
printf("The value is:%d\n",value);
```

Library functions

This chapter gives an introduction to the C and C++ library functions. It also lists the header files used for accessing library definitions.

For detailed reference information about the library functions, see the online help system.

Introduction

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

For 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 several 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 project configuration*, page 19. 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 these functions and parts are not reentrant because they need static data:

- Heap functions—`malloc`, `free`, `realloc`, `calloc`, and the C++ operators `new` and `delete`
- Time functions—`asctime`, `localtime`, `gmtime`, `mktime`
- Multibyte functions—`mbrlen`, `mbrtowc`, `mbsrtowc`, `wcrtomb`, `wcsrtomb`, `wctomb`
- The miscellaneous functions `setlocale`, `rand`, `atexit`, `strerror`, `strtok`
- Functions that use files in some way. This includes `printf`, `scanf`, `getchar`, and `putchar`. The functions `sprintf` and `sscanf` are not included.

Some functions also 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 is 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 behaviour* in this guide.
- Standard C library definitions, for user programs.
- Embedded C++ library definitions, for user programs.
- `CSTARTUP`, the module containing the start-up code. It is described in the chapter *The DLIB runtime environment* in this guide.
- Runtime support libraries; for example low-level floating-point routines.

- Intrinsic functions, allowing low-level use of ARM features. See the chapter *Intrinsic functions* for more information.

In addition, the IAR DLIB Library includes some added C functionality, partly taken from the C99 standard, see *Added C functionality*, page 294.

C HEADER FILES

This section lists the header files specific to the DLIB library C definitions. Header files may additionally contain target-specific definitions; these are documented in the chapter *Compiler extensions*.

The following table lists the C header files:

Header file	Usage
`assert.h`	Enforcing assertions when functions execute
`ctype.h`	Classifying characters
`errno.h`	Testing error codes reported by library functions
`float.h`	Testing floating-point type properties
`inttypes.h`	Defining formatters for all types defined in `stdint.h`
`iso646.h`	Using Amendment 1—iso646.h standard header
`limits.h`	Testing integer type properties
`locale.h`	Adapting to different cultural conventions
`math.h`	Computing common mathematical functions
`setjmp.h`	Executing non-local goto statements
`signal.h`	Controlling various exceptional conditions
`stdarg.h`	Accessing a varying number of arguments
`stdbool.h`	Adds support for the `bool` data type in C.
`stddef.h`	Defining several useful types and macros
`stdint.h`	Providing integer characteristics
`stdio.h`	Performing input and output
`stdlib.h`	Performing a variety of operations
`string.h`	Manipulating several kinds of strings
`time.h`	Converting between various time and date formats
`wchar.h`	Support for wide characters
`wctype.h`	Classifying wide characters

Table 33: Traditional standard C header files—DLIB

C++ HEADER FILES

This section lists the C++ header files.

Embedded C++

The following table lists the Embedded C++ header files:

Header file	Usage
complex	Defining a class that supports complex arithmetic
exception	Defining several functions that control exception handling
fstream	Defining several I/O stream classes that manipulate external files
iomanip	Declaring several I/O stream manipulators that take an argument
ios	Defining the class that serves as the base for many I/O streams classes
iosfwd	Declaring several I/O stream classes before they are necessarily defined
iostream	Declaring the I/O stream objects that manipulate the standard streams
istream	Defining the class that performs extractions
new	Declaring several functions that allocate and free storage
ostream	Defining the class that performs insertions
sstream	Defining several I/O stream classes that manipulate string containers
stdexcept	Defining several classes useful for reporting exceptions
streambuf	Defining classes that buffer I/O stream operations
string	Defining a class that implements a string container
strstream	Defining several I/O stream classes that manipulate in-memory character sequences

Table 34: Embedded C++ header files

The following table lists additional C++ header files:

Header file	Usage
fstream.h	Defining several I/O stream classes that manipulate external files
iomanip.h	Declaring several I/O stream manipulators that take an argument
iostream.h	Declaring the I/O stream objects that manipulate the standard streams
new.h	Declaring several functions that allocate and free storage

Table 35: Additional Embedded C++ header files—DLIB

Extended Embedded C++ standard template library

The following table lists the Extended EC++ standard template library (STL) header files:

Header file	Description
`algorithm`	Defines several common operations on sequences
`deque`	A deque sequence container
`functional`	Defines several function objects
`hash_map`	A map associative container, based on a hash algorithm
`hash_set`	A set associative container, based on a hash algorithm
`iterator`	Defines common iterators, and operations on iterators
`list`	A doubly-linked list sequence container
`map`	A map associative container
`memory`	Defines facilities for managing memory
`numeric`	Performs generalized numeric operations on sequences
`queue`	A queue sequence container
`set`	A set associative container
`slist`	A singly-linked list sequence container
`stack`	A stack sequence container
`utility`	Defines several utility components
`vector`	A vector sequence container

Table 36: Standard template library header files

Using standard C libraries in C++

The C++ library works in conjunction with 15 of the header files from the standard C library, sometimes with small alterations. The header files come in two forms—new and traditional—for example, `cassert` and `assert.h`.

The following table shows the new header files:

Header file	Usage
`cassert`	Enforcing assertions when functions execute
`cctype`	Classifying characters
`cerrno`	Testing error codes reported by library functions
`cfloat`	Testing floating-point type properties
`cinttypes`	Defining formatters for all types defined in `stdint.h`

Table 37: New standard C header files—DLIB

Header file	Usage
`climits`	Testing integer type properties
`clocale`	Adapting to different cultural conventions
`cmath`	Computing common mathematical functions
`csetjmp`	Executing non-local goto statements
`csignal`	Controlling various exceptional conditions
`cstdarg`	Accessing a varying number of arguments
`cstdbool`	Adds support for the `bool` data type in C.
`cstddef`	Defining several useful types and macros
`cstdint`	Providing integer characteristics
`cstdio`	Performing input and output
`cstdlib`	Performing a variety of operations
`cstring`	Manipulating several kinds of strings
`ctime`	Converting between various time and date formats
`cwchar`	Support for wide characters
`cwctype`	Classifying wide characters

Table 37: New standard C header files—DLIB (Continued)

LIBRARY FUNCTIONS AS INTRINSIC FUNCTIONS

Certain C library functions will under some circumstances be handled as intrinsic functions and will generate inline code instead of an ordinary function call, for example `memcpy`, `memset`, and `strcat`.

ADDED C FUNCTIONALITY

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

The following include files provide these features:

- `ctype.h`
- `inttypes.h`
- `math.h`
- `stdbool.h`
- `stdint.h`
- `stdio.h`
- `stdlib.h`
- `wchar.h`

- wctype.h

ctype.h

In ctype.h, the C99 function isblank is defined.

inttypes.h

This include file defines the formatters for all types defined in stdint.h to be used by the functions printf, scanf, and all their variants.

math.h

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

The following C99 macro symbols are defined:

HUGE_VALF, HUGE_VALL, INFINITY, NAN, FP_INFINITE, FP_NAN, FP_NORMAL, FP_SUBNORMAL, FP_ZERO, MATH_ERRNO, MATH_ERREXCEPT, math_errhandling.

The following C99 macro functions are defined:

fpclassify, signbit, isfinite, isinf, isnan, isnormal, isgreater, isless, islessequal, islessgreater, isunordered.

The following C99 type definitions are added:

float_t, double_t.

stdbool.h

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

stdint.h

This include file provides integer characteristics.

stdio.h

In stdio.h, the following C99 functions are defined:

vscanf, vfscanf, vsscanf, vsnprintf, snprintf

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

The following functions providing I/O functionality for libraries built without FILE support are defined:

__write_array	Corresponds to fwrite on stdout.
__ungetchar	Corresponds to ungetc on stdout.
__gets	Corresponds to fgets on stdin.

stdlib.h

In stdlib.h, the following C99 functions are defined:

_Exit, llabs, lldiv, strtoll, strtoull, atoll, strtof, strtold.

The function strtod has added functionality from the C99 standard. For reference information about this functions, see the library reference available from the **Help** menu.

The __qsortbbl function is defined; it provides sorting using a bubble sort algorithm. This is useful for applications that have a limited stack.

wchar.h

In wchar.h, the following C99 functions are defined:

vfwscanf, vswscanf, vwscanf, wcstof, wcstolb.

wctype.h

In wctype.h, the C99 function iswblank is defined.

The linker configuration file

This chapter describes the purpose of the linker configuration file and describes its contents.

To read this chapter you must be familiar with the concept of *sections*, see *Modules and sections*, page 40.

Overview

To link and locate an application in memory according to your requirements, ILINK needs information about how to handle sections and how to place them into the available memory regions. In other words, ILINK needs a *configuration*, passed to it by means of the *linker configuration file*.

This file consists of a sequence of directives and typically, provides facilities for:

- Defining available addressable memories

  giving the linker information about the maximum size of possible addresses and defining the available physical memory, as well as dealing with memories that can be addressed in different ways.
- Defining the regions of the available memories that are populated with ROM or RAM

  giving the start and end address for each region.
- Section groups

  dealing with how to group sections into blocks and overlays depending on the section requirements.
- Defining how to handle initialization of the application

  giving information about which sections that are to be initialized, and how that initialization should be made.
- Memory allocation

  defining where—in what memory region—each set of sections should be placed.
- Using symbols, expressions, and numbers

  expressing addresses and sizes, etc, in the other configuration directives. The symbols can also be used in the application itself.

- Structural configuration

  meaning that you can include or exclude directives depending on a condition, and to split the configuration file into several different files.

Comments can be written either as C comments (`/*...*/`) or as C++ comments (`//...`).

Defining memories and regions

ILINK needs information about the available memory spaces, or more specifically it needs information about:

- The maximum size of possible addressable memories

  The `define memory` directive defines a *memory space* with a given size, which is the maximum possible amount of addressable memory, not necessarily physically available. See *Define memory directive*, page 298.
- Available physical memory

  The `define region` directive defines a region in the available memories in which specific sections of application code and sections of application data can be placed. See *Define region directive*, page 299.

  A region consists of one or several memory ranges. A range is a continuous sequence of bytes in a memory and several ranges can be expressed by using region expressions. See *Region expression*, page 301.

Define memory directive

Syntax

```
define memory [ name ] with size = size_expr [ ,unit-size ];
```

where *`unit-size`* is one of:

```
unitbitsize = bitsize_expr
unitbytesize = bytesize_expr
```

and where *`expr`* is an expression, see *Expressions*, page 315.

Parameters

`size_expr`	Specifies how many *units* the memory space contains; always counted from address zero.
`bitsize_expr`	Specifies how many bits each unit contains.
`bytesize_expr`	Specifies how many bytes each unit contains. Each byte contains 8 bits.

Description

The `define memory` directive defines a *memory space* with a given size, which is the maximum possible amount of addressable memory, not necessarily physically available. This sets the limits for the possible addresses to be used in the linker configuration file. For many microcontrollers, one memory space is sufficient. However, some microcontrollers require two or more. For example, a Harvard architecture usually requires two different memory spaces, one for code and one for data. If only one memory is defined, the memory name is optional. If no *`unit-size`* is given, the unit contains 8 bits.

Example

```
/* Declare the memory space Mem of four Gigabytes */
define memory Mem with size = 4G;
```

Define region directive

Syntax

```
define region name = region-expr;
```

where *`region-expr`* is a region expression, see also *Regions*, page 299.

Parameters

name	The name of the region.

Description

The `define region` directive defines a region in which specific sections of code and sections of data can be placed. A region consists of one or several memory ranges, where each memory range consists of a continuous sequence of bytes in a specific memory. Several ranges can be combined by using region expressions. Note that those ranges do not need to be consecutive or even in the same memory.

Example

```
/* Define the 0x10000-byte code region ROM located at address
   0x10000 in memory Mem */
define region ROM = Mem:[from 0x10000 size 0x10000];
```

Regions

A *region* is s a set of non-overlapping memory ranges. A *region expression* is built up out of *region literals* and set operations (union, intersection, and difference) on regions.

Region literal

Syntax

```
[ memory-name: ][from expr { to expr | size expr }

  [ repeat expr [ displacement expr ]]]
```

where *expr* is an expression, see *Expressions*, page 315.

Parameters

memory-name	The name of the memory space in which the region literal will be located. If there is only one memory, the name is optional.
from	The start address of the memory range (inclusive).
to	The end address of the memory range (inclusive).
size	The size of the memory range.
repeat	Defines several ranges in the same memory for the region literal.
displacement	Displacement from the previous range start in the repeat sequence. Default displacement is the same value as the range size.

Description

A region literal consists of one memory range. When you define a range, the memory it resides in, a start address, and a size must be specified. The range size can be stated explicitly by specifying a size, or implicitly by specifying the final address of the range. The final address is included in the range and a zero-sized range will only contain an address. A range can span over the address zero and the range can even be expressed by unsigned values, because it is known where the memory wraps.

The `repeat` parameter will create a region literal that contains several ranges, one for each repeat. This is useful for *banked* or *far* regions.

Example

```
/* The 5-byte size range spans over the address zero */
Mem:[from -2 to 2]

/* The 512-byte size range spans over zero, in a 64-Kbyte memory
*/
Mem:[from 0xFF00 to 0xFF]

/* Defining several ranges in the same memory, a repeating
   literal */
Mem:[from 0 size 0x100 repeat 3 displacement 0x1000]

/* Resulting in a region containing:
   Mem:[from 0 size 0x100]
   Mem:[from 0x1000 size 0x100]
   Mem:[from 0x2000 size 0x100]
*/
```

See also

Define region directive, page 299, and *Region expression*, page 301.

Region expression

Syntax

```
region-operand
 | region-expr | region-operand
 | region-expr - region-operand
 | region-expr & region-operand
```

where `region-operand` is one of:

```
( region-expr )
region-name
region-literal
empty-region
```

where `region-name` is a region, see *Define region directive*, page 299

where `region-literal` is a region literal, see *Region literal*, page 299

and where `empty-region` is an empty region, see *Empty region*, page 302.

Description

Normally, a region consists of one memory range, which means a *region literal* is sufficient to express it. When a region contains several ranges, possibly in different memories, it is instead necessary to use a *region expression* to express it. Region expressions are actually set expressions on sets of memory ranges.

To create region expressions, three operators are available: union (`|`), intersection (`&`), and difference (`-`). These operators work as in *set theory*. For example, if you have the sets A and B, then the result of the operators would be:

- `A | B`: all elements in either set A or set B
- `A & B`: all elements in both set A and B
- `A - B`: all elements in set A but not in B.

Example

```
/* Resulting in a range starting at 1000 and ending at 2FFF, in
   memory Mem */
Mem:[from 0x1000 to 0x1FFF] | Mem:[from 0x1500 to 0x2FFF]

/* Resulting in a range starting at 1500 and ending at 1FFF, in
   memory Mem */
Mem:[from 0x1000 to 0x1FFF] & Mem:[from 0x1500 to 0x2FFF]

/* Resulting in a range starting at 1000 and ending at 14FF, in
   memory Mem */
Mem:[from 0x1000 to 0x1FFF] - Mem:[from 0x1500 to 0x2FFF]
```

```
/* Resulting in two ranges. The first starting at 1000 and ending
   at 1FFF, the second starting at 2501 and ending at 2FFF.
   Both located in memory Mem */
Mem:[from 0x1000 to 0x2FFF] - Mem:[from 0x2000 to 0x24FF]
```

Empty region

Syntax

```
[ ]
```

Description

The empty region does not contain any memory ranges. If the empty region is used in a placement directive that actually is used for placing one or more sections, ILINK will issue an error.

Example

```
define region Code = Mem:[from 0 size 0x10000];
if (Banked) {
  define region Bank = Mem:[from 0x8000 size 0x1000];
} else {
  define region Bank = [];
}
define region NonBanked = Code - Bank;

/* Depending on the Banked symbol, the NonBanked region is either
   one range with 0x10000 bytes, or two ranges with 0x8000 and
   0x7000 bytes, respectively. */
```

See also

Region expression, page 301.

Section handling

Section handling describes how ILINK should handle the sections of the execution image, which means:

- Placing sections in regions

  The `place at` and `place into` directives place sets of sections with similar attributes into previously defined regions. See *Place at directive*, page 309 and *Place in directive*, page 310.

- Making sets of sections with special requirements

  The `block` directive makes it possible to create empty sections with specific sizes and alignments, sequentially sorted sections of different types, etc.

  The `overlay` directive makes it possible to create an area of memory that can contain several overlay images. See *Define block directive*, page 303, and *Define overlay directive*, page 304.

- Initializing the application

  The directives `initialize` and `do not initialize` control how the application should be started. With these directives, the application can initialize global symbols at startup, and copy pieces of code. The initializers can be stored in several ways, for example they can be compressed. See *Initialize directive*, page 305 and *Do not initialize directive*, page 308.

- Keeping removed sections

  The `keep` directive retains sections even though they are not referred to by the rest of the application, which means it is equivalent to the *root* concept in the assembler and compiler. See *Keep directive*, page 309.

Define block directive

Syntax

```
define block name
  [ with param, param... ]
{
  extended-selectors
}
[except
 {
   section_selectors
  }];
```

where *param* can be one of:

```
size = expr
maximum size = expr
alignment = expr
fixed order
```

and where the rest of the directive selects sections to include in the block, see *Section selection*, page 311.

Parameters

name	The name of the defined block.
`size`	Customizes the size of the block. By default, the size of a block is the sum of its parts dependent of its contents.
`maximum size`	Specifies an upper limit for the size of the block. An error is generated if the sections in the block do not fit.
`alignment`	Specifies a minimum alignment for the block. If any section in the block has a higher alignment than the minimum alignment, the block will have that alignment.

`fixed order`	Places sections in fixed order; if not specified, the order of the sections will be arbitrary.

Description

The `block` directive defines a named set of sections. By defining a block you can create empty blocks of bytes that can be used, for example as stacks or heaps. Another use for the directive is to group certain types of sections, consecutive or non-consecutive. A third example of use for the directive is to group sections into one memory area to access the start and end of that area from the application.

Example

```
/* Create a 0x1000-byte block for the heap */
define block HEAP with size = 0x1000, alignment = 8 { };
```

See also

Interaction between the tools and your application, page 117. See *Define overlay directive*, page 304 for an *accessing* example.

Define overlay directive

Syntax

```
define overlay name [ with param, param... ]
{
  extended-selectors;
}
[except
 {
   section_selectors
  }];
```

For information about extended selectors and except clauses, see *Section selection*, page 311.

Parameters

`name`	The name of the overlay.
`size`	Customizes the size of the overlay. By default, the size of a overlay is the sum of its parts dependent of its contents.
`maximum size`	Specifies an upper limit for the size of the overlay. An error is generated if the sections in the overlay do not fit.
`alignment`	Specifies a minimum alignment for the overlay. If any section in the overlay has a higher alignment than the minimum alignment, the overlay will have that alignment.
`fixed order`	Places sections in fixed order; if not specified, the order of the sections will be arbitrary.

Description

The `overlay` directive defines a named set of sections. In contrast to the `block` directive, the overlay directive can define the same name several times. Each definition will then be grouped in memory at the same place as all other definitions of the same name. This creates an *overlaid* memory area, which can be useful for an application that has several independent sub-applications.

Place each sub-application image in ROM and reserve a RAM overlay area that can hold all sub-applications. To execute a sub-application, first copy it from ROM to the RAM overlay. Note that ILINK does not help you with managing interdependent overlay definitions, apart from generating a diagnostic message for any reference from one overlay to another overlay.

The size of an overlay will be the same size as the largest definition of that overlay name and the alignment requirements will be the same as for the definition with the highest alignment requirements.

Note: Sections that were overlaid must be split into a RAM and a ROM part and you must take care of all the copying needed.

See also

Manual initialization, page 54.

Initialize directive

Syntax

```
initialize { by copy | manually }
   [ with param, param... ]
{
  section-selectors
}
[except
 {
   section_selectors
  }];
```

where *param* is one of:

```
packing = { none | zeros | packbits | bwt | lzw | auto |
            smallest }
copy routine = functionname
```

and where the rest of the directive selects sections to include in the block. See *Section selection*, page 311.

Parameters

`by copy`	Splits the section into sections for initializers and initialized data, and handles the initialization at application startup automatically.

`manually`	Splits the section into sections for initializers and initialized data. The initialization at application startup is not handled automatically.
`packing`	Specifies how to handle the initializers. Choose between:

`none`	Disables compression of the selected section contents. This is the default method for `initialize manually`.
`zeros`	Compresses sequential bytes with the value zero.
`packbits`	Compresses with the PackBits algorithm. This method generates good results for data with many consecutive bytes of the same value.
`bwt`	Compresses with the Burrows-Wheeler algorithm. This method improves the `packbits` method by transforming blocks of data before they are compressed.
`lzw`	Compresses with the Lempel-Ziv-Welch algorithm. This method uses a dictionary to store byte patterns in the data.
`auto`	Similar to `smallest`, but ILINK chooses between `none` and `packbits`. This is the default method for `initialize by copy`.
`smallest`	ILINK estimates the resulting size using each packing method (except for `auto`), and then chooses the packing method that produces the smallest estimated size. Note that the size of the decompressor is also included.

`copy routine`	Uses your own initialization routine instead of the default routine. It will be automatically called at the application startup.

Description

The `initialize` directive splits the initialization section into one section holding the initializers and another section holding the initialized data. You can choose whether the initialization at startup should be handled automatically (`initialize by copy`) or whether you should handle it yourself (`initialize manually`).

When you use the packing method `auto` (default for `initialize by copy`) or `smallest`, ILINK will automatically choose an appropriate packing algorithm for the initializers and automatically revert it at the initialization process when the application starts. To override this, specify a different `packing` method. Use the `copy routine` parameter to override the method for copying the initializers. The `--log initialization` option shows how ILINK decided which packing algorithm to use.

When initializers are compressed, a decompressor is automatically added to the image. The decompressors for `bwt` and `lzw` use significantly more execution time and RAM than `zeros` and `packbits`. Approximately 9 Kbytes of stack space is needed for `bwt` and 3.5 Kbytes for `lzw`.

When initializers are compressed, the exact size of the compressed initializers is unknown until the exact content of the uncompressed data is known. If this data contains other addresses, and some of these addresses are dependent on the size of the compressed initializers, the linker fails with error Lp017. To avoid this, place compressed initializers last, or in a memory region together with sections whose addresses do not need to be known.

Optionally, ILINK will also produce a table that an initialization function in the system startup code uses for copying the section contents from the initializer sections to the corresponding original sections. Normally, the section content is initialized variables.

Zero-initialized sections are not affected by the `initialize` directive.

Sections that must execute before the initialization finishes are not affected by the `initialize by copy` directive. This includes the `__low_level_init` function and anything it references.

Anything reachable from the program entry label is considered *needed for initialization* unless reached via a section fragment with a label starting with `__iar_init$$done`. The `--log sections` option can be used for creating a log of this process (in addition to the more general process of marking section fragments to be included in the application).

The `initialize` directive can be used for copying other things as well, for example copying executable code from slow ROM to fast RAM. For another example, see *Define overlay directive*, page 304.

Example

```
/* Copy all read-write sections automatically from ROM to RAM at
   program start */
initialize by copy { rw };
place in RAM { rw };
place in ROM { ro };
```

```
/* Initialize special sections (initializers placed in flash) */
initialize by copy with packing = none, copy routine =
                   my_initializers { section .special };
place in RAM { section .special };
place in ROM { section .special_init };
```

See also

Initialization at system startup, page 45, and *Do not initialize directive*, page 308.

Do not initialize directive

Syntax

```
do not initialize
{
  section-selectors
}
[except
 {
   section-selectors
 }];
```

For information about extended selectors and except clauses, see *Section selection*, page 311.

Description

The `do not initialize` directive specifies the sections that should not be initialized by the system startup code. The directive can only be used on `zeroinit` sections.

The compiler keyword `__no_init` places variables into sections that must be handled by a `do not initialize` directive.

Example

```
/* Do not initialize read-write sections whose name ends with
   _noinit at program start */
do not initialize { rw section .*_noinit };
place in RAM { rw section .*_noinit };
```

See also

Initialization at system startup, page 45, and *Initialize directive*, page 305.

Keep directive

Syntax

```
keep
{
  section-selectors
}
[except
 {
   section-selectors
 }];
```

For information about extended selectors and except clauses, see *Section selection*, page 311.

Description

The `keep` directive specifies that all selected sections should be kept in the executable image, even if there are no references to the sections.

Example

```
keep { section .keep* } except {section .keep};
```

Place at directive

Syntax

```
[ "name": ]
place at { address [ memory: ] expr | start of region_expr |
           end of region_expr }
{
  extended-selectors
}
[except
 {
   section-selectors
 }];
```

For information about extended selectors and except clauses, see *Section selection*, page 311.

Parameters

memory: *expr*	A specific address in a specific memory. The address must be available in the supplied memory defined by the `define memory` directive. The memory specifier is optional if there is only one memory.
`start of` *region_expr*	A region expression. The start of the region is used.
`end of` *region_expr*	A region expression. The end of the region is used.

Description

The `place at` directive places sections and blocks either at a specific address or, at the beginning or the end of a region. The same address cannot be used for two different `place at` directives. It is also not possible to use an empty region in a `place at` directive. If placed in a region, the sections and blocks will be placed before any other sections or blocks placed in the same region with a `place in` directive.

The sections and blocks will be placed in the region in an arbitrary order. To specify a specific order, use the `block` directive.

The *name*, if specified, is used in the map file and in some log messages.

Example

```
/* Place the read-only section .startup at the beginning of the
   code_region */
"START": place at start of ROM { readonly section .startup };
```

See also

Place in directive, page 310.

Place in directive

Syntax

```
[ "name": ]
place in region-expr
{
  extended-selectors
}
 [{
   section-selectors
 }];
```

where *region-expr* is a region expression, see also *Regions*, page 299.

and where the rest of the directive selects sections to include in the block. See *Section selection*, page 311.

Description

The `place in` directive places sections and blocks in a specific region. The sections and blocks will be placed in the region in an arbitrary order.

To specify a specific order, use the `block` directive. The region can have several ranges.

The *name*, if specified, is used in the map file and in some log messages.

Example

```
/* Place the read-only sections in the code_region */
"ROM": place in ROM { readonly };
```

See also

Place at directive, page 309.

Section selection

The purpose of *section selection* is to specify—by means of *section selectors* and *except clauses*—the sections that an ILINK directive should be applied to. All sections that match one or more of the section selectors will be selected, and none of the sections selectors in the except clause, if any. Each section selector can match sections on section attributes, section name, and object or library name.

Some directives provide functionality that requires more detailed selection capabilities, for example directives that can be applied on both sections and blocks. In this case, the *extended-selectors* are used.

Section-selectors

Syntax

```
{ [ section-selector ][, section-selector...] }
```

where `section-selector` is:

```
[ section-attribute ][ section sectionname ]
    [object {module | filename} ]
```

where `section-attribute` is:

```
[ ro [ code | data ] | rw [ code | data ] | zi ]
```

and where `ro`, `rw`, and `zi` also can be `readonly`, `readwrite`, and `zeroinit`, respectively.

Parameters

`ro or readonly`	Read-only sections.
`rw or readwrite`	Read/write sections.
`zi or zeroinit`	Zero-initialized sections. These sections should be initialized with zeros during system startup.
`code`	Sections that contain code.
`data`	Sections that contain data.
`sectionname`	The section name. Two wildcards are allowed: `?` matches any single character `*` matches zero or more characters.
`module`	A name in the form *`objectname(libraryname)`*. Sections from object modules where both the object name and the library name match their respective patterns are selected. An empty library name pattern selects only sections from object files.

filename	The name of an object file, a library, or an object in a library. Two wildcards are allowed: `?` matches any single character `*` matches zero or more characters.

Description

A section selector selects all sections that match the section attribute, section name, and the name of the *object*, where *object* is an object file, a library, or an object in a library. It is only possible to omit one or two of the three conditions. If the section attribute is omitted, any section will be selected, without restrictions on the section attribute.

If the section name part or the object name part is omitted, sections will be selected without restrictions on the section name or object name, respectively.

It is also possible to use only `{ }` without any section selectors, which can be useful when defining blocks.

Note that a section selector with narrower scope has higher priority than a more generic section selector.

Example

```
{ rw }                         /* Selects all read-write sections */

{ section .mydata* }           /* Selects only .mydata* sections */
/* Selects .mydata* sections available in the object special.o */
{ section .mydata* object special.o }
```

Assuming a section in an object named `foo.o` in a library named `lib.a`, any of these selectors will select that section:

```
object foo.o(lib.a)
object f*(lib*)
object foo.o
object lib.a
```

See also

Initialize directive, page 305, *Do not initialize directive*, page 308, and *Keep directive*, page 309.

Extended-selectors

Syntax

```
{ [ extended-selector ][, extended-selector...] }
```

where *extended-selector* is:

```
[ first | last ]{ section-selector |
                  block name [ inline-block-def ]|
                  overlay name }
```

where *inline-block-def* is:

```
[ block-params ] extended-selectors
```

Parameters

`first`	Places the selected name first in the region, block, or overlay.
`last`	Places the selected name last in the region, block, or overlay.
`block`	The name of the block.
`overlay`	The name of the overlay.

Description

In addition to what the `section-selector` does, `extended-selector` provides functionality for placing blocks or overlays first or last in a set of sections, a block, or an overlay. It is also possible to create an *inline* definition of a block. This means that you can get more precise control over section placement.

Example

```
define block First { section .first }; /* Define a block holding
                                          the section .first */
define block Table { first block First }; /* Define a block where
                                             the first is placed
                                             first */
```

or, equivalently using an inline definition of the block `First`:

```
define block Table { first block First { section .first }};
```

See also

Define block directive, page 303, *Define overlay directive*, page 304, and *Place at directive*, page 309.

Using symbols, expressions, and numbers

In the linker configuration file, you can also:

- Define and export symbols

  The `define symbol` directive defines a symbol with a specified value that can be used in expressions in the configuration file. The symbol can also be exported to be used by the application or the debugger. See *Define symbol directive*, page 314, and *Export directive*, page 315.

- Use expressions and numbers

  In the linker configuration file, expressions and numbers are used for specifying addresses, sizes, et cetera. See *Expressions*, page 315.

Define symbol directive

Syntax

```
define [ exported ] symbol name = expr;
```

Parameters

`exported`	Exports the symbol to be usable by the executable image.
`name`	The name of the symbol.
`expr`	The symbol value.

Description

The `define symbol` directive defines a symbol with a specified value. The symbol can then be used in expressions in the configuration file. The symbols defined in this way work exactly like the symbols defined with the option `--config_def` outside of the configuration file.

The `define exported symbol` variant of this directive is a shortcut for using the directive `define symbol` in combination with the `export symbol` directive. On the command line this would require both a `--config_def` option and a `--define_symbol` option to achieve the same effect.

Note:

- A symbol cannot be redefined
- Symbols that are either prefixed by _*X*, where *X* is a capital letter, or that contain __ (double underscore) are reserved for toolset vendors.

Example

```
/* Define the symbol my_symbol with the value 4 */
define symbol my_symbol = 4;
```

See also

Export directive, page 315 and *Interaction between ILINK and the application*, page 56.

Export directive

Syntax

```
export symbol name;
```

Parameters

name	The name of the symbol.

Description

The `export` directive defines a symbol to be exported, so that it can be used both from the executable image and from a global label. The application, or the debugger, can then refer to it for setup purposes etc.

Example

```
/* Define the symbol my_symbol to be exported */
export symbol my_symbol;
```

Expressions

Syntax

An expression is built up of the following constituents:

```
expression binop expression
unop expression
expression ? expression : expression
(expression)
number
symbol
func-operator
```

where *binop* is one of these binary operators:

`+, -, *, /, %, <<, >>, <, >, ==, !=, &, ^, |, &&, ||`

where `unop` is one of this unary operators:

`+, -, !, ~`

where *number* is a number, see *Numbers*, page 316

where *symbol* is a defined symbol, see *Define symbol directive*, page 314 and *--config_def*, page 192

and where `func-operator` is one of these function-like operators:

`minimum(`*expr*`,`*expr*`)`	Returns the smallest of the two parameters.
`maximum(`*expr*`,`*expr*`)`	Returns the largest of the two parameters.
`isempty(`*r*`)`	Returns True if the region is empty, otherwise False.

`isdefinedsymbol(`*`expr-symbol`*`)`	Returns True if the expression symbol is defined, otherwise False.
`start(`*`r`*`)`	Returns the lowest address in the region.
`end(`*`r`*`)`	Returns the highest address in the region.
`size(`*`r`*`)`	Returns the size of the complete region.

where *`expr`* is an expression, and *`r`* is a region expression, see *Region expression*, page 301.

Description

In the linker configuration file, an expression is a 65-bit value with the range -2^64 to 2^64. The expression syntax closely follows C syntax with some minor exceptions. There are no assignments, casts, pre- or post-operations, and no address operations (`*`, `&`, `[]`, `->`, and `.`). Some operations that extract a value from a region expression, etc, use a syntax resembling that of a function call. A boolean expression returns 0 (false) or 1 (true).

Numbers

Syntax

`nr` `[`*`nr-suffix`*`]`

where `nr` is either a decimal number or a hexadecimal number (`0x...` or `0X...`).

and where *`nr-suffix`* is one of:

```
K          /* Kilo = (1 << 10) 1024 */
M          /* Mega = (1 << 20) 1048576 */
G          /* Giga = (1 << 30) 1073741824 */
T          /* Tera = (1 << 40) 1099511627776 */
P          /* Peta = (1 << 50) 1125899906842624 */
```

Description

A number can be expressed either by normal C means or by suffixing it with a set of useful suffixes, which provides a compact way of specifying numbers.

Example

`1024` is the same as `0x400`, which is the same as `1K`.

Structural configuration

The structural directives provide means for creating structure within the configuration, such as:

- Conditional inclusion

  An `if` directive includes or excludes other directives depending on a condition, which makes it possible to have directives for several different memory configurations in the same file. See *If directive*, page 317.
- Dividing the linker configuration file into several different files

  The `include` directive makes it possible to divide the configuration file into several logically distinct files. See *Include directive*, page 318.

If directive

Syntax

```
if (expr) {
  directives
[ } else if (expr) {
  directives ]
[ } else {
  directives ]
}
```

where *expr* is an expression, see *Expressions*, page 315.

Parameters

directives	Any ILINK directive.

Description

An `if` directive includes or excludes other directives depending on a condition, which makes it possible to have directives for several different memory configurations, for example both a banked and non-banked memory configuration, in the same file.

The directives inside an `if` part, `else if` part, or an `else` part are syntax checked and processed regardless of whether the conditional expression was true or false, but only the directives in the part where the conditional expression was true, or the `else` part if none of the conditions were true, will have any effect outside the `if` directive. The `if` directives can be nested.

Example

See *Empty region*, page 302.

Include directive

Syntax `include filename;`

Parameters

filename	A string literal where both / and \ can be used as the directory delimiter.

Description The `include` directive makes it possible to divide the configuration file into several logically distinct files. For instance, files that you need to change often and files that you seldom edit.

Section reference

The compiler places code and data into sections. Based on a configuration specified in the linker configuration file, ILINK places sections in memory.

This chapter lists all predefined sections and blocks that are available for the IAR build tools for ARM, and gives detailed reference information about each section.

For more information about sections, see the chapter *Modules and sections*, page 40.

Summary of sections

This table lists the sections and blocks that are used by the IAR build tools:

Section	Description
`.bss`	Holds zero-initialized static and global variables.
`CSTACK`	Holds the stack used by C or C++ programs.
`.cstart`	Holds the startup code.
`.data`	Holds static and global initialized variables, including the initializers.
`.data_init`	Holds initializers for.data sections.
`.difunct`	Holds pointers to code, typically C++ constructors, that should be executed by the system startup code before `main` is called.
`HEAP`	Holds the heap used for dynamically allocated data.
`.iar.dynexit`	Holds the `atexit` table.
`.intvec`	Holds the reset and interrupt vectors.
`IRQ_STACK`	Holds the stack for interrupt requests, IRQ, and exceptions.
`.noinit`	Holds `__no_init` static and global variables.
`.rodata`	Holds constant data.
`.text`	Holds the program code.

Table 38: Section summary

In addition to the ELF sections used for your application, the tools use a number of other ELF sections for a variety of purposes:

- Sections starting with `.debug` generally contain debug information in the DWARF format
- Sections starting with `.iar.debug` contain supplemental debug information in an IAR format
- The section `.comment` contains the tools and command lines used for building the file
- Sections starting with `.rel` or `.rela` contain ELF relocation information
- The section `.symtab` contains the symbol table for a file
- The section `.strtab` contains the names of the symbol in the symbol table
- The section `.shstrtab` contains the names of the sections.

Descriptions of sections and blocks

This section gives reference information about each section, where the:

- *Description* describes what type of content the section is holding and, where required, how the section is treated by the linker
- *Memory placement* describes memory placement restrictions.

For information about how to allocate sections in memory by modifying the linker configuration file, see the *Placing code and data—the linker configuration file*, page 42.

.bss

Description

Holds zero-initialized static and global variables.

Memory placement

This section can be placed anywhere in memory.

CSTACK

Description

Block that holds the internal data stack.

Memory placement

This block can be placed anywhere in memory.

See also

Setting up the stack, page 52.

.cstart

Description	Holds the startup code.
Memory placement	This section can be placed anywhere in memory.

.data

Description	Holds static and global initialized variables inlcuding initializers.
Memory placement	This section can be placed anywhere in memory.

.data_init

Description	Holds initializers for .data sections. This section is created by the linker.
Memory placement	This section can be placed anywhere in memory.

.difunct

Description	Holds the dynamic initialization vector used by C++.
Memory placement	This section can be placed anywhere in memory.

HEAP

Description	Holds the heap used for dynamically allocated data, in other words data allocated by `malloc` and `free`, and in C++, `new` and `delete`.
Memory placement	This section can be placed anywhere in memory.
See also	*Setting up the heap*, page 52.

.iar.dynexit

Description	Holds the table of calls to be made at exit.
Memory placement	This section can be placed anywhere in memory.

See also — *Setting up the atexit limit*, page 53.

.intvec

Description — Holds the reset vector and exception vectors which contain branch instructions to `cstartup`, interrupt service routines etc.

Memory placement — Must be placed at address range `0x00` to `0x3F`.

IRQ_STACK

Description — Holds the stack which is used when servicing IRQ exceptions. Other stacks may be added as needed for servicing other exception types: `FIQ`, `SVC`, `ABT`, and `UND`. The `cstartup.s` file must be modified to initialize the exception stack pointers used.

Note: This section is not used when compiling for Cortex-M.

Memory placement — This section can be placed anywhere in memory.

See also — *Exception stacks*, page 116.

.noinit

Description — Holds static and global `__no_init` variables.

Memory placement — This section can be placed anywhere in memory.

.rodata

Description — Holds constant data. This can include constant variables, string and aggregate literals, etc.

Memory placement — This section can be placed anywhere in memory.

.text

Description — Holds program code, except the code for system initialization.

Memory placement — This section can be placed anywhere in memory.

IAR utilities

This chapter describes the IAR command line utilities that are available:

- The IAR Archive Tool—iarchive—creates and manipulates a library (an archive) of several ELF object files
- The IAR ELF Tool—ielftool—performs various transformations on an ELF executable image (such as fill, checksum, format conversions, etc)
- The IAR ELF Dumper for ARM—ielfdumparm—creates a text representation of the contents of an ELF relocatable or executable image
- The IAR ELF Object Tool—iobjmanip—is used for performing low-level manipulation of ELF object files
- The IAR Absolute Symbol Exporter—isymexport—exports absolute symbols from a ROM image file, so that they can be used when you link an add-on application.

The IAR Archive Tool—iarchive

The IAR Archive Tool, `iarchive`, can create a library (an archive) file from several ELF object files. You can also use `iarchaive` to manipulate ELF libraries.

A library file contains several relocatable ELF object modules, each of which can be independently used by a linker. In contrast with object modules specified directly to the linker, each module in a library is only included if it is needed.

For information about how to build a library in the IDE, see the *IAR Embedded Workbench® IDE User Guide for ARM®.*

INVOCATION SYNTAX

The invocation syntax for the archive builder is:

`iarchive` *`parameters`*

Parameters

The parameters are:

Parameter	Description
command	Command line options that define an operation to be performed. Such an option must be specified before the name of the library file.
libraryfile	The library file to be operated on.
objectfile1 ... objectfileN	The object file(s) that the specified command operates on.
options	Command line options that define actions to be performed. These options can be placed anywhere on the command line.

Table 39: iarchive parameters

Examples

This example creates a library file called `mylibrary.a` from the source object files `module1.o`, `module.2.o`, and `module3.o`:

```
iarchive mylibrary.a module1.o module2.o module3.o.
```

This example lists the contents of `mylibrary.a`:

```
iarchive --toc mylibrary.a
```

This example replaces `module3.o` in the library with the content in the `module3.o` file and appends `module4.o` to `mylibrary.a`:

```
iarchive --replace mylibrary.a module3.o module4.o
```

SUMMARY OF IARCHIVE COMMANDS

This table summarizes the `iarchive` commands:

Command line option	Description
`--create`	Creates a library that contains the listed object files.
`--delete, -d`	Deletes the listed object files from the library.
`--extract, -x`	Extracts the listed object files from the library.
`--replace, -r`	Replaces or appends the listed object files to the library.
`--symbols`	Lists all symbols defined by files in the library.
`--toc, -t`	Lists all files in the library.

Table 40: iarchive commands summary

For more detailed descriptions, see *Descriptions of command line options*, page 325.

SUMMARY OF IARCHIVE OPTIONS

This table summarizes the `iarchive` options:

Command line option	Description
`-f`	Extends the command line.
`-o`	Specifies the library file.
`--silent, -S`	Sets silent operation.
`--verbose, -V`	Reports all performed operations.

Table 41: iarchive options summary

DESCRIPTIONS OF COMMAND LINE OPTIONS

This section gives detailed reference information about each `iarchive` command line option.

-f

Syntax

`-f filename`

Parameters

For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 156.

Description

Use this option to make `iarchive` read command line options from the named file, with the default filename extension `xcl`.

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

Both C and C++ style comments are allowed in the file. Double quotes behave in the same way as in the Microsoft Windows command line environment.

--create

Syntax

`--create libraryfile objectfile1 ... objectfileN`

Parameters

`libraryfile`	The library file that the command operates on. For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 156.

objectfile1 ... objectfileN	The object file(s) to build the library from.

Description

Use this command to build a new library from a set of object files (modules). The object files are added to the library in the exact order that they are specified on the command line.

If no command is specified on the command line, `--create` is used by default.

--delete, -d

Syntax

```
--delete libraryfile objectfile1 ... objectfileN
-d libraryfile objectfile1 ... objectfileN
```

Parameters

libraryfile	The library file that the command operates on. For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 156.
objectfile1 ... objectfileN	The object file(s) that the command operates on.

Description

Use this command to remove object files (modules) from an existing library. All object files that are specified on the command line will be removed from the library.

--extract, -x

Syntax

```
--extract libraryfile [objectfile1 ... objectfileN]
-x libraryfile [objectfile1 ... objectfileN]
```

Parameters

libraryfile	The library file that the command operates on. For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 156.
objectfile1 ... objectfileN	The object file(s) that the command operates on.

Description

Use this command to extract object files (modules) from an existing library. If a list of object files is specified, only these files are extracted. If a list of object files is not specified, all object files in the library are extracted.

-o

Syntax

```
-o libraryfile
```

Parameters

For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 156.

Description

By default, `iarchive` assumes that the first argument after the `iarchive` command is the name of the destination library. Use this option to explicitly specify a different filename for the library.

--replace, -r

Syntax

```
--replace libraryfile objectfile1 ... objectfileN
-r libraryfile objectfile1 ... objectfileN
```

Parameters

libraryfile	The library file that the command operates on. For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 156.
objectfile1 ... objectfileN	The object file(s) that the command operates on.

Description

Use this command to replace or add object files (modules) to an existing library. The object files specified on the command line either replace existing object files in the library (if they have the same name) or are appended to the library.

--silent, -S

Syntax

```
--silent
-S
```

Description

Use this option to make `iarchive` operate without sending any messages to the standard output stream.

By default, `iarchive` sends various messages via the standard output stream. You can use this option to prevent this. `iarchive` sends error and warning messages to the error output stream, so they are displayed regardless of this setting.

--toc, -t

Syntax

```
--toc libraryfile
-t libraryfile
```

Parameters

libraryfile	The library file that the command operates on. For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 156.

Description

Use this command to list the names of all object files (modules) in a specified library.

In silent mode (`--silent`), this command performs basic syntax checks on the library file, and displays only errors and warnings.

--symbols

Syntax

```
--symbols libraryfile
```

Parameters

libraryfile	The library file that the command operates on. For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 156.

Description

Use this command to list all external symbols that are defined by any object file (module) in the specified library, together with the name of the object file (module) that defines it.

In silent mode (`--silent`), this command performs symbol table-related syntax checks on the library file and displays only errors and warnings.

--verbose, -V

Syntax

```
--verbose
-V
```

Description

Use this option to make `iarchive` report which operations it performs, in addition to giving diagnostic messages.

DIAGNOSTIC MESSAGES

This section lists the messages produced by `iarchive`:

La001: could not open file *filename*

`iarchive` failed to open an object file.

La002: illegal path *pathname*

The path *`pathname`* is not a valid path.

La006: too many parameters to *cmd* command

A list of object modules was specified as parameters to a command that only accepts a single library file.

La007: too few parameters to *cmd* command

A command that takes a list of object modules was issued without the expected modules.

La008: *lib* is not a library file

The library file did not pass a basic syntax check. Most likely the file is not the intended library file.

La009: *lib* has no symbol table

The library file does not contain the expected symbol information. The reason might be that the file is not the intended library file, or that it does not contain any ELF object modules.

La010: no library parameter given

The tool could not identify which library file to operate on. The reason might be that a library file has not been specified.

La011: file *file* already exists

The file could not be created because a file with the same name already exists.

La013: file confusions, *lib* given as both library and object

The library file was also mentioned in the list of object modules.

La014: module *module* not present in archive *lib*

The specified object module could not be found in the archive.

La015: internal error

The invocation triggered an unexpected error in `iarchive`.

Ms003: could not open file *filename* for writing

`iarchive` failed to open the archive file for writing. Make sure that it is not write protected.

Ms004: problem writing to file *filename*

An error occurred while writing to file *`filename`*. A possible reason for this is that the volume is full.

Ms005: problem closing file *filename*

An error occurred while closing the file *`filename`*.

The IAR ELF Tool—ielftool

The IAR ELF Tool, `ielftool`, can generate a checksum on specific ranges of memories. This checksum can be compared with a checksum calculated on your application.

The source code for `ielftool` and a Microsoft VisualStudio 2005 template project are available in the `arm\src\elfutils` directory. If you have specific requirements for how the checksum should be generated or requirements for format conversion, you can modify the source code accordingly.

INVOCATION SYNTAX

The invocation syntax for the IAR ELF Tool is:

```
ielftool [options] inputfile outputfile [options]
```

The `ielftool` tool will first process all the fill options, then it will process all the checksum options (from left to right).

Parameters

The parameters are:

Parameter	Description
inputfile	An absolute ELF executable image produced by the ILINK linker.
options	Any of the available command line options, see *Summary of ielftool options*, page 331.
outputfile	An absolute ELF executable image.

Table 42: ielftool parameters

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 156.

Example

This example fills a memory range with `0xFF` and then calculates a checksum on the same range:

```
ielftool my_input.out my_output.out --fill 0xFF;0-0xFF
           --checksum __checksum:4,crc32;0-0xFF
```

SUMMARY OF IELFTOOL OPTIONS

This table summarizes the `ielftool` command line options:

Command line option	Description
`--bin`	Sets the format of the output file to binary.
`--checksum`	Generates a checksum.
`--fill`	Specifies fill requirements.
`--ihex`	Sets the format of the output file to linear Intel hex.
`--silent`	Sets silent operation.
`--simple`	Sets the format of the output file to Simple code.
`--srec`	Sets the format of the output file to Motorola S-records.
`--srec-len`	Restricts the number of data bytes in each S-record.
`--srec-s3only`	Restricts the S-record output to contain only a subset of records.
`--strip`	Removes debug information.
`--verbose`	Prints all performed operations.

Table 43: ielftool options summary

DESCRIPTIONS OF OPTIONS

This section gives detailed reference information about each `ielftool` option.

--bin

Syntax `--bin`

Description Sets the format of the output file to binary.

To set related options, choose:

Project>Options>Output converter

--checksum

Syntax `--checksum {symbol[+offset]|address}:size,algorithm[:flags][,start];range[;range...]`

Parameters

symbol	The name of the symbol where the checksum value should be stored. Note that it must exist in the symbol table in the input ELF file.
offset	An offset to the symbol.
address	The absolute address where the checksum value should be stored.
size	The number of bytes in the checksum: `1`, `2`, or `4`; must not be larger than the size of the checksum symbol.
algorithm	The checksum algorithm used, one of: • `sum`, a byte-wise calculated arithmetic sum. The result is truncated to 8 bits. • `sum8wide`, a byte-wise calculated arithmetic sum. The result is truncated to the size of the symbol. • `sum32`, a word-wise (32 bits) calculated arithmetic sum • `crc16`, CRC16 (generating polynomial `0x11021`); used by default • `crc32`, CRC32 (generating polynomial `0x104C11DB7`) • `crc=`*n*, CRC with a generating polynomial of *n*.
flags	`1` specifies one's complement and `2` specifies two's complement. `m` reverses the order of the bits within each byte when calculating the checksum. For example, `2m`.
start	By default, the initial value of the checksum is 0. If necessary, use *start* to supply a different initial value.
range	The address range on which the checksum should be calculated. Hexadecimal and decimal notation is allowed (for example, `0x8002–0x8FFF`).

Description

Use this option to calculate a checksum with the specified algorithm for the specified ranges. The checksum will then replace the original value in *symbol*. A new absolute symbol will be generated; with the *symbol* name suffixed with `_value` containing the calculated checksum. This symbol can be used for accessing the checksum value later when needed, for example during debugging.

If the `--checksum` option is used more than once on the command line, the options are evaluated from left to right. If a checksum is calculated for a *symbol* that is specified in a later evaluated `--checksum` option, an error is issued.

To set related options, choose:

Project>Options>Linker>Checksum

--fill

Syntax

`--fill` *`pattern`*`;range[;range...]`

Parameters

`range`	Specifies the address range for the fill. Hexadecimal and decimal notation is allowed (for example, `0x8002–0x8FFF`). Note that each address must be 4-byte aligned.
`pattern`	A hexadecimal string with the `0x` prefix (for example, `0xEF`) interpreted as a sequence of bytes, where each pair of digits corresponds to one byte (for example `0x123456`, for the sequence of bytes `0x12`, `0x34`, and `0x56`). This sequence is repeated over the fill area. If the length of the fill pattern is greater than `1` byte, it is repeated as if it started at address 0.

Description

Use this option to fill all gaps in one or more ranges with a pattern, which can be either an expression or a hexadecimal string. The contents will be calculated as if the fill pattern was repeatedly filled from the start address until the end address is passed, and then the real contents will overwrite that pattern.

If the `--fill` option is used more than once on the command line, the fill ranges cannot overlap each other.

To set related options, choose:

Project>Options>Linker>Checksum

--ihex

Syntax `--ihex`

Description Sets the format of the output file to linear Intel hex.

To set related options, choose:

Project>Options>Linker>Output converter

--silent

Syntax `--silent`

Description Causes `ielftool` to operate without sending any messages to the standard output stream.

By default, `ielftool` sends various messages via the standard output stream. You can use this option to prevent this. `ielftool` sends error and warning messages to the error output stream, so they are displayed regardless of this setting.

This option is not available in the IDE.

--simple

Syntax `--simple`

Description Sets the format of the output file to Simple code.

To set related options, choose:

Project>Options>Output converter

--srec

Syntax `--srec`

Description Sets the format of the output file to Motorola S-records.

To set related options, choose:

Project>Options>Output converter

--srec-len

Syntax `--srec-len=`*`length`*

Parameters

length	The number of data bytes in each S-record.

Description Restricts the number of data bytes in each S-record. This option can be used in combination with the `--srec` option.

This option is not available in the IDE.

--srec-s3only

Syntax `--srec-s3only`

Description Restricts the S-record output to contain only a subset of records, that is S3 and S7 records. This option can be used in combination with the `--srec` option.

This option is not available in the IDE.

--strip

Syntax `--strip`

Description Removes debug information from the ELF output file. Note that `ielftool` needs an unstripped input ELF image. If you use the `--strip` option in the linker, remove it and use the `--strip` option in `ielftool` instead.

To set related options, choose:

Project>Options>Linker>Output>Include debug information in output

--verbose

Syntax `--verbose`

Description Use this option to make `ielftool` report which operations it performs, in addition to giving diagnostic messages.

This option is not available in the IDE because this setting is always enabled.

The IAR ELF Dumper for ARM—ielfdumparm

The IAR ELF Dumper for ARM, `ielfdumparm`, can be used for creating a text representation of the contents of a relocatable or absolute ELF file.

`ielfdumparm` can be used in one of three ways:

- To produce a listing of the general properties of the input file and the ELF segments and ELF sections it contains. This is the default behavior when no command line options are used.
- To also include a textual representation of the contents of each ELF section in the input file. To specify this behavior, use the command line option `--all`.
- To produce a textual representation of selected ELF sections from the input file. To specify this behavior, use the command line option `--section`.

INVOCATION SYNTAX

The invocation syntax for `ielfdumparm` is:

`ielfdumparm` *`filename`*

Note: `ielfdumparm` is a command line tool which is not primarily intended to be used in the IDE.

Parameters

The parameters are:

Parameter	Description
`filename`	An ELF relocatable or executable file to use as input.

Table 44: ielfdumparm parameters

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 156.

SUMMARY OF IELFDUMPARM OPTIONS

This table summarizes the `ielfdumparm` command line options:

Command line option	Description
`--all`	Generates output for all input sections regardless of their names or numbers.
`-o`	Specifies an output file.

Table 45: ielfdumparm options summary

Command line option	Description
`--raw`	Uses the generic hexadecimal/ASCII output format for the contents of any selected section, instead of any dedicated output format for that section.
`--section/-s`	Generates output for selected input sections.

Table 45: ielfdumparm options summary (Continued)

DESCRIPTIONS OF OPTIONS

This section gives detailed reference information about each `ielfdumparm` option.

--all

Syntax

`--all`

Description

Use this option to include the contents of all ELF sections in the output, in addition to the general properties of the input file. Sections are output in index order, except that each relocation section is output immediately after the section it holds relocations for.

By default, no section contents are included in the output.

-o, --output

Syntax

```
-o {filename|directory}
--output {filename|directory}
```

Parameters

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 156.

Description

By default, output from the dumper is directed to the console. Use this option to direct the output to a file instead.

If you specify a directory, the output file will be named the same as the input file, only with an extra `id` extension.

--section, -s

Syntax
```
--section section_number|section_name[,...]
--s section_number|section_name[,...]
```

Parameters

section_number	The number of the section to be dumped.
section_name	The name of the section to be dumped.

Description

Use this option to dump the contents of a section with the specified number, or any section with the specified name. If a relocation section is associated with a selected section, its contents are output as well.

If you use this option, the general properties of the input file will not be included in the output.

You can specify multiple section numbers or names by separating them with commas, or by using this option more than once.

By default, no section contents are included in the output.

Example
```
-s 3,17                 /* Sections #3 and #17
-s .debug_frame,42      /* Any sections named .debug_frame and
                           also section #42 */
```

--raw

Syntax
```
--raw
```

Description

By default, many ELF sections will be dumped using a text format specific to a particular kind of section. Use this option to dump each selected ELF section using the generic text format.

The generic text format dumps each byte in the section in hexadecimal format, and where appropriate, as ASCII text.

The IAR ELF Object Tool—iobjmanip

Use the IAR ELF Object Tool, `iobjmanip`, to perform low-level manipulation of ELF object files.

INVOCATION SYNTAX

The invocation syntax for the IAR ELF Object Tool is:

`iobjmanip` *`options inputfile outputfile`*

Parameters

The parameters are:

Parameter	Description
options	Command line options that define actions to be performed. These options can be placed anywhere on the command line. At least one of the options must be specified.
inputfile	A relocatable ELF object file.
outputfile	A relocatable ELF object file with all the requested operations applied.

Table 46: iobjmanip parameters

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 156.

Examples

This example renames the section `.example` in `input.o` to `.example2` and stores the result in `output.o`:

```
iobjmanip --rename_section .example=.example2 input.o output.o
```

SUMMARY OF IOBJMANIP OPTIONS

This table summarizes the `iobjmanip` options:

Command line option	Description
`-f`	Extends the command line.
`--remove_section`	Removes a section.
`--rename_section`	Renames a section.
`--rename_symbol`	Renames a symbol.
`--strip`	Removes debug information.

Table 47: iobjmanip options summary

DESCRIPTIONS OF COMMAND LINE OPTIONS

This section gives detailed reference information about each `iobjmanip` command line option.

-f

Syntax `-f filename`

Parameters For information about specifying a filename, see *Rules for specifying a filename or directory as parameters*, page 156.

Description Use this option to make `iobjmanip` read command line options from the named file, with the default filename extension `xcl`.

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

Both C and C++ style comments are allowed in the file. Double quotes behave in the same way as in the Microsoft Windows command line environment.

--remove_section

Syntax `--remove_section {section|number}`

Parameters

section	The section—or sections, if there are more than one section with the same name—to be removed.
number	The number of the section to be removed. Section numbers can be obtained from an object dump created using `ielfdumparm`.

Description Use this option to make `iobjmanip` omit the specified section when generating the output file.

--rename_section

Syntax `--rename_section {oldname|oldnumber}=newname`

Parameters

oldname	The section—or sections, if there are more than one section with the same name—to be renamed.
oldnumber	The number of the section to be renamed. Section numbers can be obtained from an object dump created using `ielfdumparm`.
newname	The new name of the section.

Description

Use this option to make `iobjmanip` rename the specified section when generating the output file.

--rename_symbol

Syntax

`--rename_symbol` *`oldname`* `=`*`newname`*

Parameters

`oldname`	The symbol to be renamed.
`newname`	The new name of the symbol.

Description

Use this option to make `iobjmanip` rename the specified symbol when generating the output file.

--strip

Syntax

`--strip`

Description

Use this option to remove all sections containing debug information before writing the output file.

To set related options, choose:

Project>Options>Linker>Output>Include debug information in output

DIAGNOSTIC MESSAGES

This section lists the messages produced by `iobjmanip`:

Lm001: No operation given

None of the command line parameters specified an operation to perform.

Lm002: Expected *nr1* parameters but got *nr2*

Too few or too many parameters. Check invocation syntax for `iobjmanip` and for the used command line options.

Lm003: Invalid section/symbol renaming pattern *pattern*

The pattern does not define a valid renaming operation.

Lm004: Could not open file *filename*

`iobjmanip` failed to open the input file.

Lm005: ELF format error *msg*

The input file is not a valid ELF object file.

Lm006: Unsupported section type *nr*

The object file contains a section that `iobjmanip` cannot handle. This section will be ignored when generating the output file.

Lm007: Unknown section type *nr*

`iobjmanip` encountered an unrecognized section. `iobjmanip` will try to copy the content as is.

Lm008: Symbol *symbol* has unsupported format

`iobjmanip` encountered a symbol that cannot be handled. `iobjmanip` will ignore this symbol when generating the output file.

Lm009: Group type *nr* not supported

`iobjmanip` only supports groups of type `GRP_COMDAT`. If any other group type is encountered, the result is undefined.

Lm010: Unsupported ELF feature in *file*: *msg*

The input file uses a feature that `iobjmanip` does not support.

Lm011: Unsupported ELF file type

The input file is not a relocatable object file.

Lm012: Ambiguous rename for section/symbol *name* (*alt1* and *alt2*)

An ambiguity was detected while renaming a section or symbol. One of the alternatives will be used.

Lm013: Section *name1* removed due to transitive dependency on *name2*

A section was removed as it depends on an explicitly removed section.

Lm014: File has no section with index *nr*

A section index, used as a parameter to `--remove_section` or `--rename_section`, did not refer to a section in the input file.

Ms003: could not open file *filename* for writing

`iobjmanip` failed to open the output file for writing. Make sure that it is not write protected.

Ms004: problem writing to file *filename*

An error occurred while writing to file *`filename`*. A possible reason for this is that the volume is full.

Ms005: problem closing file *filename*

An error occurred while closing the file *`filename`*.

The IAR Absolute Symbol Exporter—isymexport

The IAR Absolute Symbol Exporter, `isymexport`, can export absolute symbols from a ROM image file, so that they can be used when you link an add-on application.

INVOCATION SYNTAX

The invocation syntax for the IAR Absolute Symbol Exporter is:

```
isymexport [options] inputfile outputfile [options]
```

Parameters

The parameters are:

Parameter	Description
inputfile	A ROM image in the form of an executable ELF file (output from linking).
options	Any of the available command line options, see *Summary of isymexport options*, page 344.
outputfile	A relocatable ELF file that can be used as input to linking, and which contains all or a selection of the absolute symbols in the input file. The output file contains only the symbols, not the actual code or data sections. A steering file can be used to control which symbols that are included, and also to rename some of the symbols if that is desired.

Table 48: ielftool parameters

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 156.

SUMMARY OF ISYMEXPORT OPTIONS

This table summarizes the `isymexport` command line options:

Command line option	Description
`--edit`	Specifies a steering file.
`-f`	Extends the command line; for more information, see *-f*, page 171.

Table 49: isymexport options summary

DESCRIPTIONS OF OPTIONS

This section gives detailed reference information about each `isymexport` option.

--edit

Syntax

`--edit` *`steering_file`*

Description

Use this option to specify a steering file to control what symbols that are included in the `isymexport` output file, and also to rename some of the symbols if that is desired.

See also

Steering files, page 345.

STEERING FILES

A steering file can be used for controlling which symbols that are included, and also to rename some of the symbols if that is desired. In the file, you can use `show` and `hide` directives to select which public symbols from the input file that are to be included in the output file. `rename` directives can be used for changing the names of symbols in the input file.

Syntax

The following syntax rules apply:

- Each directive is specified on a separate line.
- C comments (`/*...*/`) and C++ comments (`//...`) can be used.
- Patterns can contain wildcard characters that match more than one possible character in a symbol name.
- The `*` character matches any sequence of zero or more characters in a symbol name.
- The `?` character matches any single character in a symbol name.

Example

```
rename xxx_* as YYY_* /*Change symbol prefix from xxx_ to YYY_ */
show YYY_*            /* Export all symbols from YYY package */
hide *_internal       /* But do not export internal symbols */
show zzz?             /* Export zzza, but not zzzaaa */
hide zzzx             /* But do not export zzzx */
```

Show directive

Syntax

`show` *`pattern`*

Parameters

`pattern`	A pattern to match against a symbol name.

Description

A symbol with a name that matches the pattern will be included in the output file unless this is overridden by a later `hide` directive.

Example

```
/* Include all public symbols ending in _pub. */
show *_pub
```

Hide directive

Syntax `hide pattern`

Parameters

pattern	A pattern to match against a symbol name.

Description

A symbol with a name that matches the pattern will not be included in the output file unless this is overridden by a later `show` directive.

Example

```
/* Do not include public symbols ending in _sys. */
hide *_sys
```

Rename directive

Syntax `rename pattern1 pattern2`

Parameters

pattern1	A pattern used for finding symbols to be renamed. The pattern can contain no more than one `*` or `?` wildcard character.
pattern2	A pattern used for the new name for a symbol. If the pattern contains a wildcard character, it must be of the same kind as in *pattern1*.

Description

Use this directive to rename symbols from the output file to the input file. No exported symbol is allowed to match more than one `rename` pattern.

`rename` directives can be placed anywhere in the steering file, but they are executed before any `show` and `hide` directives. Thus, if a symbol will be renamed, all `show` and `hide` directives in the steering file must refer to the new name.

If the name of a symbol matches a *pattern1* pattern that contains no wildcard characters, the symbol will be renamed *pattern2* in the output file.

If the name of a symbol matches a *pattern1* pattern that contains a wildcard character, the symbol will be renamed *pattern2* in the output file, with part of the name matching the wildcard character preserved.

Example

```
/* xxx_start will be renamed Y_start_X in the output file,
   xxx_stop will be renamed Y_stop_X in the output file. */
rename xxx_* Y_*_X
```

DIAGNOSTIC MESSAGES

This section lists the messages produced by `isymexport`:

Es001: could not open file *filename*

`isymexport` failed to open the specified file.

Es002: illegal path *pathname*

The path *`pathname`* is not a valid path.

Es003: format error: *message*

A problem occurred while reading the input file.

Es004: no input file

No input file was specified.

Es005: no output file

An input file, but no output file was specified.

Es006: too many input files

More than two files were specified.

Es007: input file is not an ELF executable

The input file is not an ELF executable file.

Es008: unknown directive: *directive*

The specified directive in the steering file is not recognized.

Es009: unexpected end of file

The steering file ended when more input was required.

Es010: unexpected end of line

A line in the steering file ended before the directive was complete.

Es011: unexpected text after end of directive

There is more text on the same line after the end of a steering file directive.

Es012: expected *text*

The specified text was not present in the steering file, but must be present for the directive to be correct.

Es013: pattern can contain at most one * or ?

Each pattern in the current directive can contain at most one `*` or one `?` wildcard character.

Es014: rename patterns have different wildcards

Both patterns in the current directive must contain exactly the same kind of wildcard. That is, both must either contain:

- No wildcards
- Exactly one `*`
- Exactly one `?`

This error occurs if the patterns are not the same in this regard.

Es014: ambiguous pattern match: *symbol* matches more than one rename pattern

A symbol in the input file matches more than one `rename` pattern.

Implementation-defined behavior

This chapter describes how the 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 compiler adheres to a freestanding implementation of the ISO standard for the C programming language. This means that parts of a standard library can be excluded in the implementation.

Descriptions of implementation-defined behavior

This section follows the same order as the ISO appendix. Each item covered includes references to the ISO chapter and section (in parenthesis) that explains the implementation-defined behavior.

Translation

Diagnostics (5.1.1.3)

Diagnostics are produced in the form:

`filename,linenumber level[tag]: message`

where `filename` is the name of the source file in which the error was encountered, `linenumber` is the line number at which the compiler detected the error, `level` is the level of seriousness of the message (remark, warning, error, or fatal error), `tag` is a unique tag that identifies the message, and `message` is an explanatory message, possibly several lines.

Environment

Arguments to main (5.1.2.2.2.1)

The function called at program startup is called `main`. No prototype was declared for `main`, and the only definition supported for `main` is:

```
int main(void)
```

To change this behavior for the IAR DLIB runtime environment, see *Customizing system initialization*, page 76.

Interactive devices (5.1.2.3)

The streams `stdin` and `stdout` are treated as interactive devices.

Identifiers

Significant characters without external linkage (6.1.2)

The number of significant initial characters in an identifier without external linkage is 200.

Significant characters with external linkage (6.1.2)

The number of significant initial characters in an identifier with external linkage is 200.

Case distinctions are significant (6.1.2)

Identifiers with external linkage are treated as case-sensitive.

Characters

Source and execution character sets (5.2.1)

The source character set is the set of legal characters that can appear in source files. The default source character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the source character set will be the host computer's default character set.

The execution character set is the set of legal characters that can appear in the execution environment. The default execution character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the execution character set will be the host computer's default character set. The IAR DLIB Library needs a multibyte character scanner to support a multibyte execution character set.

See *Locale*, page 81.

Bits per character in execution character set (5.2.4.2.1)

The number of bits in a character is represented by the manifest constant `CHAR_BIT`. The standard include file `limits.h` defines `CHAR_BIT` as 8.

Mapping of characters (6.1.3.4)

The mapping of members of the source character set (in character and string literals) to members of the execution character set is made in a one-to-one way. In other words, the same representation value is used for each member in the character sets except for the escape sequences listed in the ISO standard.

Unrepresented character constants (6.1.3.4)

The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or in the extended character set for a wide character constant generates a diagnostic message, and will be truncated to fit the execution character set.

Character constant with more than one character (6.1.3.4)

An integer character constant that contains more than one character will be treated as an integer constant. The value will be calculated by treating the leftmost character as the most significant character, and the rightmost character as the least significant character, in an integer constant. A diagnostic message will be issued if the value cannot be represented in an integer constant.

A wide character constant that contains more than one multibyte character generates a diagnostic message.

Converting multibyte characters (6.1.3.4)

The only locale supported—that is, the only locale supplied with the IAR C/C++ Compiler—is the ‘C’ locale. If you use the command line option `--enable_multibytes`, the IAR DLIB Library will support multibyte characters if you add a locale with multibyte support or a multibyte character scanner to the library.

See *Locale*, page 81.

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 210, for information about the ranges for the different integer types.

Demotion of integers (6.2.1.2)

Converting an integer to a shorter signed integer is made by truncation. If the value cannot be represented when converting an unsigned integer to a signed integer of equal length, the bit-pattern remains the same. In other words, a large enough value will be converted into a negative value.

Signed bitwise operations (6.3)

Bitwise operations on signed integers work the same way as bitwise operations on unsigned integers; in other words, the sign-bit will be treated as any other bit.

Sign of the remainder on integer division (6.3.5)

The sign of the remainder on integer division is the same as the sign of the dividend.

Negative valued signed right shifts (6.3.7)

The result of a right-shift of a negative-valued signed integral type preserves the sign-bit. For example, shifting `0xFF00` down one step yields `0xFF80`.

Floating point

Representation of floating-point values (6.1.2.5)

The representation and sets of the various floating-point numbers adheres to IEEE 854–1987. A typical floating-point number is built up of a sign-bit (`s`), a biased exponent (`e`), and a mantissa (`m`).

See *Floating-point types*, page 213, 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 215, for information about `size_t`.

Conversion from/to pointers (6.3.4)

See *Casting*, page 215, for information about casting of data pointers and function pointers.

ptrdiff_t (6.3.6, 7.1.1)

See *ptrdiff_t*, page 216, 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 210, 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 unsigned `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 can 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 is not 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, except for escape sequences. Thus, to specify a path for an include file, use only one backslash:

```
#include "mydirectory\myfile"
```

Within source code, two backslashes are necessary:

```
file = fopen("mydirectory\\myfile","rt");
```

Recognized pragma directives (6.8.6)

In addition to the pragma directives described in the chapter *Pragma directives*, the following directives are recognized and will have an indeterminate effect:

```
alignment
baseaddr
basic_template_matching
building_runtime
can_instantiate
codeseg
cspy_support
define_type_info
do_not_instantiate
early_dynamic_initialization
function
hdrstop
important_typedef
instantiate
```

```
keep_definition
memory
module_name
no_pch
once
__printf_args
public_equ
__scanf_args
STDC
system_include
vector
warnings
```

Default __DATE__ and __TIME__ (6.8.8)

The definitions for `__TIME__` and `__DATE__` are always available.

IAR DLIB Library functions

The information in this section is valid only if the runtime library configuration you have chosen supports file descriptors. See the chapter *The DLIB runtime environment* for more information about runtime library configurations.

NULL macro (7.1.6)

The `NULL` macro is defined to `0`.

Diagnostic printed by the assert function (7.2)

The `assert()` function prints:

`filename``:`*`linenr expression`* `-- assertion failed`

when the parameter evaluates to zero.

Domain errors (7.5.1)

`NaN` (Not a Number) will be returned by the mathematic functions on domain errors.

Underflow of floating-point values sets errno to ERANGE (7.5.1)

The mathematics functions set the integer expression `errno` to `ERANGE` (a macro in `errno.h`) on underflow range errors.

fmod() functionality (7.5.6.4)

If the second argument to `fmod()` is zero, the function returns `NaN`; `errno` is set to `EDOM`.

signal() (7.7.1.1)

The signal part of the library is not supported.

Note: Low-level interface functions exist in the library, but will not perform anything. Use the template source code to implement application-specific signal handling. See *Signal and raise*, page 84.

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 80.

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 80.

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 80.

%p in printf() (7.9.6.1)

The argument to a `%p` conversion specifier, print pointer, to `printf()` is treated as having the type `void *`. The value will be printed as a hexadecimal number, similar to using the `%x` conversion specifier.

%p in scanf() (7.9.6.2)

The `%p` conversion specifier, scan pointer, to `scanf()` reads a hexadecimal number and converts it into a value with the type `void *`.

Reading ranges in scanf() (7.9.6.2)

A - (dash) character is always treated as a range symbol.

File position errors (7.9.9.1, 7.9.9.4)

On file position errors, the functions `fgetpos` and `ftell` store `EFPOS` in `errno`.

Message generated by perror() (7.9.10.4)

The generated message is:

`usersuppliedprefix``:`*`errormessage`*

Allocating zero bytes of memory (7.10.3)

The `calloc()`, `malloc()`, and `realloc()` functions accept zero as an argument. Memory will be allocated, a valid pointer to that memory is returned, and the memory block can be modified later by `realloc`.

Behavior of abort() (7.10.4.1)

The `abort()` function does not flush stream buffers, and it does not handle files, because this is an unsupported feature.

Behavior of exit() (7.10.4.3)

The argument passed to the `exit` function will be the return value returned by the `main` function to `cstartup`.

Environment (7.10.4.4)

The set of available environment names and the method for altering the environment list is described in *Environment interaction*, page 83.

system() (7.10.4.5)

How the command processor works depends on how you have implemented the `system` function. See *Environment interaction*, page 83.

Message returned by strerror() (7.11.6.2)

The messages returned by `strerror()` depending on the argument is:

Argument	Message
`EZERO`	`no error`
`EDOM`	`domain error`
`ERANGE`	`range error`
`EFPOS`	`file positioning error`
`EILSEQ`	`multi-byte encoding error`
`<0 || >99`	`unknown error`
all others	`error` *nnn*

Table 50: 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 85.

clock() (7.12.2.1)

From where the system clock starts counting depends on how you have implemented the `clock` function. See *Time*, page 85.

Glossary

This is a general glossary for terms relevant to embedded systems programming. Some of the terms do not apply to the IAR Embedded Workbench® version that you are using.

A

Absolute location
A specific memory address for an object specified in the source code, as opposed to the object being assigned a location by the IAR ILINK Linker.

Address expression
An expression which has an address as its value.

AEABI
Embedded Application Binary Interface for ARM, defined by ARM Limited.

Application
The program developed by the user of the IAR Systems toolkit and which will be run as an embedded application on a target processor.

Ar
The GNU binary utility for creating, modifying, and extracting from archives, that is, libraries. See also *Iarchive*.

Architecture
A term used by computer designers to designate the structure of complex information-processing systems. It includes the kinds of instructions and data used, the memory organization and addressing, and the methods by which the system is implemented. The two main architecture types used in processor design are *Harvard architecture* and *von Neumann architecture*.

Archive
See *Library*.

Assembler directives
The set of commands that control how the assembler operates.

Assembler options
Parameters you can specify to change the default behavior of the assembler.

Assembler language
A machine-specific set of mnemonics used to specify operations to the target processor and input or output registers or data areas. Assembler language might sometimes be preferred over C/C++ to save memory or to enhance the execution speed of the application.

Attributes
See *Section attributes*.

Auto variables
The term refers to the fact that each time the function in which the variable is declared is called, a new instance of the variable is created automatically. This can be compared with the behavior of local variables in systems using static overlay, where a local variable only exists in one instance, even if the function is called recursively. Also called local variables. Compare *Register variables*.

B

Backtrace
Information that allows the IAR C-SPY® Debugger to show, without any runtime penalty, the complete stack of function calls wherever the program counter is, provided that the code comes from compiled C functions.

Bank
See *Memory bank*.

Bank switching
Switching between different sets of memory banks. This software technique increases a computer's usable memory by allowing different pieces of memory to occupy the same address space.

Banked code
Code that is distributed over several banks of memory. Each function must reside in only one bank.

Banked data
Data that is distributed over several banks of memory. Each data object must fit inside one memory bank.

Banked memory
Has multiple storage locations for the same address. See also *Memory bank*.

Bank-switching routines
Code that selects a memory bank.

Batch files
A text file containing operating system commands which are executed by the command line interpreter. In Unix, this is called a "shell script" because it is the Unix shell which includes the command line interpreter. Batch files can be used as a simple way to combine existing commands into new commands.

Bitfield
A group of bits considered as a unit.

Block, in linker configuration file
A continuous piece of code or data. It is either built up of blocks, overlays, and sections or it is empty. A block has a name, and the start and end address of the block can be referred to from the application. It can have attributes such as a maximum size, a specific size, or a minimum alignment. The contents can have a specific order or not.

Breakpoint
1. Code breakpoint. A point in a program that, when reached, triggers some special behavior useful to the process of debugging. Generally, breakpoints are used for stopping program execution or dumping the values of some or all of the program variables. Breakpoints can be part of the program itself, or they can be set by the programmer as part of an interactive session with a debugging tool for scrutinizing the program's execution.

2. Data breakpoint. A point in memory that, when accessed, triggers some special behavior useful to the process of debugging. Generally, data breakpoints are used to stop program execution when an address location is accessed either by a read operation or a write operation.

3. Immediate breakpoint. A point in memory that, when accessed, trigger some special behavior useful in the process of debugging. Immediate breakpoints are generally used for halting the program execution in the middle of a memory access instruction (before or after the actual memory access depending on the access type) while performing some user-specified action. The execution is then resumed. This feature is only available in the simulator version of C-SPY.

C

Calling convention
A calling convention describes the way one function in a program calls another function. This includes how register parameters are handled, how the return value is returned, and which registers that will be preserved by the called function. The compiler handles this automatically for all C and C++ functions. All code written in assembler language must conform to the rules in the calling convention to be callable from C or C++, or to be able to call C and C++ functions. The C calling convention and the C++ calling conventions are not necessarily the same.

Cheap
As in *cheap memory access*. A cheap memory access either requires few cycles to perform, or few bytes of code to implement. A cheap memory access is said to have a low cost. See *Memory access cost*.

Checksum
A computed value which depends on the ROM content of the whole or parts of the application, and which is stored along with the application to detect corruption of the data. The checksum is produced by the linker to be verified with the application. Several algorithms are supported. Compare *CRC (cyclic redundancy checking)*.

Code banking
See *Banked code*.

Code model
The code model controls how code is generated for an application. Typically, the code model controls behavior such as how functions are called and in which code section functions will be located. All object files of an application must be compiled using the same code model.

Code pointers
A code pointer is a function pointer. As many cores allow several different methods of calling a function, compilers for embedded systems usually provide the users with the ability to use all these methods.

Do not confuse code pointers with data pointers.

Code sections
Read-only sections that contain code. See also *Section*.

Compilation unit
See *Translation unit*.

Compiler options
Parameters you can specify to change the default behavior of the compiler.

Configuration
See *ILINK configuration*, and *Linker configuration file*.

Cost
See *Memory access cost*.

CRC (cyclic redundancy checking)
A number derived from, and stored with, a block of data to detect corruption. A CRC is based on polynomials and is a more advanced way of detecting errors than a simple arithmetic checksum. Compare *Checksum*.

C-SPY options
Parameters you can specify to change the default behavior of the IAR C-SPY Debugger.

Cstartup
Code that sets up the system before the application starts executing.

C-style preprocessor
A preprocessor is either a stand-alone application or an integrated part of a compiler, that performs preprocessing of the input stream before the actual compilation occurs. A C-style preprocessor follows the rules set up in the ANSI specification of the C language and implements commands like `#define`, `#if`, and `#include`, which are used to handle textual macro substitution, conditional compilation, and inclusion of other files.

D

Data banking
See *Banked data*.

Data model
The data model specifies the default memory type. This means that the data model typically controls one or more of the following: The method used and the code generated to access static and global variables, dynamically allocated data, and the runtime stack. It also controls the default pointer type and in which data sections static and global variables will be located. A project can only use one data model at a time, and the same model must be used by all user modules and all library modules in the project.

Data pointers
Many cores have different addressing modes to access different memory types or address spaces. Compilers for embedded systems usually have a set of different data pointer types so they can access the available memory efficiently.

Data representation
How different data types are laid out in memory and what value ranges they represent.

Declaration
A specification to the compiler that an object, a variable or function, exists. The object itself must be defined in exactly one translation unit (source file). An object must either be declared or defined before it is used. Normally an object that is used in many files is defined in one source file. A declaration is normally placed in a header file that is included by the files that use the object.

For example:

```
/* Variable "a" exists somewhere. Function
 "b" takes two int parameters and returns an
 int. */

extern int a;
int b(int, int);
```

Definition
The variable or function itself. Only one definition can exist for each variable or function in an application. See also *Tentative definition*.

For example:

```
int a;
int b(int x, int y)
{
   return x + y;
}
```

Demangling
To restore a mangled name to the more common C/C++ name. See also *Mangling*.

Derivative
One of two or more processor variants in a series or family of microprocessors or microcontrollers.

Device description file
A file used by C-SPY that contains various device-specific information such as I/O registers (SFR) definitions, interrupt vectors, and control register definitions.

Device driver
Software that provides a high-level programming interface to a particular peripheral device.

Digital signal processor (DSP)
A device that is similar to a microprocessor, except that the internal CPU is optimized for use in applications involving discrete-time signal processing. In addition to standard microprocessor instructions, digital signal processors usually support a set of complex instructions to perform common signal-processing computations quickly.

Disassembly window
A C-SPY window that shows the memory contents disassembled as machine instructions, interspersed with the corresponding C source code (if available).

DWARF
An industry-standard debugging format which supports source level debugging. This is the format used by the IAR ILINK Linker for representing debug information in an object.

Dynamic initialization
Variables in a program written in C are initialized during the initial phase of execution, before the main function is called. These variables are always initialized with a static value, which is determined either at compile time or at link time. This is called static initialization. In C++, variables might require initialization to be performed by executing code, for example, running the constructor of global objects, or performing dynamic memory allocation.

Dynamic memory allocation
There are two main strategies for storing variables: statically at link time, or dynamically at runtime. Dynamic memory allocation is often performed from the heap and it is the size of the heap that determines how much memory that can be used for dynamic objects and variables. The advantage of dynamic memory allocation is that several variables or objects that are not active at the same time can be stored in the same memory, thus reducing the memory requirements of an application. See also *Heap memory*.

Dynamic object
An object that is allocated, created, destroyed, and released at runtime. Dynamic objects are almost always stored in memory that is dynamically allocated. Compare *Static object*.

E

EEPROM
Electrically Erasable, Programmable Read-Only Memory. A type of ROM that can be erased electronically, and then be re-programmed.

ELF
Executable and Linking Format, an industry-standard object file format. This is the format used by the IAR ILINK Linker. The debug information is formatted using DWARF.

EPROM
Erasable, Programmable Read-Only Memory. A type of ROM that can be erased by exposing it to ultraviolet light, and then be re-programmed.

Embedded C++
A subset of the C++ programming language, which is intended for embedded systems programming. The fact that performance and portability are particularly important in embedded systems development was considered when defining the language.

Embedded system
A combination of hardware and software, designed for a specific purpose. Embedded systems are often part of a larger system or product.

Emulator
An emulator is a hardware device that performs emulation of one or more derivatives of a processor family. An emulator can often be used instead of the actual core and connects directly to the printed circuit board—where the core would have been connected—via a connecting device. An emulator always behaves exactly as the processor it emulates, and is used when debugging requires all systems actuators, or when debugging device drivers.

Enea OSE Load module format
A specific ELF format that is loadable by the OSE operating system. See also *ELF*.

Enumeration
A type which includes in its definition an exhaustive list of possible values for variables of that type. Common examples include Boolean, which takes values from the list [true, false], and day-of-week which takes values [Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday]. Enumerated types are a feature of typed languages, including C and Ada.

Characters, (fixed-size) integers, and even floating-point types might be (but are not usually) considered to be (large) enumerated types.

Executable image
Contains the executable image; the result of linking several relocatable object files and libraries. The file format used for an object file is ELF with embedded DWARF for debug information.

Exceptions
An exception is an interrupt initiated by the processor hardware, or hardware that is tightly coupled with the processor, for instance, a memory management unit (MMU). The exception signals a violation of the rules of the architecture (access to protected memory), or an extreme error condition (division by zero).

Do not confuse this use of the word exception with the term *exception* used in the C++ language (but not in Embedded C++).

Expensive
As in *expensive memory access*. An expensive memory access either requires many cycles to perform, or many bytes of code to implement. An expensive memory access is said to have a high cost. See *Memory access cost*.

Extended keywords
Non-standard keywords in C and C++. These usually control the definition and declaration of objects (that is, data and functions). See also *Keywords*.

F

Filling
How to fill up bytes—with a specific fill pattern—that exists between the sections in an executable image. These bytes exist because of the alignment demands on the sections.

Format specifiers
Used to specify the format of strings sent by library functions such as `printf`. In the following example, the function call contains one format string with one format specifier, `%c`, that prints the value of `a` as a single ASCII character:

```
printf("a = %c", a);
```

G

General options
Parameters you can specify to change the default behavior of all tools that are included in the IDE.

Generic pointers
Pointers that have the ability to point to all different memory types in, for example, a core based on the Harvard architecture.

H

Harvard architecture
A core based on the Harvard architecture has separate data and instruction buses. This allows execution to occur in parallel. As an instruction is being fetched, the current instruction is executing on the data bus. Once the current instruction is complete, the next instruction is ready to go. This theoretically allows for much faster execution than a von Neumann architecture, but adds some silicon complexity. Compare *von Neumann architecture*.

Heap memory
The heap is a pool of memory in a system that is reserved for dynamic memory allocation. An application can request parts of the heap for its own use; once memory is allocated from the heap it remains valid until it is explicitly released back to the heap by the application. This type of memory is useful when the number of objects is not known until the application executes. Note that this type of memory is risky to use in systems with a limited amount of memory or systems that are expected to run for a very long time.

Heap size
Total size of memory that can be dynamically allocated.

Host
The computer that communicates with the target processor. The term is used to distinguish the computer on which the debugger is running from the core the embedded application you develop runs on.

I

Iarchive
The IAR Systems utility for creating archives, that is, libraries. Iarchive is delivered with IAR Embedded Workbench.

IDE (integrated development environment)
A programming environment with all necessary tools integrated into one single application.

Ielfdumparm
The IAR Systems utility for creating a text representation of the contents of ELF relocatable or executable image.

Ielftool
The IAR Systems utility for performing various transformations on an ELF executable image, such as fill, checksum, and format conversion.

ILINK
The IAR ILINK Linker which produces absolute output in the ELF/DWARF format.

ILINK configuration
The definition of available physical memories and the placement of sections—pieces of code and data—into those memories. ILINK requires a configuration to build an executable image.

Image
See *Executable image*.

Include file
A text file which is included into a source file. This is often done by the preprocessor.

Initialization setup in linker configuration file
Defines how to initialize RAM sections with their initializers. Normally, only non-constant non-noinit variables are initialized but, for example, pieces of code can be initialized as well.

Initialized sections
Read-write sections that should be initialized with specific values at startup. See also *Section*.

Inline assembler
Assembler language code that is inserted directly between C statements.

Inlining
An optimization that replaces function calls with the body of the called function. This optimization increases the execution speed and can even reduce the size of the generated code.

Instruction mnemonics
A word or acronym used in assembler language to represent a machine instruction. Different processors have different instruction sets and therefore use a different set of mnemonics to represent them, such as, `ADD`, `BR` (branch), `BLT` (branch if less than), `MOVE`, `LDR` (load register).

Interrupt vector
A small piece of code that will be executed, or a pointer that points to code that will be executed when an interrupt occurs.

Interrupt vector table
A table containing interrupt vectors, indexed by interrupt type. This table contains the processor's mapping between interrupts and interrupt service routines and must be initialized by the programmer.

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

Interrupts are asynchronous events that suspend normal processing and temporarily divert the flow of control through an "interrupt handler" routine. Interrupts can be caused by both hardware (I/O, timer, machine check) and software (supervisor, system call or trap instruction). Compare *Trap*.

Intrinsic
An adjective describing native compiler objects, properties, events, and methods.

Intrinsic functions
1. Function calls that are directly expanded into specific sequences of machine code. 2. Functions called by the compiler for internal purposes (that is, floating point arithmetic etc.).

Iobjmanip
The IAR Systems utility for performing low-level manipulation of ELF object files.

K

Key bindings
Key shortcuts for menu commands used in the IDE.

Keywords
A fixed set of symbols built into the syntax of a programming language. All keywords used in a language are reserved—they cannot be used as identifiers (in other words, user-defined objects such as variables or procedures). See also *Extended keywords*.

L

L-value
A value that can be found on the left side of an assignment and thus be changed. This includes plain variables and de-referenced pointers. Expressions like `(x + 10)` cannot be assigned a new value and are therefore not L-values.

Language extensions
Target-specific extensions to the C language.

Library
See *Runtime library*.

Library configuration file
A file that contains a configuration of the runtime library. The file contains information about what functionality is part of the runtime environment. The file is used for tailoring a build of a runtime library. See also *Runtime library*.

Linker configuration file
A file that contains a configuration used by ILINK when building an executable image. See also *ILINK configuration*.

Local variable
See *Auto variables*.

Location counter
See *Program location counter (PLC)*.

Logical address
See *Virtual address (logical address)*.

M

MAC (Multiply and accumulate)
A special instruction, or on-chip device, that performs a multiplication together with an addition. This is very useful when performing signal processing where many filters and transforms have the form:

$$y_j = \sum_{i=0}^{N} c_i \cdot x_{i+j}$$

The accumulator of the MAC usually has a higher precision (more bits) than normal registers. See also *Digital signal processor (DSP)*.

Macro
1. Assembler macros are user-defined sets of assembler lines that can be expanded later in the source file by referring to the given macro name. Parameters will be substituted if referred to.

2. C macro. A text substitution mechanism used during preprocessing of source files. Macros are defined using the `#define` preprocessing directive. The replacement text of each macro is then substituted for any occurrences of the macro name in the rest of the translation unit.

3. C-SPY macros are programs that you can write to enhance the functionality of C-SPY. A typical application of C-SPY macros is to associate them with breakpoints; when such a breakpoint is hit, the macro is run and can for example be used to simulate peripheral devices, to evaluate complex conditions, or to output a trace.

The C-SPY macro language is like a simple dialect of C, but is less strict with types.

Mailbox
A mailbox in an RTOS is a point of communication between two or more tasks. One task can send messages to another task by placing the message in the mailbox of the other task. Mailboxes are also known as message queues or message ports.

Mangling
Mangling is a technique used for mapping a complex C/C++ name into a simple name. Both mangled and unmangled names can be produced for C/C++ symbols in ILINK messages.

Memory, in linker configuration file
A physical memory. The number of units it contains and how many bits a unit consists of, are defined in the linker configuration file. The memory is always addressable from `0x0` to size -1.

Memory access cost
The cost of a memory access can be in clock cycles, or in the number of bytes of code needed to perform the access. A memory which requires large instructions or many instructions is said to have a higher access cost than a memory which can be accessed with few, or small instructions.

Memory area
A region of the memory.

Memory bank
The smallest unit of continuous memory in banked memory. One memory bank at a time is visible in a core's physical address space.

Memory map
A map of the different memory areas available to the core.

Memory model
Specifies the memory hierarchy and how much memory the system can handle. Your application must use only one memory model at a time, and the same model must be used by all user modules and all library modules.

Microcontroller
A microprocessor on a single integrated circuit intended to operate as an embedded system. In addition to a CPU, a microcontroller typically includes small amounts of RAM, PROM, timers, and I/O ports.

Microprocessor
A CPU contained on one (or a few) integrated circuits. A single-chip microprocessor can include other components such as memory, memory management, caches, floating-point unit, I/O ports and timers. Such devices are also known as microcontrollers.

Multi-file compilation
A technique which means that the compiler compiles several source files as one compilation unit, which enables for interprocedural optimizations such as inlining, cross call, and cross jump on multiple source files in a compilation unit.

Module
An object. An object file contains a module and library contains one or more objects. The basic unit of linking. A module contains definitions for symbols (exports) and references to external symbols (imports). When you compile C/C++, each translation unit produces one module.

N

Nested interrupts
A system where an interrupt can be interrupted by another interrupt is said to have nested interrupts.

Non-banked memory
Has a single storage location for each memory address in a core's physical address space.

Non-initialized memory
Memory that can contain any value at reset, or in the case of a soft reset, can remember the value it had before the reset.

No-init sections
Read-write sections that should not be initialized at startup. See also *Section*.

Non-volatile storage
Memory devices such as battery-backed RAM, ROM, magnetic tape and magnetic disks that can retain data when electric power is shut off. Compare *Volatile storage*.

NOP
No operation. This is an instruction that does not do anything, but is used to create a delay. In pipelined architectures, the `NOP` instruction can be used for synchronizing the pipeline. See also *Pipeline*.

O

Objcopy
A GNU binary utility for converting an absolute object file in ELF format into an absolute object file, for example the format Motorola-std or Intel-std. See also *Ielftool*.

Object
An object file or a library member.

Object file, absolute
See *Executable image*.

Object file, relocatable
The result of compiling or assembling a source file. The file format used for an object file is ELF with embedded DWARF for debug information.

Operator
A symbol used as a function, with infix syntax if it has two arguments (`+`, for example) or prefix syntax if it has only one (for instance, bitwise negation, `~`). Many languages use operators for built-in functions such as arithmetic and logic.

Operator precedence
Each operator has a precedence number assigned to it that determines the order in which the operator and its operands are evaluated. The highest precedence operators are evaluated first. Use parentheses to group operators and operands to control the order in which the expressions are evaluated.

Output image
The resulting application after linking. This term is equivalent to *executable image*, which is the term used in the IAR Systems user documentation.

Overlay, in linker configuration file
Like a block, but it contains several overlaid entities, each built up of blocks, overlays, and sections. The size of an overlay is determined by its largest constituent.

P

Parameter passing
See *Calling convention*.

Peripheral unit
A hardware component other than the processor, for example memory or an I/O device.

Pipeline
A structure that consists of a sequence of stages through which a computation flows. New operations can be initiated at the start of the pipeline even though other operations are already in progress through the pipeline.

Placement, in linker configuration file
How to place blocks, overlays, and sections into a region. It determines how pieces of code and data are actually placed in the available physical memory.

Pointer
An object that contains an address to another object of a specified type.

#pragma
During compilation of a C/C++ program, the #pragma preprocessing directive causes the compiler to behave in an implementation-defined manner. This can include, for example, producing output on the console, changing the declaration of a subsequent object, changing the optimization level, or enabling/disabling language extensions.

Pre-emptive multitasking
An RTOS task is allowed to run until a higher priority process is activated. The higher priority task might become active as the result of an interrupt. The term preemptive indicates that although a task is allotted to run a given length of time (a timeslice), it might lose the processor at any time. Each time an interrupt occurs, the task scheduler looks for the highest priority task that is active and switches to that task. If the located task is different from the task that was executing before the interrupt, the previous task is suspended at the point of interruption.

Compare *Round Robin*.

Preprocessing directives
A set of directives that are executed before the parsing of the actual code is started.

Preprocessor
See *C-style preprocessor*.

Processor variant
The different chip setups that the compiler supports. See *Derivative*.

Program counter (PC)
A special processor register that is used to address instructions. Compare *Program location counter (PLC)*.

Program location counter (PLC)
Used in the IAR Assembler to denote the code address of the current instruction. The `PLC` is represented by a special symbol (typically `$`) that can be used in arithmetic expressions. Also called simply location counter (`LC`).

PROM
Programmable Read-Only Memory. A type of ROM that can be programmed only once.

Project
The user application development project.

Project options
General options that apply to an entire project, for example the target processor that the application will run on.

Q

Qualifiers
See *Type qualifiers*.

R

Range, in linker configuration file
A range of consecutive addresses in a memory. A region is built up of ranges.

R-value
A value that can be found on the right side of an assignment. This is just a plain value. See also *L-value*.

Read-only sections
Sections that contain code or constants. See also *Section*.

Real-time operating system (RTOS)
An operating system which guarantees the latency between an interrupt being triggered and the interrupt handler starting, and how tasks are scheduled. An RTOS is typically much smaller than a normal desktop operating system. Compare *Real-time system*.

Real-time system
A computer system whose processes are time-sensitive. Compare *Real-time operating system (RTOS)*.

Region, in linker configuration file
A set of non-overlapping ranges. The ranges can lie in one or more memories. Blocks, overlays, and sections are placed into regions in the linker configuration file.

Region expression, in linker configuration file
A region built up from region literals, regions, and the common set operations possible in the linker configuration file.

Region literal, in linker configuration file
A literal that defines a set of one or more non-overlapping ranges in a memory.

Register constant
A register constant is a value that is loaded into a dedicated processor register when the system is initialized. The compiler can then generate code that assumes that the constants are present in the dedicated registers.

Register
A small on-chip memory unit, usually just one or a few bytes in size, which is particularly efficient to access and therefore often reserved as a temporary storage area during program execution.

Register locking
Register locking means that the compiler can be instructed that some processor registers shall not be used during normal code generation. This is useful in many situations. For example, some parts of a system might be written in assembler language to gain speed. These parts might be given dedicated processor registers. Or the register might be used by an operating system, or by other third-party software.

Register variables
Typically, register variables are local variables that are placed in registers instead of on the (stack) frame of the function. Register variables are much more efficient than other variables because they do not require memory accesses, so the compiler can use shorter/faster instructions when working with them. See also *Auto variables*.

Relay
A synonym to veneer, see *Veneer*.

Relocatable sections
Sections that have no fixed location in memory before linking.

Reset
A reset is a restart from the initial state of a system. A reset can originate from hardware (hard reset), or from software (soft reset). A hard reset can usually not be distinguished from the power-on condition, which a soft reset can be.

ROM-monitor
A piece of embedded software designed specifically for use as a debugging tool. It resides in the ROM of the evaluation board chip and communicates with a debugger via a serial port or network connection. The ROM-monitor provides a set of primitive commands to view and modify memory locations and registers, create and remove breakpoints, and execute your application. The debugger combines these primitives to fulfill higher-level requests like program download and single-step.

Round Robin
Task scheduling in an operating system, where all tasks have the same priority level and are executed in turn, one after the other. Compare *Pre-emptive multitasking*.

RTOS
See *Real-time operating system (RTOS)*.

Runtime library
A collection of relocatable object files that will be included in the executable image only if referred to from an object file, in other words conditionally linked.

Runtime model attributes
A mechanism that is designed to prevent modules that are not compatible to be linked into an application. ILINK uses the runtime model attributes when automatically choosing library to verify that the correct one is used.

S

Saturation arithmetics
Most, if not all, C and C++ implementations use mod–2^N 2-complement-based arithmetics where an overflow wraps the value in the value domain, that is, (127 + 1) = -128. Saturation arithmetics, on the other hand, does *not* allow wrapping in the value domain, for instance, (127 + 1) = 127, if 127 is the upper limit. Saturation arithmetics is often used in signal processing, where an overflow condition would have been fatal if value wrapping had been allowed.

Scheduler
The part of an RTOS that performs task-switching. It is also responsible for selecting which task that should be allowed to run. Many scheduling algorithms exist, but most of them are either based on static scheduling (performed at compile-time), or on dynamic scheduling (where the actual choice of which task to run next is taken at runtime, depending on the state of the system at the time of the task-switch). Most real-time systems use static scheduling, because it makes it possible to prove that the system will not violate the real-time requirements.

Scope
The section of an application where a function or a variable can be referenced by name. The scope of an item can be limited to file, function, or block.

Section
An entity that either contains data or text. Typically, one or more variables, or functions. A section is the smallest linkable unit.

Section attributes
Each section has a name and an attribute. The attribute defines what a section contains, that is, if the section content is read-only, read/write, code, data, etc.

Section fragment
A part of a section, typically a variable or a function.

Section selection
In the linker configuration file, defining a set of sections by using section selectors. A section belongs to the most restrictive section selector if it can be part of more than one selection. Three different selectors can be used individually or in conjunction to select the set of sections: *section attribute* (selecting by the section content), *section name* (selecting by the section name), and *object name* (selecting from a specific object).

Semaphore
A semaphore is a type of flag that is used for guaranteeing exclusive access to resources. The resource can be a hardware port, a configuration memory, or a set of variables. If several tasks must access the same resource, the parts of the code (the critical sections) that access the resource must be made exclusive for every task. This is done by obtaining the semaphore that protects that resource, thus blocking all other tasks from it. If another task wishes to use the resource, it also must obtain the semaphore. If the semaphore is already in use, the second task must wait until the semaphore is released. After the semaphore is released, the second task is allowed to execute and can obtain the semaphore for its own exclusive access.

Severity level
The level of seriousness of the diagnostic response from the assembler, compiler, or debugger, when it notices that something is wrong. Typical severity levels are remarks, warnings, errors, and fatal errors. A remark just points to a possible problem, while a fatal error means that the programming tool exits without finishing.

Sharing
A physical memory that can be addressed in several ways; defined in the linker configuration file.

Short addressing
Many cores have special addressing modes for efficient access to internal RAM and memory mapped I/O. Short addressing is therefore provided as an extended feature by many compilers for embedded systems. See also *Data pointers*.

Side effect
An expression in C or C++ is said to have a side-effect if it changes the state of the system. Examples are assignments to a variable, or using a variable with the post-increment operator. The C and C++ standards state that a variable that is subject to a side-effect should not be used more that once in an expression. As an example, this statement violates that rule:

```
*d++ = *d;
```

Signal
Signals provide event-based communication between tasks. A task can wait for one or more signals from other tasks. Once a task receives a signal it waits for, execution continues. A task in an RTOS that waits for a signal does not use any processing time, which allows other tasks to execute.

Simulator
A debugging tool that runs on the host and behaves as similar to the target processor as possible. A simulator is used to debug the application when the hardware is unavailable, or not needed for proper debugging. A simulator is usually not connected to any physical peripheral devices. A simulated processor is often slower, or even much slower, than the real hardware.

Single stepping
Executing one instruction or one C statement at a time in the debugger.

Skeleton code
An incomplete code framework that allows the user to specialize the code.

Special function register (SFR)
A register that is used to read and write to the hardware components of the core.

Stack frames
Data structures containing data objects like preserved registers, local variables, and other data objects that must be stored temporary for a particular scope (usually a function).

Earlier compilers usually had a fixed size and layout on a stack frame throughout a complete function, while modern compilers might have a very dynamic layout and size that can change anywhere and anytime in a function.

Standard libraries
The C and C++ library functions as specified by the C and C++ standard, and support routines for the compiler, like floating-point routines.

Statically allocated memory
This kind of memory is allocated once and for all at link-time, and remains valid all through the execution of the application. Variables that are either global or declared `static` are allocated this way.

Static object
An object whose memory is allocated at link-time and is created during system startup (or at first use). Compare *Dynamic object*.

Static overlay
Instead of using a dynamic allocation scheme for parameters and auto variables, the linker allocates space for parameters and auto variables at link time. This generates a worst-case scenario of stack usage, but might be preferable for small chips with expensive stack access or no stack access at all.

Structure value
A collecting names for structs and unions. A struct is a collection of data object placed sequentially in memory (possibly with pad bytes between them). A union is a collection of data sharing the same memory location.

Symbol
1. A name that represents a register, an absolute value, or a memory address (relative or absolute).

2. A configuration symbol that can be referred to from the executable image. The symbol is defined to be used in the linker configuration file and it has a constant value.

Symbolic location
A location that uses a symbolic name because the exact address is unknown.

T

Target
1. An architecture. 2. A piece of hardware. The particular embedded system you are developing the application for. The term is usually used to distinguish the system from the host system.

Task (thread)
A task is an execution thread in a system. Systems that contain many tasks that execute in parallel are called multitasking systems. Because a processor only executes one instruction stream at the time, most systems implement some sort of task-switch mechanism (often called context switch) so that all tasks get their share of processing time. The process of determining which task that should be allowed to run next is called scheduling. Two common scheduling methods are *Pre-emptive multitasking* and *Round Robin*.

Tentative definition
A variable that can be defined in multiple files, provided that the definition is identical and that it is an absolute variable.

Terminal I/O
A simulated terminal window in C-SPY.

Timeslice
The (longest) time an RTOS allows a task to run without running the task-scheduling algorithm. A task might be allowed to execute during several consecutive timeslices before being switched out. A task might also not be allowed to use its entire time slice, for example if, in a preemptive system, a higher priority task is activated by an interrupt.

Timer
A peripheral that counts independent of the program execution.

Translation unit
A source file together with all the header files and source files included via the preprocessor directive `#include`, except for the lines skipped by conditional preprocessor directives such as `#if` and `#ifdef`.

Trap
A trap is an interrupt initiated by inserting a special instruction into the instruction stream. Many systems use traps to call operating system functions. Another name for trap is software interrupt.

Type qualifiers
In standard C/C++, `const` or `volatile`. IAR Systems compilers usually add target-specific type qualifiers for memory and other type attributes.

U

UBROF (Universal Binary Relocatable Object Format)
File format produced by some of the IAR Systems programming tools, however, not by these tools.

V

Value expressions, in linker configuration file
A constant number that can be built up out of expressions that has a syntax similar to C expressions.

Veneer
A small piece of code that is inserted as a springboard between caller and callee when:

- There is a mismatch in mode, for example ARM and Thumb
- The call instruction does not reach its destination.

Virtual address (logical address)
An address that must be translated by the compiler, linker or the runtime system into a physical memory address before it is used. The virtual address is the address seen by the application, which can be different from the address seen by other parts of the system.

Virtual space
An IAR Embedded Workbench Editor feature which allows you to place the insertion point outside of the area where there are actual characters.

Volatile storage
Data stored in a volatile storage device is not retained when the power to the device is turned off. To preserve data during a power-down cycle, you should store it in non-volatile storage. This should not be confused with the C keyword `volatile`. Compare *Non-volatile storage*.

von Neumann architecture
A computer architecture where both instructions and data are transferred over a common data channel. Compare *Harvard architecture*.

W

Watchpoints
Watchpoints keep track of the values of C variables or expressions in the C-SPY Watch window as the application is being executed.

X

XLINK
The IAR XLINK Linker which uses the UBROF output format.

Z

Zero-initialized sections
Sections that should be initialized to zero at startup. See also *Section*.

Zero-overhead loop
A loop in which the loop condition, including branching back to the beginning of the loop, does not take any time at all. This is usually implemented as a special hardware feature of the processor and is not available in all architectures.

Zone

Different processors have widely differing memory architectures. *Zone* is the term C-SPY uses for a named memory area. For example, on processors with separately addressable code and data memory there would be at least two zones. A processor with an intricate banked memory scheme might have several zones.

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Symbols

Numerics