IAR Embedded Workbench

IAR C/C++ Development Guide

Compiling and Linking

for Arm Limited's

Arm® Cores

DARM-27

IAR SYSTEMS

EDITION NOTICE

Twenty-seventh edition: November 2021

Part number: DARM-27

This guide applies to version 9.20.x of IAR Embedded Workbench® for Arm.

Internal reference: BB9, csrct2010.1, V_110411, IJOA.

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 32-bit or 64-bit Arm cores, and need detailed reference information on how to use the build tools.

REQUIRED KNOWLEDGE

To use the tools in IAR Embedded Workbench, you should have working knowledge of:

- The architecture and instruction set of the Arm core you are using (refer to the chip manufacturer's documentation)
- The C or C++ programming language
- Application development for embedded systems
- The operating system of your host computer.

For more information about the other development tools incorporated in the IDE, refer to their respective documentation, see *Other documentation*, page 43.

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 this product, we suggest that you first go through the tutorials, which you can find in IAR Information Center in the product. They will help you get started using IAR Embedded Workbench.

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 various Arm cores and devices.
- *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.s`, 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 supported variants of the C language, and an overview of the compiler extensions, such as extensions to Standard C.
- *Using C++* gives an overview of the level of C++ support.
- *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.
- *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.
- *C/C++ standard 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.
- *The stack usage control file* describes the syntax and semantics of stack usage control files.
- *IAR utilities* describes the IAR utilities that handle the ELF and DWARF object formats.
- *Implementation-defined behavior for Standard C++* describes how the compiler handles the implementation-defined areas of Standard C++.
- *Implementation-defined behavior for Standard C* describes how the compiler handles the implementation-defined areas of Standard C.
- *Implementation-defined behavior for C89* describes how the compiler handles the implementation-defined areas of the C language standard C89.

Other documentation

User documentation is available as hypertext PDFs and as a context-sensitive online help system in HTML format. You can access the documentation from the Information Center or from the **Help** menu in the IAR Embedded Workbench IDE. The online help system is also available via the F1 key.

USER AND REFERENCE GUIDES

The complete set of IAR Systems development tools is described in a series of guides. Information about:

- System requirements and information about how to install and register the IAR Systems products are available in the *Installation and Licensing Quick Reference Guide* and the *Licensing Guide*.
- Using the IDE for project management and building, is available in the *IDE Project Management and Building Guide for Arm.*
- Using the IAR C-SPY® Debugger and C-RUN runtime error checking, is available in the *C-SPY® Debugging Guide for Arm.*
- Programming for the IAR C/C++ Compiler for Arm and linking using the IAR ILINK Linker, is available in the *IAR C/C++ Development Guide for Arm.*
- Programming for the IAR Assembler for Arm, is available in the *IAR Assembler User Guide for Arm.*
- Performing a static analysis using C-STAT and the required checks, is available in the *C-STAT® Static Analysis Guide*.
- Using I-jet, refer to the *IAR Debug Probes User Guide for I-jet®, I-jet Trace, and I-scope*.
- Using IAR J-Link and IAR J-Trace, refer to the *J-Link/J-Trace User Guide*.
- Porting application code and projects created with a previous version of the IAR Embedded Workbench for Arm, is available in the *IAR Embedded Workbench® Migration Guide*.

Note: Additional documentation might be available depending on your product installation.

THE ONLINE HELP SYSTEM

The context-sensitive online help contains information about:

- IDE project management and building
- Debugging using the IAR C-SPY® Debugger
- The IAR C/C++ Compiler
- The IAR Assembler
- 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.
- C-STAT

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.
- Labrosse, Jean J. *Embedded Systems Building Blocks: Complete and Ready-To-Use Modules in C.* R&D Books.
- Mann, Bernhard. *C für Mikrocontroller.* Franzis-Verlag. [Written in German.]
- Meyers, Scott. *Effective C++.* Addison-Wesley.
- Meyers, Scott. *More Effective C++.* Addison-Wesley.
- Meyers, Scott. *Effective STL.* Addison-Wesley.
- Sloss, Andrew N. et al, *ARM System Developer's Guide: Designing and Optimizing System Software.* Morgan Kaufmann.
- Sutter, Herb. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions.* Addison-Wesley.

The web site **isocpp.org** also has a list of recommended books about C++ programming.

WEB SITES

Recommended web sites:

- The chip manufacturer's web site.
- The Arm Limited web site, **www.arm.com**, that contains information and news about the Arm cores.
- The IAR Systems web site, **www.iar.com**, that holds application notes and other product information.
- The web site of the C standardization working group, **www.open-std.org/jtc1/sc22/wg14**.
- The web site of the C++ Standards Committee, **www.open-std.org/jtc1/sc22/wg21**.
- The C++ programming language web site, **isocpp.org**. This web site also has a list of recommended books about C++ programming.
- The C and C++ reference web site, **en.cppreference.com**.

Document conventions

When, in the IAR Systems documentation, 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 N.n\arm\doc`, where the initial digit of the version number reflects the initial digit of the version number of the IAR Embedded Workbench shared components.

TYPOGRAPHIC CONVENTIONS

The IAR Systems documentation set 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 linker or stack usage control directive, where **[** and **]** are not part of the actual directive, but any `[`, `]`, `{`, or `}` are part of the directive syntax.
{`option`**}**	A mandatory part of a linker or stack usage control directive, where **{** and **}** are not part of the actual directive, but any `[`, `]`, `{`, or `}` are part of the directive syntax.
`[option]`	An optional part of a command line option, pragma directive, or library filename.
`[a|b|c]`	An optional part of a command line option, pragma directive, or library filename with alternatives.
`{a|b|c}`	A mandatory part of a command line option, pragma directive, or library filename 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.

Table 1: Typographic conventions used in this guide

Style	Used for
	Identifies instructions specific to the IAR Embedded Workbench® IDE interface.
	Identifies instructions specific to the command line interface.
	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®, when referred to in the documentation:

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 Runtime Environment™	the DLIB runtime environment

Table 2: Naming conventions used in this guide

In 32-bit mode refers to using IAR Embedded Workbench for Arm configured for the instruction sets T32/T and A32.

In 64-bit mode refers to using IAR Embedded Workbench for Arm configured for the instruction set A64.

For more information, see *Execution modes*, page 56.

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
- Using C++
- Application-related considerations
- Efficient coding for embedded applications

Introduction to the IAR build tools

- The IAR build tools—an overview
- IAR language overview
- Device support
- Execution modes
- Special support for embedded systems

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.

IAR Embedded Workbench® is a 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, and comprehensive and specific target support. IAR Embedded Workbench promotes a useful working methodology, and therefore a significant reduction in development time.

For information about the IDE, see the *IDE Project Management and Building Guide for Arm*.

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.

THE 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.

THE 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 Arm Limited Arm Assembler, which simplifies the migration of existing code. For more information, see the *IAR Assembler User Guide for Arm*.

THE IAR ILINK LINKER

The IAR ILINK Linker for Arm 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

ILINK both uses and produces industry-standard ELF and DWARF as object format, additional IAR utilities that handle these formats are provided:

- 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 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 linking an add-on application.

Note: These ELF utilities are well-suited for object files produced by the tools from IAR Systems. Therefore, 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 *IDE Project Management and Building Guide for Arm*.

IAR language overview

The IAR C/C++ Compiler for Arm supports:

- C, the most widely used high-level programming language in the embedded systems industry. You can build freestanding applications that follow these standards:
  - Standard C—also known as C18. Hereafter, this standard is referred to as *Standard C* in this guide.
  - C89—also known as C94, C90, and ANSI C.
- Standard C++—also known as C++14. A well-established object-oriented programming language with a full-featured library well suited for modular programming. The IAR implementation of Standard C++ can be used with different levels of support for exceptions and runtime type information (RTTI).

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 common deviations from the standard. Both the strict and the relaxed mode might contain support for features in future versions of the C/C++ standards.

For more information about C, see the chapter *Using C*.

For more information about C++, see the chapter *Using C++*.

For information about how the compiler handles the implementation-defined areas of the languages, see the chapters *Implementation-defined behavior for Standard C* and *Implementation-defined behavior for Standard C++*.

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

Device support

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

Note: The object code that the compiler generates is not always binary compatible between the cores. Therefore it is crucial to specify a processor. The default core is Cortex-M3.

32-BIT ARM DEVICES

Most of the cores and devices that belong to the Armv4, Armv5, Armv6, and Armv7 generations are supported.

Armv7 profiles

The Armv7 generation consists of three *architectural profiles*:

- The A profile, the *application* profile, implemented by the Cortex-A series, compatible with AArch32.
- The R profile, the *real-time* profile, implemented by the Cortex-R series.
- The M profile, the *microcontroller* profile, implemented by most cores in the Cortex-M series.

Armv7 properties

- Armv7 devices (except the M profile) have CPU modes: User mode, Interrupt (FIQ, IRQ) mode, Supervisor mode, etc.
- Armv7 devices have these instruction sets (not all cores have all instruction sets):
  - Thumb (T), 16-bit wide instructions. Used for compact code.
  - Arm (A32), 32-bit wide instructions. Used for faster code.
  - Thumb-2 (T32), extended 32-bit wide instructions to the Thumb instruction set.
- Addresses are always 32-bit.
- The register set consists of thirteen generic 32-bit registers.
- Armv7 devices have coprocessors like VFP (vector floating point) and SIMD (serial instructions multiple data (NEON)). The coprocessors have sixteen 64-bit registers or thirty-two 128-bit registers.
- Armv7 devices use 32-bit ELF as object and image format.

64-BIT ARM DEVICES

These 64-bit Armv8-A devices are supported: Cortex-A35, Cortex-A53, and Cortex-A55. (Armv8-R and Armv8-M are not compatible.)

The Armv8-A generation defines two execution states: AArch32 and AArch64. (Not all Cortex-A cores support both execution states.)

The AArch32 execution state

The 32-bit AArch32 execution state is compatible with the Armv7-A architecture—it has the same CPU modes, instruction sets, register set, etc—and it has VFP and advanced SIMD. In this execution state, the CPU always runs **in 32-bit mode** (see *Execution modes*, page 56).

The AArch64 execution state

- AArch64 supports four levels of privilege:
  - EL0, exception level 0, user mode.
  - EL1, exception level 1, OS mode.
  - EL2, exception level 2, hypervisor mode. Optional.
  - EL3, exception level 3, secure monitor mode. Optional.

  The CPU can traverse from a higher EL to a lower one, and during that traversion it can change from the AArch64 into the AArch32 execution state.
- In the AArch64 state, the CPU runs **in 64-bit mode**. See *Execution modes*, page 56.
- AArch64 supports one instruction set, A64, that has 32-bit instructions.
- Addresses are always 64-bit.
- The register set has thirty-one 64-bit wide generic registers.
- A VFP and NEON module is always present. That module have 32 registers that are 128-bits wide.
- There are three defined data models for AArch64:
  - ILP32. It has 32-bit `long` and pointer types, and 32-bit `wchar_t` type. It uses 32-bit ELF as object and image format.
  - LP64. It has 64-bit `long` and pointer types, and 32-bit `wchar_t` type. It uses 64-bit ELF as object and image format.
  - LLP64. It has 32-bit `long` type, 64-bit pointer type, and 16-bit `wchar_t` type. IAR Embedded Workbench for Arm does **not** support this data model.

  Note: Code generated for AArch64 using the ILP32 data model cannot be linked with code generated using the LP64 data model. Neither can code generated for AArch32 and AArch64 be linked together.

PRECONFIGURED SUPPORT FILES

The IAR product installation contains preconfigured files for supporting different devices. If you need additional files for device support, they can be created using one of the provided ones as a template.

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.

Linker configuration files

The arm\config directory contains ready-made linker configuration files for all supported devices. The files have the filename extension icf and contain the information required by the linker. For more information about the linker configuration file, see *Placing code and data—the linker configuration file*, page 99, and for reference information, the chapter *The linker configuration file*.

Device description files

The debugger handles several of the device-specific requirements, such as definitions of available memory areas, peripheral registers and groups of these, by using device description files. These files are located in the arm\config directory and they have the filename extension ddf. The peripheral registers and groups of these can be defined in separate files (filename extension sfr), which in that case are included in the ddf file. For more information about these files, see the *C-SPY® Debugging Guide for Arm* and EWARM_DDFFORMAT.pdf located in the arm\doc directory.

EXAMPLES FOR GETTING STARTED

Example applications are provided with IAR Embedded Workbench. You can use these examples to get started using the development tools from IAR Systems. You can also use the examples as a starting point for your application project.

The examples are ready to be used as is. They are supplied with ready-made workspace files, together with source code files and all other related files. For information about how to run an example project, see the *IDE Project Management and Building Guide for Arm*.

Execution modes

IAR Embedded Workbench for Arm supports the 32-bit and 64-bit Arm architectures by means of execution modes.

In 32-bit mode refers to using IAR Embedded Workbench for Arm configured to generate and debug code for the instruction sets T32/T and A32, either on an Armv4/5/6/7 core or in the AArch32 execution state on an Arm v8-A core. In 32-bit mode you can use both the A32 and T32/T instruction sets and switch between them using jump instructions.

In 64-bit mode refers to using IAR Embedded Workbench for Arm configured to generate and debug code for the instruction set A64 in the AArch64 execution state on an Arm v8-A core. Code in 64-bit mode can trap into code in 32-bit mode, and that code can return back. However, the IAR translator tools do not support this switch being used in a single linked image. Switching between A32/T32/T code and A64 code must be performed by using several images. For example, an OS using 64-bit mode can start applications in either 64-bit or in 32-bit mode.

The AArch32 execution state is compatible with the Arm v7 architecture. The AArch32 execution state is emulated inside the AArch64 execution state.

Special support for embedded systems

This section briefly describes the extensions provided by the compiler to support specific features of the various Arm cores and devices.

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 controlling how to access and store data objects, as well as for controlling how a function should work internally and how it should be called/returned.

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 293 for additional information.

For more information, see the chapter *Extended keywords*. See also *Data storage* and *Functions*.

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 standard C, and are useful when you want to make sure that the source code is portable.

For more information about the pragma directives, see the chapter *Pragma directives*.

PREDEFINED SYMBOLS

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

For more information about the predefined symbols, see the chapter *The preprocessor*.

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 171.

Developing embedded applications

- Developing embedded software using IAR build tools
- The build process—an overview
- Application execution—an overview
- Building applications—an overview
- Basic project configuration

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. To assist you, compiler options, extended keywords, pragma directives, etc., are included.

MAPPING OF 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 types of memory. 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 seldom accessed but must maintain its value after power off, so it 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 more information, see *Controlling data and function placement in memory*, page 242.

The linker places sections of code and data in memory according to the directives you specify in the linker configuration file, see *Placing code and data—the linker configuration file*, page 99.

COMMUNICATION WITH PERIPHERAL UNITS

If external devices are connected to the microcontroller, you might need to initialize and control the signaling 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 (SFR). 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 53. For an example, see *Accessing special function registers*, page 255.

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 immediately stops executing the code it runs, and starts executing an interrupt routine instead.

The compiler provides various primitives for managing hardware and software interrupts, which means that you can write your interrupt routines in C, see *Interrupt functions for Cortex-M devices*, page 80 and *Interrupt functions for Arm7/9/11, Cortex-A, and Cortex-R devices*, page 81. See also *Exception functions for 64-bit mode*, page 86.

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 in conjunction with the system startup code. For more information, see *Application execution—an overview*, page 64.

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 into tasks that are 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.

See also *Managing a multithreaded environment*, page 165.

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 230.

The IAR build tools for Arm with version numbers from 8.xx and up are not fully compatible with earlier versions of the product. For more information, see the *IAR Embedded Workbench® Migration Guide for ARM*.

For more information, see *Linker optimizations*, page 127.

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.

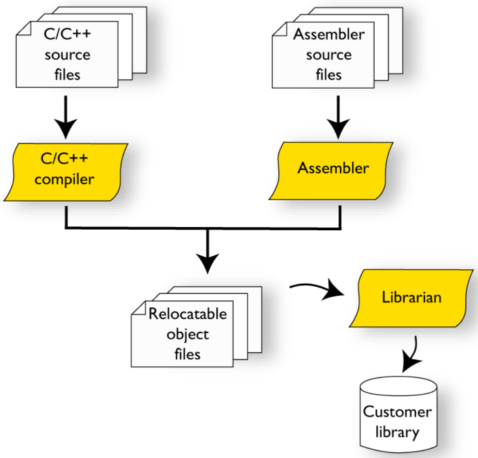
To become familiar with the process in practice, you should go through the tutorials available from the IAR Information Center.

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 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 source code into assembler source code. If required, you can modify the assembler source code which can then be assembled into object code. For more information about the IAR Assembler, see the *IAR Assembler User Guide for Arm*.

This illustration shows the translation process:



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

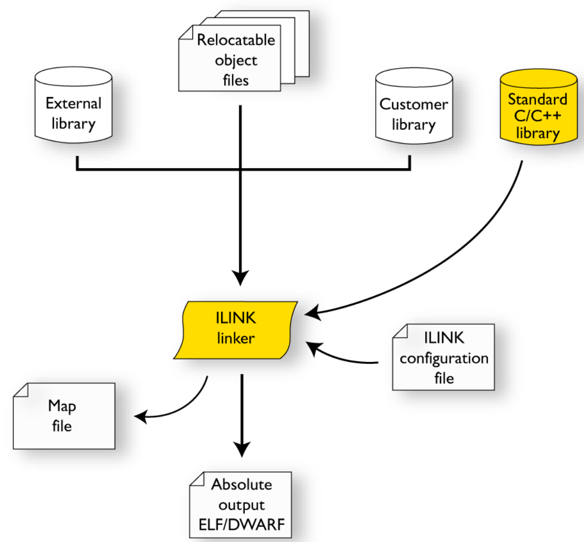
The relocatable modules in object files and libraries, produced by the IAR compiler and assembler cannot be executed as is. To become an executable application, they must be *linked*.

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

The IAR ILINK Linker (`ilinkarm.exe`) is used for building the final application. Normally, the linker 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:



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

While linking, the linker 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.

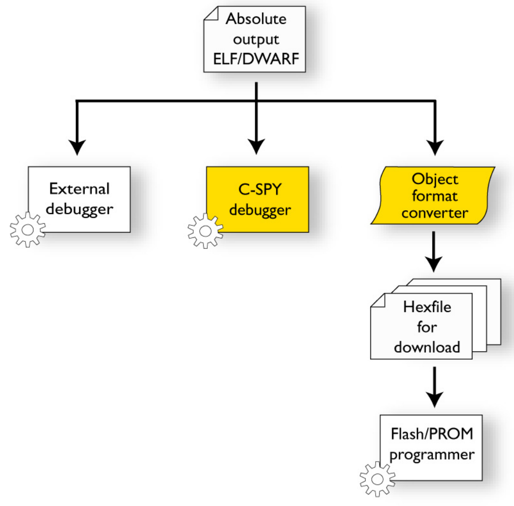
For more information about the procedure performed by the linker, see *The linking process in detail*, page 97.

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 compatible 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 557.

This illustration shows the 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. For simplicity, the initialization phase can be divided into:

- Hardware initialization, which as a minimum, generally 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/restarting the rest of the hardware, setting up the CPU, etc, in preparation for the software C/C++ system initialization.
- Software C/C++ system initialization

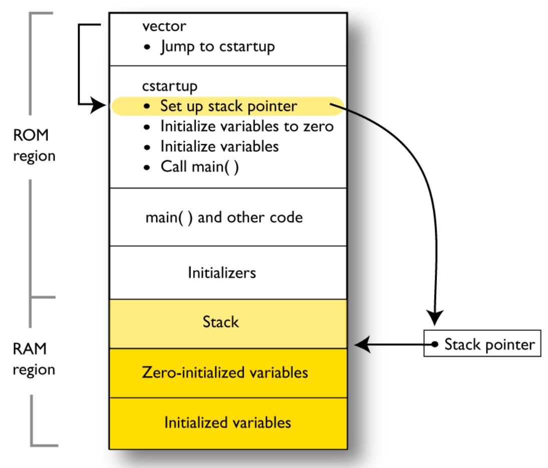
  Typically, this includes assuring that every global (statically linked) C/C++ symbol receives its proper initialization value before the `main` function is called.
- Application initialization

  This depends entirely on your application. 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.

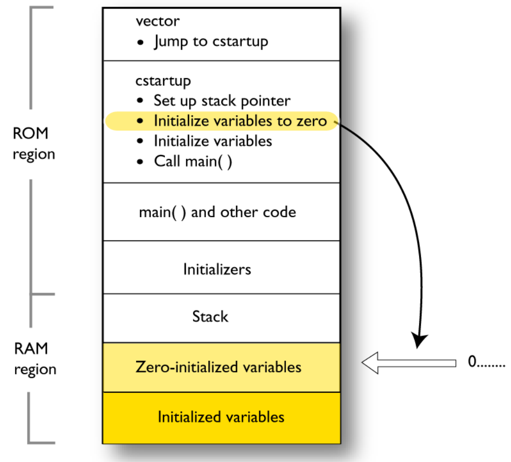
For a ROM/flash-based system, constants and functions are already placed in ROM. The linker has already divided the available RAM into different areas for variables, stack, heap, etc. All symbols placed in RAM must be initialized before the `main` function is called.

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:

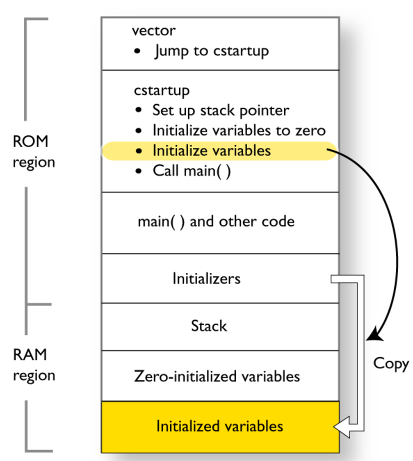


2 Then, memories that should be zero-initialized are cleared, in other words, filled with zeros:



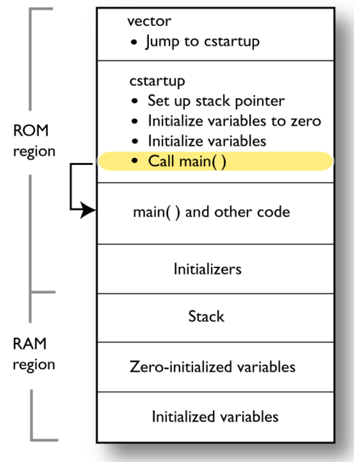
Typically, this is data referred to as *zero-initialized data*—variables declared as, for example, `int i = 0;`

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



Then, dynamically initialized static objects are constructed, such as C++ objects.

4 Finally, the `main` function is called:



For more information about each stage, see *System startup and termination*, page 150. For more information about data initialization, see *Initialization at system startup*, page 102.

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 (RTOS). 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`, `quick_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.

For more information about this, see *System termination*, page 152.

Building applications—an overview

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

```
iccarm myfile.c
```

You must also specify some critical options, see *Basic project configuration*, page 69.

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

```
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.

When building a project, the IAR Embedded Workbench IDE can produce extensive build information in the **Build** messages window. This information can be useful, for example, as a base for producing batch files for building on the command line. You can copy the information and paste it in a text file. To activate extensive build information, right-click in the **Build** messages window, and select **All** on the context menu.

Basic project configuration

This section gives an overview of the basic settings needed to 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. On the command line, you must specify each option separately, but if you use the IDE, many options will be set automatically, based on your settings of some of the fundamental options.

You need to make settings for:

- Processor configuration, that is processor variant, CPU mode, VFP and floating-point arithmetic, and byte order
- Optimization settings

- Runtime environment, see *Setting up the runtime environment*, page 133
- Customizing the ILINK configuration, see the chapter *Linking your application*.

In addition to these settings, you can use many other options and settings to fine-tune the result even further. For information about how to set options and for a list of all available options, see the chapters *Compiler options*, *Linker options*, and the *IDE Project Management and Building Guide for Arm*, respectively.

32-BIT MODE 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 most 32-bit Arm cores and devices. 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 Cortex-M3.

Execution mode should be **32-bit**. For information about setting the **Processor variant** option, see the *IDE Project Management and Building Guide for Arm*.

Use the `--cpu` option to specify the Arm core. For syntax information, see *--arm*, page 283 and *--thumb*, page 322.

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 *IDE Project Management and Building Guide for Arm*, for information about setting the **FPU** option in the IDE.

Use the `--fpu` option to specify the Arm core. For syntax information, see *--fpu*, page 296.

Byte order

The compiler supports the big-endian and little-endian byte order. All user and library modules in your application must use the same byte order.

See the *IDE Project Management and Building 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. For syntax information, see *--endian*, page 294.

64-BIT MODE PROCESSOR CONFIGURATION

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

Processor variant

Select a 64-bit Armv8-A core that the IAR C/C++ Compiler for Arm supports. 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.

Execution mode should be **64-bit**. For information about setting the **Processor variant** option, see the *IDE Project Management and Building Guide for Arm*.

Use the `--cpu` option to specify the Arm core. For syntax information, see *--cpu*, page 284, and *--aarch64*, page 281.

Data model

Select a data model to use for the generated code, ILP32 or LP64.

For information about setting the **Data model** option, see the *IDE Project Management and Building Guide for Arm*.

Use the `--abi` option to specify the data model. For syntax information, see *--abi*, page 281.

OPTIMIZATION FOR SPEED AND SIZE

The compiler's optimizer performs, among other things, dead-code elimination, constant propagation, inlining, common sub-expression elimination, static clustering, instruction scheduling, and precision reduction. It also performs loop optimizations, such as unrolling and induction variable elimination.

You can choose 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 information about compiler optimizations and for more information about efficient coding techniques, see the chapter *Efficient coding for embedded applications*.

Data storage

- Introduction
- Storage of auto variables and parameters
- Dynamic memory on the heap

Introduction

A 32-bit Arm core can address 4 Gbytes of continuous memory, ranging from `0x0` to `0xFFFF'FFFF`. A 64-bit Arm core can address 16 Exbibytes of continuous memory, ranging from `0x0` to `0xFFFF'FFFF'FFFF'FFFF`. 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 either in registers or 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. For more information, see *Storage of auto variables and parameters*, page 74.
- *Global variables, module-static 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: 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 75.

Storage of auto variables and parameters

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, while 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).
- Canaries, used in stack-protected functions. See *Stack protection*, page 92.

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.

See also *Stack considerations*, page 218 and *Setting up stack memory*, page 118.

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 either directly or indirectly—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 space. 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 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 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`.

For information about how to set up the size for heap memory, see *Setting up heap memory*, page 118.

POTENTIAL PROBLEMS

Applications that use heap-allocated data objects must be carefully designed, as 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

- Function-related extensions
- 32-bit Arm and Thumb code
- 64-bit A64 code
- Execution in RAM
- Interrupt functions for Cortex-M devices
- Interrupt functions for Arm7/9/11, Cortex-A, and Cortex-R devices
- Exception functions for 64-bit mode
- Inlining functions
- Stack protection
- TrustZone interface

Function-related extensions

In addition to supporting Standard C, the compiler provides several extensions for writing functions in C. Using these, you can:

- Generate code for the 32-bit CPU modes Arm and Thumb
- Generate code for the A64 instruction set
- Execute functions in RAM
- Write interrupt functions for the different devices
- Control function inlining
- Facilitate function optimization
- Access hardware features.
- Create interface functions for TrustZone

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 239. For information about the available intrinsic functions for accessing hardware operations, see the chapter *Intrinsic functions*.

32-bit Arm and Thumb code

In 32-bit mode, 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 using the extended keywords `__arm` and `__thumb`. You can freely mix Arm and Thumb code in the same application.

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 `0x0-0xFFFF'FFFF` 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 restrictions, see *Pointer types*, page 376.

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.

64-bit A64 code

In 64-bit mode, the IAR C/C++ Compiler for Arm can generate code for the A64 instruction set. Use the `--cpu_mode` option, alternatively the `--aarch64` or `--abi` options, to specify which instruction set should be used for your project.

When performing function calls, the compiler always attempts to generate the most efficient assembler language instruction or instruction sequence available. As a result, 16 Exbibytes of continuous memory in the range `0x0-0xFFFF'FFFF'FFFF'FFFF` can be used for placing code. There is a limit of 64 Mbytes per code module.

The size of code pointers is 4 or 8 bytes, depending on the data model. There are restrictions to implicit and explicit casts from code pointers to data pointers or integer types or vice versa. For further information about restrictions, see *Pointer types*, page 376.

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. In other words it 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 150.

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:

```
__ramfunc void test()
{
  /* myc: initializer in ROM */
  const int myc[] = { 10, 20 };

  /* string literal in ROM */
  msg("Hello");
}
```

can be rewritten to:

```
__ramfunc void test()
{
  /* myc: initialized by cstartup */
  static int myc[] = { 10, 20 };

  /* hello: initialized by cstartup */
  static char hello[] = "Hello";

  msg(hello);
}
```

For more information, see *Initializing code—copying ROM to RAM*, page 121.

Interrupt functions for Cortex-M devices

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

INTERRUPTS FOR CORTEX-M

On Cortex-M, an interrupt service routine enters and returns in the same way as a normal function, which means no special keywords are required. Therefore, the keywords `__irq`, `__fiq`, and `__nested` are not available when you compile for 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 the C-SPY debugger 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 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 257.

INTERRUPTS FOR CORTEX-M WITH FPU

For a Cortex-M core with an FPU, the system register bit `FPCCR.ASPEN` must be set to `1` to enable automatic state preservation of floating point registers (`S0`–`S15` and `FPSCR`). This will make interrupt service routines enter and return in the same way as normal functions also when floating-point registers are used.

The floating-point context saving procedure (FPCCR.ASPEN=0) can be omitted when:

- only one application task, and no interrupt handler, is going to use the FPU, or
- no application task, and only one interrupt handler, is going to use the FPU.

An application running without an operating system is regarded as one single application task. All handlers are affected, including the SVC_Handler, so software interrupt functions (functions declared with the __svc keyword) are also affected.

Interrupt functions for Arm7/9/11, Cortex-A, and Cortex-R devices

The IAR C/C++ Compiler for Arm provides the following primitives related to writing interrupt functions for Arm7/9/11, Cortex-A, and Cortex-R devices:

- The extended keywords: __irq, __fiq, __nested,
- The intrinsic functions: __enable_interrupt, __disable_interrupt, __get_interrupt_state, __set_interrupt_state

Note: Cortex-M has a different interrupt mechanism than other Arm devices, and for these devices a different set of primitives is available. For more information, see *Interrupt functions for Cortex-M devices*, page 80.

INTERRUPT FUNCTIONS

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

Interrupt service routines

In general, when an interrupt occurs in the code, the core immediately stops executing the code it runs, and starts executing an interrupt routine instead. It is 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. This is not applicable for Cortex-M devices.

Interrupt vectors and the interrupt vector table

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`.

By default, the vector table is populated with a *default interrupt handler* which loops indefinitely. For each interrupt source that has no explicit interrupt service routine, the default interrupt handler will be called. If you write your own service routine for a specific vector, that routine will override the default interrupt handler.

Defining an interrupt function—an example

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

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

For more information about the interrupt vector table, see the Arm cores documentation.

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

Interrupt and C++ member functions

Only `static` member functions can be interrupt functions. 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.

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
SVC_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 185.

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(void)
{
}
```

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. Use any of these keywords:

- To declare an interrupt function, use the `__irq` extended keyword or the `#pragma type_attribute=__irq` directive. For syntax information, see *__irq*, page 390 and *type_attribute*, page 424, 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 390, and *type_attribute*, page 424, 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` or `svc`) 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;

  /* Get interrupt vector. */
  vector = VICVectAddr;

  interrupt_task = (void(*)()) vector;

  /* Allow other IRQ interrupts to be serviced. */
  __enable_interrupt();

  /* Execute the task associated with this interrupt. */

  (*interrupt_task)();
}
```

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 `svc_number`—in this guide. The compiler provides an easy way to implicitly generate this instruction from C/C++ source code, by using the `__svc` keyword and the `#pragma svc_number` directive when declaring the function.

An `__svc` function can, for example, be declared like this:

```
#pragma svc_number=0x23
__svc int svc_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 185.

For more information, see *__svc*, page 397, and *svc_number*, page 424, respectively.

The software interrupt handler and functions

The interrupt handler—for example `SVC_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 `SVC_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 `__svc` keyword in a function definition to make the compiler generate a return sequence suited for a specific software interrupt function. The `#pragma svc_number` directive is not needed in the interrupt function definition.

For more information, see *__svc*, page 397.

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 set up 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 stack memory*, page 118.

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.

Exception functions for 64-bit mode

The compiler provides the following primitives related to writing exception functions for the 64-bit mode:

- The extended keywords `__exception` , `__nested`, and `__svc`
- The intrinsic functions `__enable_interrupt` and `__disable_interrupt`

- The special function names Synchronous_Handler_A64, Error_Handler_A64, IRQ_Handler_A64, and FIQ_Handler_A64

EXCEPTION FUNCTIONS

An exception function is used for handling external interrupt events or internal exceptions. When an exception occurs, the code executed in the core is stopped and code in an exception starts executing instead. It is important that the environment of the excepted code is restored after the exception has been handled—this includes the values of the processor registers, status registers, etc. The execution can then continue as if no exception took place.

`__exception` is a function type attribute that defines an exception function. It must have `void` as a return value, and cannot have parameters. All used registers are saved at entrance and restored at exit. It returns with an `ERET` instruction.

```
__exception void func(void)
{
    /* Do something */
}
```

EXCEPTIONS AND C++ MEMBER FUNCTIONS

Only `static` member functions can be exception functions. When a non-static member function is called, it must be applied to an object. When an exception occurs and the exception function is called, there is no object available to apply the member function to.

EXCEPTION VECTOR TABLE

The IAR C/C++ Compiler uses the same exception vector table for the three exception levels EL1, EL2, and EL3. The exception vector table starts at the linker-defined symbol `__eevector`. It has 16 vectors, each 128 bytes large.

The IAR C/C++ Compiler only defines the vectors for exceptions that do not change exception level and that use the `SP` for the current exception level (offsets `0x200`, `0x280`, `0x300`, and `0x380`). Those four defined vectors have the names `Synchronous_Handler_A64`, `Error_Handler_A64`, `IRQ_Handler_A64`, and `FIQ_Handler_A64`. They have a default implementation that can be overridden by defining an `__exception` function with one of those names. If the function is too large to fit in a vector, the compiler will issue an error. The function cannot then be used directly as an exception function. Instead you must:

1 Write an assembler module that starts with a global symbol, for example `ee`. The symbol should jump to the exception function.

2 Edit the linker configuration file. Replace the `place at` directive for the relevant exception function (for example `Synchronous_Handler_A64`) with `place at address synchronous_evector { symbol ee };`.

By default, the exception vector table is placed at address 2048. To place it at another address, use one of these methods:

- Use the linker option `--config_def` to set the linker configuration symbol `__Exception_table_address`, like this:
  `--config_def __Exception_table_address=4096`
- Edit the linker configuration file that the project uses

The exception table must be 2Kbyte-aligned.

NESTED EXCEPTION FUNCTIONS

An exception function can be nested. This also saves the `ELR_EL1` system register at entrance. When the function exits, interrupts are disabled and all saved registers are restored.

Note that the `SPSR_EL1` system register is not saved automatically. It must be saved explicitly before interrupts are enabled, for the status flags to be preserved after the exception. Doing this explicitly allows other bits of `SPSR_EL1` to be manipulated.

An example:

```
#include <intrinsics.h>
__exception __nested void func(void)
{
    // All used registers + ELR_EL1 have been saved. SPSR_EL1
    // and ESR_EL1 can be saved/used.

    __enable_interrupt();

    // Do stuff

    __disable_interrupt();

    // The possibly changed SPSR_EL1 and ESR_EL1 can be restored.
    // At exit, interrupts will be disabled and then all used
    // registers are restored. Then ERET is executed.
}
```

SUPERVISOR-DEFINED FUNCTIONS

A function defined using the function type attribute `__svc` can have return values and can take parameters. It preserves the same registers as a normal function call, and returns with an `ERET` instruction. An SVC-defined function handles synchronous exceptions.

```
__svc void func(void)
{
    /* Do something */
}
```

See below for an example.

Supervisor call

`SVC` is an A64 instruction that makes a supervisor call, that is, an exception. It is handled by the synchronous exception vector. The IAR C/C++ Compiler supports exchanging the normal call instruction used to call a function with the `SVC` instruction, by using the pragma directive `svc_number` in front of any function declaration or definition. The supplied number will be stored in the `ESR_EL1` system register.

```
#pragma svc_number = 23
__svc int Synchronous_Handler_A64(int i)
{
    return i;
}

void f()
{
    int i = Synchronous_Handler_A64(5); // Will use an SVC
}
```

The intended use for SVC functions is to let code executing in a lower exception level call code in a higher exception level.

```
// User code
#pragma svc_number = 1
int svc1(int);
#pragma svc_number = 2
int svc2(int);

int main(void)
{
    svc1(1);
}

// Supervisor code
__svc int Synchronous_Handler_A64(int a)
{
    // Get syndrome: AARCH64 SVC
    long long nr = 0;
    __asm("MRS %x0, ESR_EL1\n" : "=r"(nr));
    int ec = (nr >> 26) & 0x3F;
    if (ec != 0x15)
        return -1;

    // Get SVC number.
    nr &= 0xFF'FFFF;

    return nr + a;
}
```

Functions declared with `#pragma svc_number` do not have to use the same function signature. If different signatures are used, `Synchronous_Handler_A64` must be written in assembler language as a trampoline to the various calling handlers, in order to pass parameters and handle return values correctly.

RESET ADDRESS

By default, the reset address is assumed to be at address 0. To place it at another address, use one of these methods:

- Use the linker option `--config_def` to set the linker configuration symbol `__Reset_address`, like this: `--config_def __Reset_address=4096`
- Edit the linker configuration file that the project uses

Inlining functions

Function inlining means that a function, whose definition is known at compile time, is integrated into the body of its caller to eliminate the overhead of the function call. This optimization, which is performed at optimization level High, normally reduces execution time, but might increase the code size. The resulting code might become more difficult to debug. Whether the inlining actually occurs is subject to the compiler's heuristics.

The compiler heuristically decides which functions to inline. Different heuristics are used when optimizing for speed, size, or when balancing between size and speed. Normally, code size does not increase when optimizing for size.

C VERSUS C++ SEMANTICS

In C++, all definitions of a specific inline function in separate translation units must be exactly the same. If the function is not inlined in one or more of the translation units, then one of the definitions from these translation units will be used as the function implementation.

In C, you must manually select one translation unit that includes the non-inlined version of an inline function. You do this by explicitly declaring the function as `extern` in that translation unit. If you declare the function as `extern` in more than one translation unit, the linker will issue a *multiple definition* error. In addition, in C, inline functions cannot refer to static variables or functions.

For example:

```
// In a header file.
static int sX;
inline void F(void)
{
  //static int sY; // Cannot refer to statics.
  //sX;            // Cannot refer to statics.
}

// In one source file.
// Declare this F as the non-inlined version to use.
extern inline void F();
```

FEATURES CONTROLLING FUNCTION INLINING

There are several mechanisms for controlling function inlining:

- The `inline` keyword advises the compiler that the function defined immediately after the directive should be inlined.

  If you compile your function in C or C++ mode, the keyword will be interpreted according to its definition in Standard C or Standard C++, respectively.

  The main difference in semantics is that in Standard C you cannot (in general) simply supply an inline definition in a header file. You must supply an external definition in one of the compilation units, by designating the inline definition as being external in that compilation unit.

- #pragma inline is similar to the inline keyword, but with the difference that the compiler always uses C++ inline semantics.

  By using the #pragma inline directive you can also disable the compiler's heuristics to either force inlining or completely disable inlining. For more information, see *inline*, page 413.
- --use_c++_inline forces the compiler to use C++ semantics when compiling a Standard C source code file.
- --no_inline, #pragma optimize=no_inline, and #pragma inline=never all disable function inlining. By default, function inlining is enabled at optimization level High.

The compiler can only inline a function if the definition is known. Normally, this is restricted to the current translation unit. However, when the --mfc compiler option for multi-file compilation is used, the compiler can inline definitions from all translation units in the multi-file compilation unit. For more information, see *Multi-file compilation units*, page 247.

For more information about the function inlining optimization, see *Function inlining*, page 250.

Stack protection

In software, a stack buffer overflow occurs when a program writes to a memory address on the program's call stack outside of the intended data structure, which is usually a fixed-length buffer. The result is, almost always, corruption of nearby data, and it can even change which function to return to. If it is deliberate, it is often called stack smashing. One method to guard against stack buffer overflow is to use stack canaries, named for their analogy to the use of canaries in coal mines.

STACK PROTECTION IN THE IAR C/C++ COMPILER

The IAR C/C++ Compiler for Arm supports stack protection.

To enable stack protection for functions considered needing it, use the compiler option --stack_protection. For more information, see *--stack_protection*, page 320.

The IAR Systems implementation of stack protection uses a heuristic to determine whether a function needs stack protection or not. If any defined local variable has the array type or a structure type that contains a member of array type, the function will need stack protection. In addition, if the address of any local variable is propagated outside of a function, such a function will also need stack protection.

If a function needs stack protection, the local variables are sorted to let the variables with array type to be placed as high as possible in the function stack block. After those variables, a canary element is placed. The canary is initialized at function entrance. The initialization value is taken from the global variable `__stack_chk_guard`. At function exit, the code verifies that the canary element still contains the original value. If not, the function `__stack_chk_fail` is called.

USING STACK PROTECTION IN YOUR APPLICATION

To use stack protection, you must define these objects in your application:

- `extern uint32_t __stack_chk_guard`

  The global variable `__stack_chk_guard` must be initialized prior to first use. If the initialization value is randomized, it will be more secure.

- `__interwork __nounwind __noreturn void __stack_chk_fail(void)`

  The purpose of the function `__stack_chk_fail` is to notify about the problem and then terminate the application.

  Note: The return address from this function will point into the function that failed.

The file `stack_protection.c` in the directory `arm\src\lib\runtime` can be used as a template for both `__stack_chk_guard` and `__stack_chk_fail`.

TrustZone interface

TrustZone for Arm v8-M (**32-bit mode**) needs some compiler support to create a secure interface between the secure and the non-secure code. For this purpose, there are two function type attributes that control how code is generated: `__cmse_nonsecure_entry` and `__cmse_nonsecure_call`. For more information, see *Arm TrustZone®*, page 234.

Note: TrustZone support is automatic **in 64-bit mode**.

Linking using ILINK

- Linking—an overview
- Modules and sections
- The linking process in detail
- Placing code and data—the linker configuration file
- Initialization at system startup
- Stack usage analysis

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.

The linker 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).

The linker 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. Furthermore, the linker 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 123.

The linker 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 compatible debugger that supports ELF/DWARF, or it can be stored in EPROM or flash.

To handle ELF files, various tools are included. For information about included utilities, see *Specific ELF tools*, page 52.

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 version of the runtime environment
- Optionally, debug information in DWARF format
- A symbol table of all global symbols and all external symbols used.

Note: In a library, each module (source file) should only contain one single function. This is important if you want to override a function in a library with a function in your own application. The linker includes modules only if they are referred to from the rest of the application. If the linker includes a library module that contains several functions because one function is referred to, and another function in that module should be overridden by a function defined by your application, the linker issues a "duplicate definitions" error.

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 main purpose of section attributes is to distinguish between sections that can be placed in ROM and sections that must be placed in RAM:

`ro|readonly`	ROM sections
`rw|readwrite`	RAM sections

In each category, sections can be further divided into those that contain code and those that contain data, resulting in four main categories:

`ro code`	Normal code
`ro data`	Constants
`rw code`	Code copied to RAM
`rw data`	Variables

`readwrite data` also has a subcategory—`zi|zeroinit`—for sections that are zero-initialized at application startup.

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 117, and *Keeping symbols and sections*, page 117.

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 addresses to the various sections used by the application.

The IAR build tools have many predefined section names. For more information about each section, see the chapter *Section reference*.

You can group sections together for placement by using blocks. See *define block directive*, page 511.

The linking process in detail

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*.

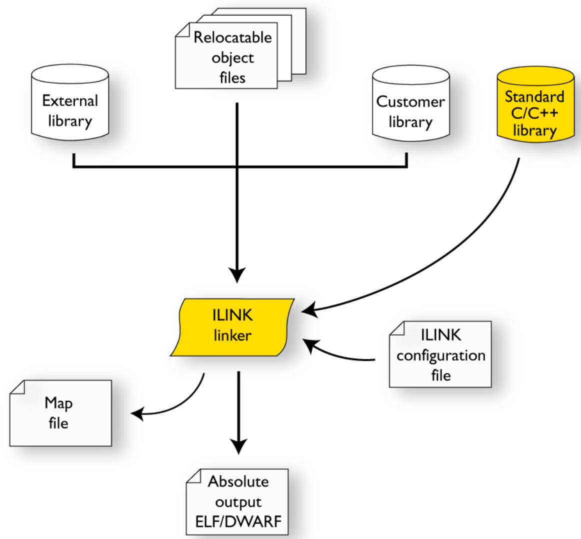
Note: Modules produced by a toolset from another vendor can be included in the build as well, as long as the module is AEABI (Arm Embedded Application Binary Interface) compliant. Be aware that this might also require a compiler utility library from the same vendor.

The 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.
- Handle symbols with more than one definition. If there is more than one non-weak definition, an error is emitted. Otherwise, one of the definitions is picked (the non-weak one, if there is one) and the others are suppressed. Weak definitions are typically used for inline and template functions. If you need to override some of the non-weak definitions from a library module, you must ensure that the library module is not included (typically by providing alternate definitions for all the symbols your application uses in that library module).
- 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 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 file that contains the executable image and any debug information provided. The contents of each needed section in the relocatable input files is calculated using the relocation information supplied in its file and the addresses determined when placing sections. This process can result in one or more relocation failures if some of the requirements for a particular section are not met, for instance if placement resulted in the destination address for a `PC`-relative jump instruction being out of range for that instruction.

- 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:



During the linking, ILINK might produce error 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—ielfdump*, page 559.

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

Assume a simple 32-bit architecture that has these memory prerequisites:

- There are 4 Gbytes of addressable memory.
- There is ROM memory in the address range `0x0000–0x10000`.
- There is RAM memory in the range `0x20000–0x30000`.
- The stack has an alignment of 8.
- The system startup code must be located at a fixed address.

A simple configuration file for this assumed architecture 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 */
initialize by copy { readwrite }; /* Initialize RW 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. Furthermore, 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.

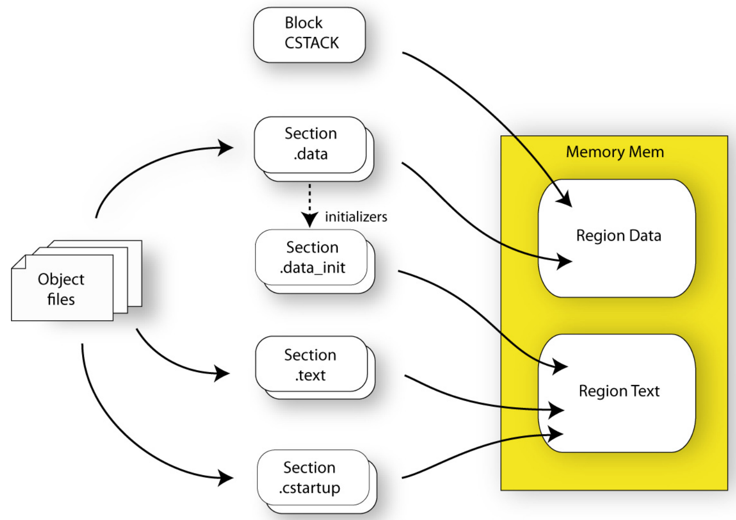
The last part of the configuration file handles the actual placement of all the sections into the available regions. First, the startup code—defined to reside in the read-only (readonly) section .cstartup—is placed at the start of the ROM region, that is at address 0x10000.

Note: 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: 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:



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

Initialization at system startup

In Standard 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 are exceptions to this rule, for example, 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 3: Sections holding initialized data

Note: Clustering of static variables might group zero-initialized variables together with initialized data in `.data`. The compiler can decide to place constants in the `.text` section to avoid loading the address of a constant from a constant table.

For information about 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, or not initialized at all (`__no_init`) are handled automatically by ILINK.
- 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 into two sections, where the original initialized section will keep the name. The contents are placed in the new initializer section, which will get 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 which the linker splits into `.data` and `.data_init`.
- Sections that contains constants should not be initialized—they should only be placed in flash/ROM.

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

```
/* Handle initialization */
initialize by copy { readwrite }; /* Initialize RW 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                   */
```

Note: When compressed initializers are used (see *initialize directive*, page 517), the contents sections (that is, sections with the `_init` suffix) are not listed as separate sections in the map file. Instead, they are combined into aggregates of "initializer bytes". You can place the contents sections the usual way in the linker configuration file, however, this affects the placement—and possibly the number—of the "initializer bytes" aggregates.

For more information about and examples of how to configure the initialization, see *Linking considerations*, page 113.

C++ DYNAMIC INITIALIZATION

The compiler places subroutine pointers for performing C++ dynamic initialization into sections of the ELF section types `SHT_PREINIT_ARRAY` and `SHT_INIT_ARRAY`. By default, the linker will place these into a linker-created block, ensuring that all sections of the section type `SHT_PREINIT_ARRAY` are placed before those of the type `SHT_INIT_ARRAY`. If any such sections were included, code to call the routines will also be included.

The linker-created blocks are only generated if the linker configuration does not contain section selector patterns for the `preinit_array` and `init_array` section types. The effect of the linker-created blocks will be very similar to what happens if the linker configuration file contains this:

```
define block SHT$$PREINIT_ARRAY { preinit_array };
define block SHT$$INIT_ARRAY { init_array };
define block CPP_INIT with fixed order { block
                                         SHT$$PREINIT_ARRAY,
                                         block SHT$$INIT_ARRAY };
```

If you put this into your linker configuration file, you must also mention the `CPP_INIT` block in one of the section placement directives. If you wish to select where the linker-created block is placed, you can use a section selector with the name `".init_array"`.

See also *section-selectors*, page 525.

Stack usage analysis

This section describes how to perform a stack usage analysis using the linker.

In the `arm\src` directory, you can find an example project that demonstrates stack usage analysis.

INTRODUCTION TO STACK USAGE ANALYSIS

Under the right circumstances, the linker can accurately calculate the maximum stack usage for each call graph, starting from the program start, interrupt functions, tasks etc. (each function that is not called from another function, in other words, the root).

If you enable stack usage analysis, a stack usage chapter will be added to the linker map file, listing for each call graph root the particular call chain which results in the maximum stack depth.

The analysis is only accurate if there is accurate stack usage information for each function in the application.

In general, the compiler will generate this information for each C function, but if there are indirect calls—calls using function pointers—in your application, you must supply a list of possible functions that can be called from each calling function.

If you use a stack usage control file, you can also supply stack usage information for functions in modules that do not have stack usage information.

You can use the `check that` directive in your stack usage control file to check that the stack usage calculated by the linker does not exceed the stack space you have allocated.

PERFORMING A STACK USAGE ANALYSIS

1 Enable stack usage analysis:

In the IDE, choose **Project>Options>Linker>Advanced>Enable stack usage analysis**.

On the command line, use the linker option `--enable_stack_usage`.

See *--enable_stack_usage*, page 340.

2 Enable the linker map file:

In the IDE, choose **Project>Options>Linker>List>Generate linker map file**

On the command line, use the linker option `--map`

3 Link your project.

Note: The linker will issue warnings related to stack usage under certain circumstances, see *Situations where warnings are issued*, page 110.

4 Review the linker map file, which now contains a stack usage chapter with a summary of the stack usage for each call graph root. For more information, see *Result of an analysis—the map file contents*, page 106.

5 For more details, analyze the call graph log, see *Call graph log*, page 110.

Note: There are limitations and sources of inaccuracy in the analysis, see *Limitations*, page 109.

You might need to specify more information to the linker to get a more representative result. See *Specifying additional stack usage information*, page 108

In the IDE, choose **Project>Options>Linker>Advanced>Enable stack usage analysis>Control file**.

On the command line, use the linker option `--stack_usage_control`.

See *--stack_usage_control*, page 360.

6 To add an automatic check that you have allocated memory enough for the stack, use the `check that` directive in your linker configuration file. For example, assuming a stack block named `MY_STACK`, you can write like this:

```
check that size(block MY_STACK) >=maxstack("Program entry")
                                + totalstack("interrupt") + 100;
```

When linking, the linker emits an error if the check fails. In this example, an error will be emitted if the sum of the following exceeds the size of the `MY_STACK` block:

- The maximum stack usage in the category `Program entry` (the main program).
- The sum of each individual maximum stack usage in the category `interrupt` (assuming that all interrupt routines need space at the same time).
- A safety margin of 100 bytes (to account for stack usage not visible to the analysis).

See also *check that directive*, page 529 and *Stack considerations*, page 218.

RESULT OF AN ANALYSIS—THE MAP FILE CONTENTS

When stack usage analysis is enabled, the linker map file contains a stack usage chapter with a summary of the stack usage for each call graph root category, and lists the call chain that results in the maximum stack depth for each call graph root. This is an example of what the stack usage chapter in the map file might look like:

```
*****************************************************************
  *** STACK USAGE
  ***

  Call Graph Root Category  Max Use  Total Use
  ------------------------  -------  ---------
  interrupt                     104        136
  Program entry                 168        168

Program entry
  "__iar_program_start": 0x000085ac
  Maximum call chain                        168 bytes

    "__iar_program_start"                     0
    "__cmain"                                 0
    "main"                                    8
    "printf"                                 24
    "_PrintfTiny"                            56
    "_Prout"                                 16
    "putchar"                                16
    "__write"                                 0
    "__dwrite"                                0
    "__iar_sh_stdout"                        24
    "__iar_get_ttio"                         24
    "__iar_lookup_ttioh"                      0

interrupt
  "FaultHandler": 0x00008434

  Maximum call chain                         32 bytes

    "FaultHandler"                           32

interrupt
  "IRQHandler": 0x00008424

  Maximum call chain                        104 bytes

    "IRQHandler"                             24
    "do_something" in suexample.o [1]        80
```

The summary contains the depth of the deepest call chain in each category as well as the sum of the depths of the deepest call chains in that category.

Each call graph root belongs to a call graph root category to enable convenient calculations in `check that` directives.

SPECIFYING ADDITIONAL STACK USAGE INFORMATION

To specify additional stack usage information you can use either a stack usage control file (`suc`) where you specify stack usage control directives or annotate the source code.

You can:

- Specify complete stack usage information (call graph root category, stack usage, and possible calls) for a function, by using the stack usage control directive `function`. Typically, you do this if stack usage information is missing, for example in an assembler module. In your `suc` file you can, for example, write like this:

  ```
  function MyFunc: 32,
    calls MyFunc2,
    calls MyFunc3, MyFunc4: 16;

  function [interrupt] MyInterruptHandler: 44;
  ```

  See also *function directive*, page 546.

- Exclude certain functions from stack usage analysis, by using the stack usage control directive `exclude`. In your `suc` file you can, for example, write like this:

  ```
  exclude MyFunc5, MyFunc6;
  ```

  See also *exclude directive*, page 546.

- Specify a list of possible destinations for indirect calls in a function, by using the stack usage control directive `possible calls`. Use this for functions which are known to perform indirect calls and where you know exactly which functions that might be called in this particular application. In your `suc` file you can, for example, write like this:

  ```
  possible calls MyFunc7: MyFunc8, MyFunc9;
  ```

  If the information about which functions that might be called is available at compile time, consider using the `#pragma calls` directive instead.

  See also *possible calls directive*, page 548 and *calls*, page 405.

- Specify that functions are call graph roots, including an optional call graph root category, by using the stack usage control directive `call graph root` or the `#pragma call_graph_root` directive. In your `suc` file you can, for example, write like this:

  ```
  call graph root [task]: MyFunc10, MyFunc11;
  ```

  If your interrupt functions have not already been designated as call graph roots by the compiler, you must do so manually. You can do this either by using the `#pragma` `call_graph_root` directive in your source code or by specifying a directive in your `suc` file, for example:

```
call graph root [interrupt]: Irq1Handler, Irq2Handler;
```

See also *call graph root directive*, page 546 and *call_graph_root*, page 406.

- Specify a maximum number of iterations through any of the cycles in the recursion nest of which the function is a member. In your `suc` file you can, for example, write like this:

```
max recursion depth MyFunc12: 10;
```

- Selectively suppress the warning about unmentioned functions referenced by a module for which you have supplied stack usage information in the stack usage control file. Use the `no calls from` directive in your `suc` file, for example, like this:

```
no calls from [file.o] to MyFunc13, MyFunc14;
```

- Instead of specifying stack usage information about assembler modules in a stack usage control file, you can annotate the assembler source with call frame information. For more information, see the *IAR Assembler User Guide for Arm*.

For more information, see the chapter *The stack usage control file*.

LIMITATIONS

Apart from missing or incorrect stack usage information, there are also other sources of inaccuracy in the analysis:

- The linker cannot always identify all functions in object modules that lack stack usage information. In particular, this might be a problem with object modules written in assembler language or produced by non-IAR tools. You can provide stack usage information for such modules using a stack usage control file, and for assembler language modules you can also annotate the assembler source code with `CFI` directives to provide stack usage information. See the *IAR Assembler User Guide for Arm*.
- If you use inline assembler to change the frame size or to perform function calls, this will not be reflected in the analysis.
- Extra space consumed by other sources (the processor, an operating system, etc) is not accounted for.
- Source code that uses exceptions is not supported.
- If you use other forms of function calls, like software interrupts, they will not be reflected in the call graph.
- Using multi-file compilation (`--mfc`) can interfere with using a stack usage control file to specify properties of module-local functions in the involved files.

Note: Stack usage analysis produces a worst case result. The program might not actually ever end up in the maximum call chain, by design, or by coincidence. In particular, the set of possible destinations for a virtual function call in C++ might sometimes include implementations of the function in question which cannot, in fact, be called from that point in the code.

Stack usage analysis is only a complement to actual measurement. If the result is important, you need to perform independent validation of the results of the analysis.

SITUATIONS WHERE WARNINGS ARE ISSUED

When stack usage analysis is enabled in the linker, warnings will be generated in the following circumstances:

- There is a function without stack usage information.
- There is an indirect call site in the application for which a list of possible called functions has not been supplied.
- There are no known indirect calls, but there is an uncalled function that is not known to be a call graph root.
- The application contains recursion (a cycle in the call graph) for which no maximum recursion depth has been supplied, or which is of a form for which the linker is unable to calculate a reliable estimate of stack usage.
- There are calls to a function declared as a call graph root.
- You have used the stack usage control file to supply stack usage information for functions in a module that does not have such information, and there are functions referenced by that module which have not been mentioned as being called in the stack usage control file.

CALL GRAPH LOG

To help you interpret the results of the stack usage analysis, there is a log output option that produces a simple text representation of the call graph (`--log call_graph`).

Example output:

```
Program entry:
0 __iar_program_start [168]
  0 __cmain [168]
    0 __iar_data_init3 [16]
      8 __iar_zero_init3 [8]
        16 - [0]
      8 __iar_copy_init3 [8]
        16 - [0]
    0 __low_level_init [0]
    0 main [168]
      8 printf [160]
        32 _PrintfTiny [136]
          88 _Prout [80]
            104 putchar [64]
              120 __write [48]
                120 __dwrite [48]
                  120 __iar_sh_stdout [48]
                    144 __iar_get_ttio [24]
                      168 __iar_lookup_ttioh [0]
                  120 __iar_sh_write [24]
                    144 - [0]
          88 __aeabi_uidiv [0]
            88 __aeabi_idiv0 [0]
          88 strlen [0]
    0 exit [8]
      0 _exit [8]
        0 __exit [8]
          0 __iar_close_ttio [8]
            8 __iar_lookup_ttioh [0] ***
    0 __exit [8] ***
```

Each line consists of this information:

- The stack usage at the point of call of the function
- The name of the function, or a single '-' to indicate usage in a function at a point with no function call (typically in a leaf function)
- The stack usage along the deepest call chain from that point. If no such value could be calculated, "[---]" is output instead. "***" marks functions that have already been shown.

CALL GRAPH XML OUTPUT

The linker can also produce a call graph file in XML format. This file contains one node for each function in your application, with the stack usage and call information relevant to that function. It is intended to be input for post-processing tools and is not particularly human-readable.

For more information about the XML format used, see the `callGraph.txt` file in your product installation.

Linking your application

- Linking considerations
- Hints for troubleshooting
- Checking module consistency
- Linker optimizations

Linking considerations

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

- *Choosing a linker configuration file*, page 113
- *Defining your own memory areas*, page 114
- *Placing sections*, page 115
- *Reserving space in RAM*, page 116
- *Keeping modules*, page 117
- *Keeping symbols and sections*, page 117
- *Application startup in 32-bit mode*, page 117
- *Application startup in 64-bit mode*, page 117
- *Setting up stack memory*, page 118
- *Setting up heap memory*, page 118
- *Setting up the atexit limit*, page 118
- *Changing the default initialization*, page 118
- *Interaction between ILINK and the application*, page 122
- *Standard library handling*, page 123
- *Producing output formats other than ELF/DWARF*, page 123
- *Veneers*, page 123

CHOOSING A LINKER CONFIGURATION FILE

The `config` directory contains ready-made templates for linker configuration files (`*.icf`) for all supported cores.

The files contain the information required by ILINK. The only change, if any, 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.

For some devices, device-specific configuration files are automatically selected.

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.

Do not 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. Note that all examples are for **32-bit mode** unless otherwise stated.

```
/* 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];
```

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:0x0 {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:

```
/* Place a variable in that section. */
const short MyVariable @ "MYOWNSECTION" = 0xF0F0;
```

This is the corresponding example in assembler language:

```
            name    createSection
            section MYOWNSECTION:CONST  ; Create a section,
                                        ; and fill it with
            dc16     0xF0F0             ; constant bytes.
            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:

```
/* Define a section for temporary storage. */
#pragma section = "TempStorage"
char *GetTempStorageStartAddress()
{
  /* 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 553.

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 96.

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 62.

APPLICATION STARTUP IN 32-BIT MODE

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 341.

APPLICATION STARTUP IN 64-BIT MODE

The point where the application starts execution is defined by the `__Reset_address` label (it determines where the `cstartup` module starts). The `__iar_program_start` label is placed at the same address. This label is also communicated via ELF to any debugger that is used.

For information about how to change the reset address, see *Reset address*, page 90.

SETTING UP STACK MEMORY

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{ };
```

Specify an appropriate size for your application. Note that **in 64-bit mode**, the stack alignment is 16.

For more information about the stack, see *Stack considerations*, page 218.

SETTING UP HEAP MEMORY

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. If you use a heap, you must allocate at least 50 bytes for it. Note that **in 64-bit mode**, the heap alignment is 16.

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:

- Suppressing initialization
- Choosing the packing algorithm
- Manual initialization
- Initializing code—copying ROM to RAM.

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

Suppressing initialization

If you do not want the linker to arrange for initialization by copying, for some or all sections, make sure that those sections do not match a pattern in an `initialize by copy` directive—or use an `except` clause to exclude them from matching. If you do not want any initialization by copying at all, you can omit the `initialize by copy` directive entirely.

This can be useful if your application, or just your variables, are loaded into RAM by some other mechanism before application startup.

Choosing a packing algorithm

To override the default packing algorithm, write for example:

```
initialize by copy with packing = lz77 { readwrite };
```

For more information about the available packing algorithms, see *initialize directive*, page 517.

Manual initialization

In the usual case, the `initialize by copy` directive is used for making the linker arrange for initialization by copying—with or without packing—of sections with content at application startup. The linker achieves this by logically creating an initialization section for each such section, holding the content of the section, and turning the original section into a section without content. Then, the linker adds table elements to the initialization table so that the initialization will be performed at application startup. You can use `initialize manually` to suppress the creation of table elements to take control over when and how the elements are copied. This is useful for overlays, but also in other circumstances.

For sections without content (zero-initialized sections), the situation is reversed. The linker arranges for zero initialization of all such sections at application startup, except for those that are mentioned in a `do not initialize` directive.

Simple copying example with an automatic block

Assume that you have some initialized variables in `MYSECTION`. If you add this directive to your linker configuration file:

```
initialize manually { section MYSECTION };
```

you can use this source code example to initialize the section:

```
#pragma section = "MYSECTION"
#pragma section = "MYSECTION_init"

void DoInit()
{
  char * from = __section_begin("MYSECTION_init");
  char * to   = __section_begin("MYSECTION");
  memcpy(to, from, __section_size("MYSECTION"));
}
```

This piece of source code takes advantage of the fact that if you use `__section_begin` (and related operators) with a section name, an automatic block is created by the linker for those sections.

Note: Automatic blocks override the normal section selection process and forces everything that matches the section name to form one block.

Example with explicit blocks

Assume that you instead of needing manual initialization for variables in a specific section, you need it for all initialized variables from a particular library. In that case, you must create explicit blocks for both the variables and the content. Like this:

```
initialize manually       { section .data      object mylib.a };
define block MYBLOCK      { section .data      object mylib.a };
define block MYBLOCK_init { section .data_init object mylib.a };
```

You must also place the two new blocks using one of the section placement directives, the block `MYBLOCK` in RAM and the block `MYBLOCK_init` in ROM.

Then you can initialize the sections using the same source code as in the previous example, only with `MYBLOCK` instead of `MYSECTION`.

Overlay example

This is a simple overlay example that takes advantage of automatic block creation:

```
initialize manually { section MYOVERLAY* };

define overlay MYOVERLAY { section MYOVERLAY1 };
define overlay MYOVERLAY { section MYOVERLAY2 };
```

You must also place `overlay MYOVERLAY` somewhere in RAM. The copying could look like this:

```
#pragma section = "MYOVERLAY"
#pragma section = "MYOVERLAY1_init"
#pragma section = "MYOVERLAY2_init"

void SwitchToOverlay1()
{
  char * from = __section_begin("MYOVERLAY1_init");
  char * to   = __section_begin("MYOVERLAY");
  memcpy(to, from, __section_size("MYOVERLAY1_init"));
}

void SwitchToOverlay2()
{
  char * from = __section_begin("MYOVERLAY2_init");
  char * to   = __section_begin("MYOVERLAY");
  memcpy(to, from, __section_size("MYOVERLAY2_init"));
}
```

Initializing code—copying ROM to RAM

Sometimes, an application copies pieces of code from flash/ROM to RAM. You can direct the linker to arrange for this to be done automatically at application startup, or do it yourself at some later time using the techniques described in *Manual initialization*, page 119.

You need to list the code sections that should be copied in an `initialize by copy` directive. The easiest way is usually to place the relevant functions in a particular section—for example, `RAMCODE`— and add `section RAMCODE` to your `initialize by copy` directive. For example:

```
initialize by copy { rw, section RAMCODE };
```

If you need to place the `RAMCODE` functions in some particular location, you must mention them in a placement directive, otherwise they will be placed together with other read/write sections.

If you need to control the manner and/or time of copying, you must use an `initialize manually` directive instead. See *Manual initialization*, page 119.

If the functions need to run without accessing the flash/ROM, you can use the `__ramfunc` keyword when compiling. See *Execution in RAM*, page 79.

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.

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 information, see *Interaction between the tools and your application*, page 221.

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 221.

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 *Prebuilt runtime libraries*, page 141.

PRODUCING OUTPUT FORMATS OTHER 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, see *The IAR ELF Tool—ielftool*, page 557.

VENEERS

Veneers are small sequences of code inserted by the linker to bridge the gap when a call instruction does not reach its destination or cannot switch to the correct mode.

Code for veneers can be inserted between any caller and called function. As a result, some registers must be treated as scratch registers at function calls, including functions written in assembler language. This applies to jumps as well. **In 32-bit mode**, `R12` must be treated as a scratch register. **In 64-bit mode**, both `X16` and `X17` must be treated as scratch registers.

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 348
- The `--map` option that makes ILINK produce a memory map file, which contains the result of the linker configuration file, see *--map*, page 350.

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 these 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`.
`Destination`	The target of the instruction, described with these details: • The instruction address, expressed both as a hexadecimal value and as a label with an offset. In this example, `0x9000000c` and "read"—therefore, 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 4: Description of a relocation error

Possible solutions

In this case, the distance from the instruction in `myfunc` 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.

Checking module consistency

This section introduces the concept of runtime model attributes, a mechanism used by the tools provided by IAR Systems to ensure that modules that are linked into an application are compatible, in other words, are built using compatible settings. The tools use a set of predefined runtime model attributes. In addition to these, you can define your own that you can use to ensure that incompatible modules are not used together.

Note: In addition to the predefined attributes, compatibility is also checked against the AEABI runtime attributes. These attributes deal mainly with object code compatibility, etc. They reflect compilation settings and are not user-configurable.

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 checks them in several ways.

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 5: 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, 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. This is how it could look like in one of the modules:

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

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

```
        rtmodel "uart", "mode1"
```

Note: Key names that start with two underscores are reserved by the compiler. For more information about the syntax, see *rtmodel*, page 420 and the *IAR Assembler User Guide for Arm*.

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.

Linker optimizations

This section contains information about:

- *Virtual function elimination*, page 127
- *Small function inlining*, page 127
- *Duplicate section merging*, page 127

VIRTUAL FUNCTION ELIMINATION

Virtual Function Elimination (VFE) is a linker optimization that removes unneeded virtual functions and dynamic runtime type information.

In order for Virtual Function Elimination to work, all relevant modules must provide information about virtual function table layout, which virtual functions are called, and for which classes dynamic runtime type information is needed. If one or more modules do not provide this information, a warning is generated by the linker and Virtual Function Elimination is not performed.

If you know that modules that lack such information do not perform any virtual function calls and do not define any virtual function tables, you can use the `--vfe=forced` linker option to enable Virtual Function Elimination anyway.

In the IDE, select **Project>Options>Linker>Optimizations>Perform C++ Virtual Function Elimination** to enable this optimization.

Note: You can disable Virtual Function Elimination entirely by using the `--no_vfe` linker option. In this case, no warning will be issued for modules that lack VFE information.

For more information, see *--vfe*, page 364 and *--no_vfe*, page 355.

SMALL FUNCTION INLINING

Small function inlining is a linker optimization that replaces some calls to small functions with the body of the function. This requires the body to fit in the space of the instruction that calls the function.

In the IDE, select **Project>Options>Linker>Optimizations>Inline small routines** to enable this optimization.

Use the linker option `--inline`.

DUPLICATE SECTION MERGING

The linker can detect read-only sections with identical contents and keep only one copy of each such section, redirecting all references to any of the duplicate sections to the retained section.

In the IDE, select **Project>Options>Linker>Optimizations>Merge duplicate sections** to enable this optimization.

Use the linker option `--merge_duplicate_sections`.

Note: This optimization can cause different functions or constants to have the same address, so if your application depends on the addresses being different, for example, by using the addresses as keys into a table, you should not enable this optimization.

The DLIB runtime environment

- Introduction to the runtime environment
- Setting up the runtime environment
- Additional information on the runtime environment
- Managing a multithreaded environment

Introduction to the runtime environment

A *runtime environment* is the environment in which your application executes.

This section contains information about:

- *Runtime environment functionality*, page 129
- *Briefly about input and output (I/O)*, page 130
- *Briefly about C-SPY emulated I/O*, page 131
- *Briefly about retargeting*, page 132

RUNTIME ENVIRONMENT FUNCTIONALITY

The *DLIB runtime environment* supports Standard C and C++ and consists of:

- The *C/C++ standard library*, both its interface (provided in the system header files) and its implementation.
- Startup and exit code.
- Low-level I/O interface for managing input and output (I/O).
- Special compiler support, for instance functions for switch handling or integer arithmetics.
- Support for hardware features:
  - Direct access to low-level processor operations by means of *intrinsic* functions, such as functions for interrupt mask handling
  - Peripheral unit registers and interrupt definitions in include files
  - The Vector Floating Point (VFP) coprocessor.

Runtime environment functions are provided in a *runtime library*.

The runtime library is delivered both as a prebuilt library and (depending on your product package) as source files. The prebuilt libraries are available in different *configurations* to meet various needs, see *Runtime library configurations*, page 140. You can find the libraries in the product subdirectories `arm\lib` and `arm\src\lib`, respectively.

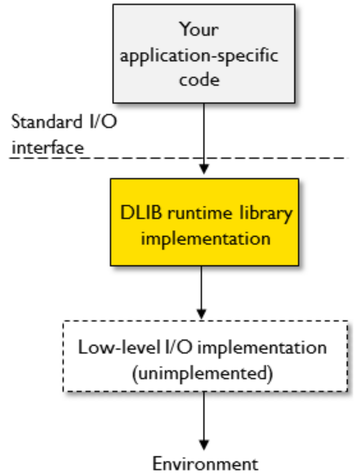
For more information about the library, see the chapter *C/C++ standard library functions*.

BRIEFLY ABOUT INPUT AND OUTPUT (I/O)

Every application must communicate with its environment. The application might for example display information on an LCD, read a value from a sensor, get the current date from the operating system, etc. Typically, your application performs I/O via the C/C++ standard library or some third-party library.

There are many functions in the C/C++ standard library that deal with I/O, including functions for: standard character streams, file system access, time and date, miscellaneous system actions, and termination and assert. This set of functions is referred to as the *standard I/O interface*.

On a desktop computer or a server, the operating system is expected to provide I/O functionality to the application via the standard I/O interface in the runtime environment. However, in an embedded system, the runtime library cannot assume that such functionality is present, or even that there is an operating system at all. Therefore, the low-level part of the standard I/O interface is not completely implemented by default:



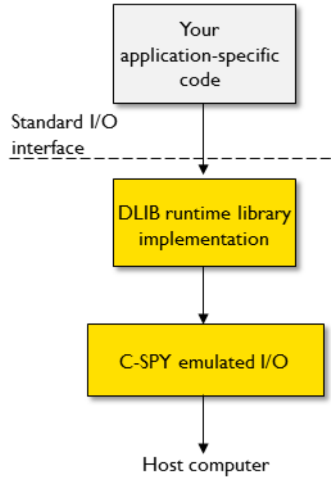
To make the standard I/O interface work, you can:

- Let the C-SPY debugger emulate I/O operations on the host computer, see *Briefly about C-SPY emulated I/O*, page 131
- *Retarget* the standard I/O interface to your target system by providing a suitable implementation of the interface, see *Briefly about retargeting*, page 132.

It is possible to mix these two approaches. You can, for example, let debug printouts and asserts be emulated by the C-SPY debugger, but implement your own file system. The debug printouts and asserts are useful during debugging, but no longer needed when running the application stand-alone (not connected to the C-SPY debugger).

BRIEFLY ABOUT C-SPY EMULATED I/O

C-SPY emulated I/O is a mechanism which lets the runtime environment interact with the C-SPY debugger to emulate I/O actions on the host computer:



For example, when C-SPY emulated I/O is enabled:

- Standard character streams are directed to the C-SPY **Terminal I/O** window
- File system operations are performed on the host computer
- Time and date functions return the time and date of the host computer
- The C-SPY debugger notifies when the application terminates or an assert fails.

This behavior can be valuable during the early development of an application, for example in an application that uses 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.

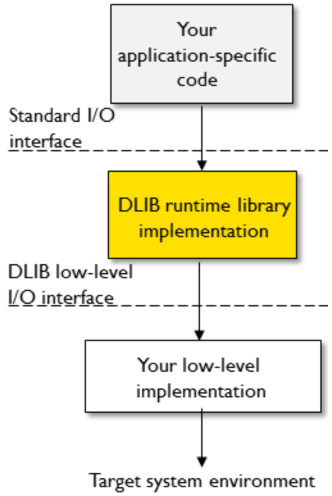
See *Setting up your runtime environment*, page 133 and *The semihosting mechanism*, page 148.

BRIEFLY ABOUT RETARGETING

Retargeting is the process where you adapt the runtime environment so that your application can execute I/O operations on your target system.

The standard I/O interface is large and complex. To make retargeting easier, the DLIB runtime environment is designed so that it performs all I/O operations through a small set of simple functions, which is referred to as the *DLIB low-level I/O interface*. By default, the functions in the low-level interface lack usable implementations. Some are unimplemented, others have stub implementations that do not perform anything except returning error codes.

To retarget the standard I/O interface, all you have to do is to provide implementations for the functions in the DLIB low-level I/O interface.



For example, if your application calls the functions `printf` and `fputc` in the standard I/O interface, the implementations of those functions both call the low-level function `__write` to output individual characters. To make them work, you just need to provide an implementation of the `__write` function—either by implementing it yourself, or by using a third-party implementation.

For information about how to override library modules with your own implementations, see *Overriding library modules*, page 137. See also *The DLIB low-level I/O interface*, page 154 for information about the functions that are part of the interface.

Setting up the runtime environment

This section contains these tasks:

- *Setting up your runtime environment*, page 133

  A runtime environment with basic project settings to be used during the initial phase of development.
- *Retargeting—Adapting for your target system*, page 135
- *Overriding library modules*, page 137
- *Customizing and building your own runtime library*, page 137

See also:

- *Managing a multithreaded environment*, page 165 for information about how to adapt the runtime environment to treat all library objects according to whether they are global or local to a thread.

SETTING UP YOUR RUNTIME ENVIRONMENT

You can set up the runtime environment based on some basic project settings. It is also often convenient to let the C-SPY debugger manage things like standard streams, file I/O, and various other system interactions. This basic runtime environment can be used for simulation before you have any target hardware.

To set up the runtime environment:

1. Before you build your project, choose **Project>Options>General Options** to open the **Options** dialog box.
2. On the **Library Configuration** page, verify the following settings:
   - **Library**: choose which *library configuration* to use. Typically, choose **None**, **Normal Full**, or **Custom**.

     For information about the various library configurations, see *Runtime library configurations*, page 140.

3 On the **Library Options** page, select **Auto with multibyte support** or **Auto without multibyte support** for both **Printf formatter** and **Scanf formatter**. This means that the linker will automatically choose the appropriate formatters based on information from the compiler. For more information about the available formatters and how to choose one manually, see *Formatters for printf*, page 145 and *Formatters for scanf*, page 147, respectively.

4 To enable C-SPY emulated I/O, choose **Project>Options>General Options>Library Configuration** and choose **Semihosted** (`--semihosted`) or **IAR breakpoint** (`--semihosting=iar_breakpoint`).

Note: For some Cortex-M devices it is also possible to direct `stdout/stderr` via SWO. This can significantly improve `stdout/stderr` performance compared to semihosting. For hardware requirements, see the *C-SPY® Debugging Guide for Arm*.

To enable `stdout` via SWO on the command line, use the linker option `--redirect __iar_sh_stdout=__iar_sh_stdout_swo`.

To enable `stdout` via SWO in the IDE, select the **Semihosted** option and the **stdout/stderr via SWO** option.

See *Briefly about C-SPY emulated I/O*, page 131 and *The semihosting mechanism*, page 148.

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

For this reason, a replacement for the `__write` function called `__write_buffered` is included in the runtime library. This module buffers the output and sends it to the debugger one line at a time, speeding up the output.

Note: This function uses about 80 bytes of RAM memory.

To use this feature in the IDE, choose **Project>Options>General Options>Library Options 1** and select the option **Buffered terminal output**.

To enable this function on the command line, add this to the linker command line:

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

6 Some math functions are available in different versions: default versions, smaller than the default versions, and larger but more accurate than default versions. Consider which versions you should use.

For more information, see *Math functions*, page 149.

7 When you build your project, a suitable prebuilt library and library configuration file are automatically used based on the project settings you made.

For information about which project settings affect the choice of library file, see *Runtime library configurations*, page 140.

You have now set up a runtime environment that can be used while developing your application source code.

RETARGETING—ADAPTING FOR YOUR TARGET SYSTEM

Before you can run your application on your target system, you must adapt some parts of the runtime environment, typically the system initialization and the DLIB low-level I/O interface functions.

To adapt your runtime environment for your target system:

1 Adapt system initialization.

It is likely that you must adapt the system initialization, for example, your application might need to initialize interrupt handling, I/O handling, watchdog timers, etc. You do this by implementing the routine `__low_level_init`, which is executed before the data sections are initialized. See *System startup and termination*, page 150 and *System initialization*, page 153.

Note: You can find device-specific examples on this in the example projects provided in the product installation, see the Information Center.

2 Adapt the runtime library for your target system. To implement such functions, you need a good understanding of the DLIB low-level I/O interface, see *Briefly about retargeting*, page 132.

Typically, you must implement your own functions if your application uses:

- Standard streams for input and output

  If any of these streams are used by your application, for example by the functions `printf` and `scanf`, you must implement your versions of the low-level functions `__read` and `__write`.

  The low-level functions identify I/O streams, such as an open file, with a file handle that is a unique integer. The I/O streams normally associated with `stdin`, `stdout`, and `stderr` have the file handles `0`, `1`, and `2`, respectively. When the handle is `-1`, all streams should be flushed. Streams are defined in `stdio.h`.

- File input and output

  The library contains a large number of powerful functions for file I/O operations, such as `fopen`, `fclose`, `fprintf`, `fputs`, etc. All these functions call a small set of low-level functions, each designed to accomplish one particular task, for example, `__open` opens a file, and `__write` outputs characters. Implement your version of these low-level functions.

- `signal` and `raise`

  If the default implementation of these functions does not provide the functionality you need, you can implement your own versions.
- Time and date

  To make the time and date functions work, you must implement the functions `clock`, `__time32`, `__time64`, and `__getzone`. Whether you use `__time32` or `__time64` depends on which interface you use for `time_t`, see *time.h*, page 499.
- Assert, see *__aeabi_assert*, page 155.
- Environment interaction

  If the default implementation of `system` or `getenv` does not provide the functionality you need, you can implement your own versions.

For more information about the functions, see *The DLIB low-level I/O interface*, page 154.

The library files that you can override with your own versions are located in the `arm\src\lib` directory.

3 When you have implemented your functions of the low-level I/O interface, you must add your version of these functions to your project. For information about this, see *Overriding library modules*, page 137.

Note: If you have implemented a DLIB low-level I/O interface function and added it to a project that you have built with support for C-SPY emulated I/O, your low-level function will be used and not the functions provided with C-SPY emulated I/O. For example, if you implement your own version of `__write`, output to the C-SPY **Terminal I/O** window will not be supported. See *Briefly about C-SPY emulated I/O*, page 131.

4 Before you can execute your application on your target system, you must rebuild your project with a Release build configuration. This means that the linker will not include the C-SPY emulated I/O mechanism and the low-level I/O functions it provides. If your application calls any of the low-level functions of the standard I/O interface, either directly or indirectly, and your project does not contain these, the linker will issue an error for every missing low-level function.

Note: By default, the `NDEBUG` symbol is defined in a Release build configuration, which means asserts will no longer be checked. For more information, see *__aeabi_assert*, page 155.

OVERRIDING LIBRARY MODULES

To override a library function and replace it with your own implementation:

1 Use a template source file—a library source file or another template—and place a copy of it in your project directory.

The library files that you can override with your own versions are located in the `arm\src\lib` directory.

2 Modify the file.

Note: To override the functions in a module, you must provide alternative implementations for all the needed symbols in the overridden module. Otherwise you will get error messages about duplicate definitions.

3 Add the modified file to your project, like any other source file.

Note: If you have implemented a DLIB low-level I/O interface function and added it to a project that you have built with support for C-SPY emulated I/O, your low-level function will be used and not the functions provided with C-SPY emulated I/O. For example, if you implement your own version of `__write`, output to the C-SPY **Terminal I/O** window will not be supported. See *Briefly about C-SPY emulated I/O*, page 131.

You have now finished the process of overriding the library module with your version.

CUSTOMIZING AND BUILDING YOUR OWN RUNTIME LIBRARY

If the prebuilt library configurations do not meet your requirements, you can customize your own library configuration, but that requires that you *rebuild* relevant parts of the library.

Note: Customizing and building your own runtime library requires access to the library source code, which is not available for all types of IAR Embedded Workbench licenses.

Building a customized library is a complex process. Therefore, consider carefully whether it is really necessary. You must build your own runtime library when:

- You want to define your own library configuration with support for locale, file descriptors, multibyte characters, etc. This will include or exclude certain parts of the DLIB runtime environment.

In those cases, you must:

- Make sure that you have installed the library source code (`src\lib`). If not already installed, you can install it using the IAR License Manager, see the *Licensing Guide*.

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

  Note that the customized library only replaces the part of the DLIB runtime environment implemented in the libraries for C and C++ library functions. Rebuilding libraries for the following is not supported:

  - math functions
  - runtime support functions
  - thread support functions
  - timezone and daylight saving time functions
  - debug support functions

To set up a library project:

1 In the IDE, choose **Project>Create New Project** and use the library project template which can be used for customizing the runtime environment configuration. There is a library template for the Full library configuration, see *Runtime library configurations*, page 140

Note: When you create a new library project from a template, the majority of the files included in the new project are the original installation files. If you are going to modify these files, make copies of them first and replace the original files in the project with these copies.

To customize the library functionality:

1 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` which you can find in `arm\inc\c`. This read-only file describes the configuration possibilities. Note that you should not modify this file.

In addition, you can create your own *library configuration file* by making a copy of the file `DLib_Config_`*`configuration`*`.h`—which you can find in the directory—and customize it by setting the values of the configuration symbols according to the application requirements.

For information about configuration symbols that you might want to customize, see:

- *Configuration symbols for file input and output*, page 164
- *Locale*, page 164
- *Managing a multithreaded environment*, page 165

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

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

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 *IDE Project Management and Building Guide for Arm*.

To use the customized runtime library in your application project:

1. In the IDE, choose **Project>Options>General Options** and click the **Library Configuration** tab.
2. From the **Library** drop-down menu, choose **Custom**.
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.

Additional information on the runtime environment

This section gives additional information on the runtime environment:

- *Bounds checking functionality*, page 139
- *Runtime library configurations*, page 140
- *Prebuilt runtime libraries*, page 141
- *Formatters for printf*, page 145
- *Formatters for scanf*, page 147
- *The C-SPY emulated I/O mechanism*, page 148
- *The semihosting mechanism*, page 148
- *Math functions*, page 149
- *System startup and termination*, page 150
- *System initialization*, page 153
- *The DLIB low-level I/O interface*, page 154
- *Configuration symbols for file input and output*, page 164
- *Locale*, page 164

BOUNDS CHECKING FUNCTIONALITY

To enable the bounds checking functions specified in Annex K (*Bounds-checking interfaces*) of the C standard, define the preprocessor symbol

`__STDC_WANT_LIB_EXT1__` to `1` prior to including any system headers. See *C bounds-checking interface*, page 498.

RUNTIME LIBRARY CONFIGURATIONS

The runtime library is provided with different *library configurations*, where each configuration is suitable for different application requirements.

The runtime library configuration is defined in the *library configuration file*. It contains information about what functionality is part of the runtime environment. The less functionality you need in the runtime environment, the smaller the environment becomes.

These predefined library configurations are available:

Library configuration	Description
Normal DLIB (default)	C locale, but no locale interface, no file descriptor support, no multibyte characters in `printf` and `scanf`.
Full DLIB	Full locale interface, C locale, file descriptor support, and optionally multibyte characters in `printf` and `scanf`.

Table 6: Library configurations

Note: In addition to these predefined library configurations, you can provide your own configuration, see *Customizing and building your own runtime library*, page 137

If you do not specify a library configuration explicitly you will get the default configuration. If you use a prebuilt runtime library, a configuration file that matches the runtime library file will automatically be used. See *Setting up the runtime environment*, page 133.

To override the default library configuration, use one of these methods:

1. Use a prebuilt configuration of your choice—to specify a runtime configuration explicitly:

   Choose **Project>Options>General Options>Library Configuration>Library** and change the default setting.

   Use the `--dlib_config` compiler option, see *--dlib_config*, page 291.

   The prebuilt libraries are based on the default configurations, see *Runtime library configurations*, page 140.

2. If you have built your own customized library, choose **Project>Options>General Options>Library Configuration>Library** and choose **Custom** to use your own configuration. For more information, see *Customizing and building your own runtime library*, page 137.

PREBUILT RUNTIME LIBRARIES

The prebuilt runtime libraries are configured for different combinations of these options:

- Processor variant
- Data model
- Library configuration—Normal or Full.

The linker will automatically include the correct library files and library configuration file. To explicitly specify a library configuration, use the `--dlib_config` compiler option.

Library filename syntax

The names of the libraries are constructed from these elements:

arch — Specifies the CPU architecture:

`4t` = Armv4T
`5E` = Armv5E
`6M` or `6Mx` = Armv6M (6Mx is built with `--no_literal_pool`)
`7M` or `7Mx` = Armv7M (7Mx is built with `--no_literal_pool`)
`7Sx` = Armv7-A and Armv7-R, built with `--no_literal_pool`
`4as` = Generic Armv4, built with bounds-checking
`7as` = Generic Armv7, built with bounds-checking
`8A` = Armv8-A

mode — Specifies the default processor/execution mode:

`a` = Arm mode
`t` = Thumb mode
`x` = 64-bit mode

endian — Specifies the byte order:

`l` = little-endian
`b` = big-endian

lib-config — Specifies the library configuration:

`n` = Normal
`f` = Full

rwpi — Specifies whether the library supports RWPI:

`s` = RWPI supported
not present = no RWPI support

fp — Specifies how floating-point operations are implemented:

`v` = VFP
`s` = VFP for single precision only
not present = software implementation

abi	Specifies the data model **in 64-bit mode**: 44 = ILP32 (4-byte long, 4-byte pointers) 88 = LP64 (8-byte long, 8-byte pointers) not present = the library is for use **in 32-bit mode**
debug-interface	Specifies a semihosting mechanism: s = SVC b = BKPT i = IAR-breakpoint

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

Groups of library files

The libraries are delivered in groups of library functions:

Library files for C library functions

These are the functions defined by Standard C, for example, functions like `printf` and `scanf`. Note that this library does not include math functions.

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

dl{*arch*}_{*mode*}{*endian*}{*lib-config*}[*rwpi*][*abi*].a

which more specifically means

```
dl{4t|5E|6M|6Mx|7M|7Mx|7Sx|8A}_{a|t|x}{l|b}{n|f}[s][44|88].a
```

Library files for C++ library functions

These are the functions defined by C++, compiled with support for Standard C++.

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

dlpp{*arch*}_{*mode*}{*endian*}_{*lib-config*}c[rwpi][*abi*].a

which more specifically means

```
dlpp{4t|5E|6M|6Mx|7M|7Mx|7Sx|4as|7as|8A}_{a|t|x}{l|b}_{n|f}c
[s][44|88].a
```

Library files for C++ runtime library functions

These are the runtime functions defined by C++, compiled with support for Standard C++.

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

`dlpprt{arch}_{mode}{endian}_{lib-config}c[rwpi][abi].a`

which more specifically means

`dlpprt{4t|5E|6M|6Mx|7M|7Mx|7Sx|4as|7as|8A}_{a|t|x}{l|b}_{n|f}c[s][44|88].a`

Library files for math functions

These are the functions for floating-point arithmetic and functions with a floating-point type in its signature as defined by Standard C, for example, functions like `sqrt`.

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

`m{arch}_{mode}{endian}[fp][abi].a`

which more specifically means

`m{4t|5E|6M|6Mx|7M|7Mx|7Sx|8A}_{a|t|x}{l|b}[v|s][44|88].a`

Library files for thread support functions

These are the functions for thread support.

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

`th{arch}_{mode}{endian}{lib-config}[abi].a`

which more specifically means

`th{4t|5E|6M|6Mx|7M|7Mx|7Sx|8A}_{a|t|x}{l|b}{n|f}[44|88].a`

Library files for timezone and daylight saving time support functions

These are the functions with support for timezone and daylight saving time functionality.

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

`tz{arch}_{mode}{endian}[rwpi][abi].a`

which more specifically means

`tz{4t|5E|6M|6Mx|7M|7Mx|7Sx|8A}_{a|t|x}{l|b}[s][44|88].a`

Library files for runtime support functions

These are functions for system startup, initialization, non floating-point AEABI support routines, and some of the functions that are part of Standard C and C++.

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

`rt{arch}_{mode}{endian}[abi].a`

which more specifically means

`rt{4t|5E|6M|6Mx|7M|7Mx|7Sx|8A}_{a|t|x}{l|b}[44|88].a`

Library files for debug support functions

These are functions for debug support for the semihosting interface. The names of the library files are constructed in the following way:

`sh{debug-interface}_{endian}.a`

or

`sh{arch}_{endian}[abi].a`

which more specifically means

`sh{s|b|i}_{l|b}.a`

or

`sh{6Mx|7Mx|7Sx|8A}_{l|b}[44|88].a`

FORMATTERS FOR PRINTF

The `printf` function uses a formatter called `_Printf`. The full 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. Note that the `wprintf` variants are not affected.

This table summarizes the capabilities of the different formatters:

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

Table 7: Formatters for printf

† `NoMb` means without multibytes.

The compiler can automatically detect which formatting capabilities are needed in a direct call to `printf`, if the formatting string is a string literal. This information is passed to the linker, which combines the information from all modules to select a suitable formatter for the application. However, if the formatting string is a variable, or if the call is indirect through a function pointer, the compiler cannot perform the analysis, forcing the linker to select the Full formatter. In this case, you might want to override the automatically selected `printf` formatter.

1 Choose **Project>Options>General Options** to open the **Options** dialog box.

2 On the **Library Options** page, select the appropriate formatter.

To override the automatically selected printf formatter from the command line:

1 Use one of these ILINK command line options:

```
--redirect _Printf=_PrintfFull
--redirect _Printf=_PrintfFullNoMb
--redirect _Printf=_PrintfLarge
--redirect _Printf=_PrintfLargeNoMb
--redirect _Printf=_PrintfSmall
--redirect _Printf=_PrintfSmallNoMb
--redirect _Printf=_PrintfTiny
--redirect _Printf=_PrintfTinyNoMb
```

If the compiler does not recognize multibyte support, you can enable it:

Select **Project>Options>General Options>Library Options 1>Enable multibyte support**.

Use the linker option `--printf_multibytes`.

FORMATTERS FOR SCANF

In a similar way to the `printf` function, `scanf` uses a common formatter, called `_Scanf`. The full version is quite large, and provides facilities that are not required in many embedded applications. To reduce the memory consumption, two smaller, alternative versions are also provided. Note that the `wscanf` versions are not affected.

This table summarizes the capabilities of the different formatters:

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

Table 8: Formatters for scanf

† `NoMb` means without multibytes.

The compiler can automatically detect which formatting capabilities are needed in a direct call to `scanf`, if the formatting string is a string literal. This information is passed to the linker, which combines the information from all modules to select a suitable formatter for the application. However, if the formatting string is a variable, or if the call is indirect through a function pointer, the compiler cannot perform the analysis, forcing the linker to select the full formatter. In this case, you might want to override the automatically selected `scanf` formatter.

To manually specify the scanf formatter in the IDE:

1. Choose **Project>Options>General Options** to open the **Options** dialog box.
2. On the **Library Options** page, select the appropriate formatter.

To manually specify the scanf formatter from the command line:

Use one of these ILINK command line options:

```
--redirect _Scanf=_ScanfFull
--redirect _Scanf=_ScanfFullNoMb
--redirect _Scanf=_ScanfLarge
--redirect _Scanf=_ScanfLargeNoMb
--redirect _Scanf=_ScanfSmall
--redirect _Scanf=_ScanfSmallNoMb
```

If the compiler does not recognize multibyte support, you can enable it:

Select **Project>Options>General Options>Library Options 1>Enable multibyte support**.

Use the linker option `--scanf_multibytes`.

THE C-SPY EMULATED I/O MECHANISM

1 The debugger will detect the presence of the function `__DebugBreak`, which will be part of the application if you linked it with the linker option for C-SPY emulated I/O.

2 In this case, the debugger will automatically set a breakpoint at the `__DebugBreak` function.

3 When your application calls a function in the DLIB low-level I/O interface, for example, `open`, the `__DebugBreak` function is called, which will cause the application to stop at the breakpoint and perform the necessary services.

4 The execution will then resume.

See also *Briefly about C-SPY emulated I/O*, page 131.

THE SEMIHOSTING MECHANISM

C-SPY emulated I/O is compatible with the semihosting interface provided by Arm Limited. When an application invokes a semihosting call, the execution stops at a debugger breakpoint. The debugger then handles the call, performs any necessary actions on the host computer and then resumes the execution.

There are three variants of semihosting mechanisms available:

- For Cortex-M, the interface uses `BKPT` instructions to perform semihosting calls
- For other Arm cores, `SVC` instructions are used for the semihosting calls **in 32-bit mode**
- **In 64-bit mode**, `HLT` instructions are used for the semihosting calls
- *IAR breakpoint*, which is an IAR-specific alternative to semihosting that uses SVC.

To support semihosting via SVC, the debugger must set its semihosting breakpoint on the Supervisor Call vector to catch SVC calls. If your application uses SVC calls for other purposes than semihosting, the handling of this breakpoint will cause a severe performance penalty for each such call. IAR breakpoint is a way to get around this. By using a special function call instead of an SVC instruction to perform semihosting, the semihosting breakpoint can be set on that special function instead. This means that semihosting will not interfere with other uses of the Supervisor Call vector.

Note: IAR breakpoint is an IAR-specific extension of the semihosting standard. If you link your application with libraries built with toolchains from other vendors than IAR Systems and use IAR breakpoint, semihosting calls made from code in those libraries will not work.

MATH FUNCTIONS

Some C/C++ standard library math functions are available in different versions:

- The default versions
- Smaller versions (but less accurate)
- More accurate versions (but larger).

Smaller versions

The functions `cos`, `exp`, `log`, `log2`, `log10`, `pow`, `sin`, and `tan` exist in additional, smaller versions in the library. They are about 20% smaller and about 20% faster than the default versions. The functions handle INF and NaN values. The drawbacks are that they almost always lose some precision and they do not have the same input range as the default versions.

The names of the functions are constructed like:

```
__iar_xxx_small<f|l>
```

where `f` is used for `float` variants, `l` is used for `long double` variants, and no suffix is used for `double` variants.

More accurate versions

The functions `cos`, `pow`, `sin`, and `tan` exist in versions in the library that are more exact and can handle larger argument ranges. The drawback is that they are larger and slower than the default versions.

The names of the functions are constructed like:

```
__iar_xxx_accurate<f|l>
```

where `f` is used for `float` variants, `l` is used for `long double` variants, and no suffix is used for `double` variants.

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` located in the `arm\src\lib\arm`, `arm\src\lib\thumb` (for Cortex-M), or `arm\src\lib\a64` directory, and `low_level_init.c` located in the `arm\src\lib\runtime` directory.

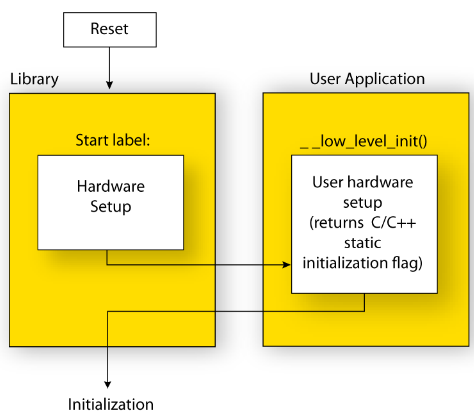
Note: To install some of these files, you must extract the IAR Library Source package.

For information about how to customize the system startup code, see *System initialization*, page 153.

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.

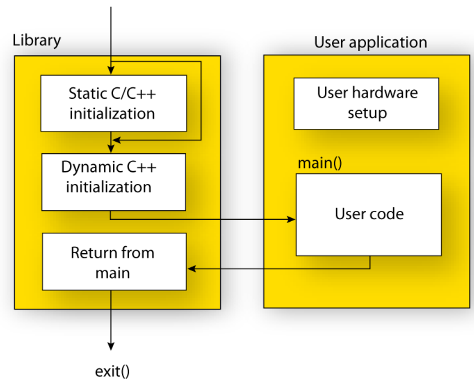
For the hardware initialization, it looks like this:



- When the CPU is reset it will start executing at the program entry label `__iar_program_start` in the system startup code. Note that **in 64-bit mode**, it is setup by the linker symbol `__Reset_address`.
- **In 64-bit mode**, the program is traversed from the exception level entered at hard reset to exception level 1.
- The stack pointer is initialized to the end of the `CSTACK` block
- For Arm7/9/11, Cortex-A, and Cortex-R devices, exception stack pointers are initialized to the end of each corresponding section

- The function `__low_level_init` is called if you defined it, giving the application a chance to perform early initializations.

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

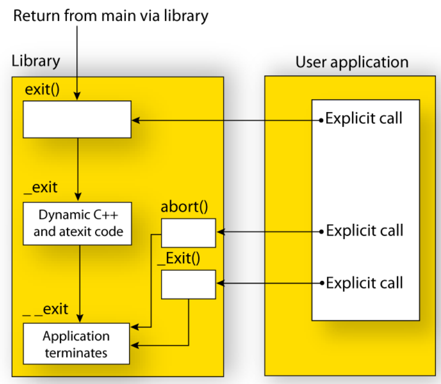


- Static and global variables are initialized. That is, zero-initialized variables are cleared and the values of other initialized variables are copied from ROM to RAM memory. This step is skipped if `__low_level_init` returns zero. For more information, see *Initialization at system startup*, page 102.
- Static C++ objects are constructed
- The `main` function is called, which starts the application.

For information about the initialization phase, see *Application execution—an overview*, page 64.

System termination

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



An application can terminate normally in two different ways:

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

Because the 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:

- 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 function `atexit`. See also *Setting up the atexit limit*, page 118.
- Close all open files
- Call `__exit`
- When `__exit` is reached, stop the system.

An application can also exit by calling the `abort`, the `_Exit`, or the `quick_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. The `quick_exit` function is equivalent to the `_Exit` function, except that it calls each function passed to `at_quick_exit` before calling `__exit`.

If you want your application to do anything extra at exit, for example, resetting the system (and if using `atexit` is not sufficient), you can write your own implementation of the `__exit(int)` function.

The library files that you can override with your own versions are located in the `arm\src\lib` directory. See *Overriding library modules*, page 137.

C-SPY debugging support for system termination

If you have enabled C-SPY emulated I/O during linking, the normal `__exit` function is replaced with a special one. C-SPY will then recognize when this function is called and can take appropriate actions to emulate program termination. For more information, see *Briefly about C-SPY emulated I/O*, page 131.

SYSTEM INITIALIZATION

It is likely that you need to adapt the 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 the system startup code.

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

The code for handling system startup is located in the source files `cstartup.s` and `low_level_init.c`. `cstartup.s` is located in the `arm\src\lib\arm`, `arm\src\lib\thumb` (for Cortex-M), or `arm\src\lib\a64` directory, and `low_level_init.c` is located in the `arm\src\lib\runtime` directory.

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

Note: Regardless of whether you implement your own version of `__low_level_init` or the file `cstartup.s`, you do not have to rebuild the library.

Customizing __low_level_init

A skeleton low-level initialization file is supplied with the product: `low_level_init.c`.

Note: Static initialized variables cannot be used within the file, because 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 cstartup file

As noted earlier, you should not modify the `cstartup.s` file if implementing your own version of `__low_level_init` is enough for your needs. However, if you do need to modify the `cstartup.s` file, we recommend that you follow the general procedure for creating a modified copy of the file and adding it to your project, see *Overriding library modules*, page 137.

Note: 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 341.

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

THE DLIB LOW-LEVEL I/O INTERFACE

The runtime library uses a set of low-level functions—which are referred to as the *DLIB low-level I/O interface*—to communicate with the target system. Most of the low-level functions have no implementation.

For more information, see *Briefly about input and output (I/O)*, page 130.

These are the functions in the DLIB low-level I/O interface:

`abort`

`__aeabi_assert`

`clock`

`__close`

`__exit`

`getenv`

`__getzone`

`__lseek`

`__open`

`raise`

`__read`

`remove`

`rename`

`signal`

`system`

`__time32, __time64`

`__write`

Note: You should normally not use the low-level functions prefixed with __ directly in your application. Instead you should use the standard library functions that use these functions. For example, to write to `stdout`, you should use standard library functions like `printf` or `puts`, which in turn calls the low-level function `__write`. If you have forgot to implement a low-level function and your application calls that function via a standard library function, the linker issues an error when you link in release build configuration.

Note: If you implement your own variants of the functions in this interface, your variants will be used even though you have enabled C-SPY emulated I/O, see *Briefly about C-SPY emulated I/O*, page 131.

abort

Source file	`arm\src\lib\runtime\abort.c`
Declared in	`stdlib.h`
Description	Standard C library function that aborts execution.
C-SPY debug action	Exits the application.
Default implementation	Calls `__exit(EXIT_FAILURE)`.
See also	*Briefly about retargeting*, page 132 *System termination*, page 152.

__aeabi_assert

Source file	`arm\src\lib\runtime\assert.c`
Declared in	`assert.h`
Description	Low-level function that handles a failed assert.

C-SPY debug action
Notifies the C-SPY debugger about the failed assert.

Default implementation
Failed asserts are reported by the function `__aeabi_assert`. By default, it prints an error message and calls `abort`. If this is not the behavior you require, you can implement your own version of the function.

The assert macro is defined in the header file `assert.h`. To turn off assertions, define the symbol `NDEBUG`.

In the IDE, the symbol `NDEBUG` is by default defined in a Release project and *not* defined in a Debug project. If you build from the command line, you must explicitly define the symbol according to your needs. See *NDEBUG*, page 487.

See also
Briefly about retargeting, page 132.

clock

Source file
`arm\src\lib\time\clock.c`

Declared in
`time.h`

Description
Standard C library function that accesses the processor time.

C-SPY debug action
Returns the clock on the host computer.

Default implementation
Returns `-1` to indicate that processor time is not available.

See also
Briefly about retargeting, page 132.

__close

Source file
`arm\src\lib\file\close.c`

Declared in
`LowLevelIOInterface.h`

Description
Low-level function that closes a file.

C-SPY debug action
Closes the associated host file on the host computer.

Default implementation
None.

See also
Briefly about retargeting, page 132.

__exit

Source file `arm\src\lib\runtime\xxexit.c`

Declared in `LowLevelIOInterface.h`

Description Low-level function that halts execution.

C-SPY debug action Notifies that the end of the application was reached.

Default implementation Loops forever.

See also *Briefly about retargeting*, page 132

System termination, page 152.

getenv

Source file `arm\src\lib\runtime\getenv.c`
`arm\src\lib\runtime\environ.c`

Declared in `Stdlib.h` and `LowLevelIOInterface.h`

C-SPY debug action Accesses the host environment.

Default implementation The `getenv` function in the library 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.

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

See also *Briefly about retargeting*, page 132.

__getzone

Source file `arm\src\lib\time\getzone.c`

Declared in `LowLevelIOInterface.h`

Description Low-level function that returns the current time zone.

Note: You must enable the time zone functionality in the library by using the linker option `--timezone_lib`.

C-SPY debug action Not applicable.

Default implementation Returns `":"`.

See also *Briefly about retargeting*, page 132 and *--timezone_lib*, page 362.

For more information, see the source file `getzone.c`.

__lseek

Source file `arm\src\lib\file\lseek.c`

Declared in `LowLevelIOInterface.h`

Description Low-level function for changing the location of the next access in an open file.

C-SPY debug action Searches in the associated host file on the host computer.

Default implementation None.

See also *Briefly about retargeting*, page 132.

__open

Source file `arm\src\lib\file\open.c`

Declared in `LowLevelIOInterface.h`

Description Low-level function that opens a file.

C-SPY debug action	Opens a file on the host computer.
Default implementation	None.
See also	*Briefly about retargeting*, page 132.

raise

Source file	`arm\src\lib\runtime\raise.c`
Declared in	`signal.h`
Description	Standard C library function that raises a signal.
C-SPY debug action	Not applicable.
Default implementation	Calls the signal handler for the raised signal, or terminates with call to `__exit(EXIT_FAILURE)`.
See also	*Briefly about retargeting*, page 132.

__read

Source file	`arm\src\lib\file\read.c`
Declared in	`LowLevelIOInterface.h`
Description	Low-level function that reads characters from `stdin` and from files.
C-SPY debug action	Directs `stdin` to the **Terminal I/O** window. All other files will read the associated host file.
Default implementation	None.

Example

The code in this example uses memory-mapped I/O to read from a keyboard, whose port is assumed to be located at `0x1000`:

```
#include <stddef.h>
#include <LowLevelIOInterface.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 handles associated with the streams, see *Retargeting—Adapting for your target system*, page 135.

For information about the `@` operator, see *Controlling data and function placement in memory*, page 242.

See also

Briefly about retargeting, page 132.

remove

Source file: `arm\src\lib\file\remove.c`

Declared in: `stdio.h`

Description: Standard C library function that removes a file.

C-SPY debug action: Removes a file on the host computer.

Default implementation: Returns `0` to indicate success, but without removing a file.

See also: *Briefly about retargeting*, page 132.

rename

Source file: `arm\src\lib\file\rename.c`

Declared in: `stdio.h`

Description: Standard C library function that renames a file.

C-SPY debug action: Renames a file on the host computer.

Default implementation: Returns `-1` to indicate failure.

See also: *Briefly about retargeting*, page 132.

signal

Source file: `arm\src\lib\runtime\signal.c`

Declared in: `signal.h`

Description: Standard C library function that changes signal handlers.

C-SPY debug action: Not applicable.

Default implementation: As specified by Standard C. You might want to modify this behavior if the environment supports some kind of asynchronous signals.

See also: *Briefly about retargeting*, page 132.

system

Source file	`arm\src\lib\runtime\system.c`
Declared in	`stdlib.h`
Description	Standard C library function that executes commands.
C-SPY debug action	Notifies the C-SPY debugger that `system` has been called and then returns `-1`.
Default implementation	The `system` function available in the library returns `0` if a null pointer is passed to it to indicate that there is no command processor, otherwise it returns `-1` to indicate failure. If this is not the functionality that you require, you can implement your own version. This does not require that you rebuild the library.
See also	*Briefly about retargeting*, page 132.

__time32, __time64

Source file	`arm\src\lib\time\time.c` `arm\src\lib\time\time64.c`
Declared in	`time.h`
Description	Low-level functions that return the current calendar time.
C-SPY debug action	Returns the time on the host computer.
Default implementation	Returns `-1` to indicate that calendar time is not available.
See also	*Briefly about retargeting*, page 132.

__write

Source file	`arm\src\lib\file\write.c`
Declared in	`LowLevelIOInterface.h`
Description	Low-level function that writes to `stdout`, `stderr`, or a file.
C-SPY debug action	Directs `stdout` and `stderr` to the **Terminal I/O** window. All other files will write to the associated host file.

Default implementation None.

Example The code in this example uses memory-mapped I/O to write to an LCD display, whose port is assumed to be located at address `0x1000`:

```
#include <stddef.h>
#include <LowLevelIOInterface.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;
}
```

For information about the handles associated with the streams, see *Retargeting—Adapting for your target system*, page 135.

See also *Briefly about retargeting*, page 132.

CONFIGURATION SYMBOLS FOR FILE INPUT AND OUTPUT

File I/O is only supported by libraries with the Full library configuration, see *Runtime library configurations*, page 140, or in a customized library when the configuration symbol `__DLIB_FILE_DESCRIPTOR` is defined. If this symbol is not defined, functions taking a `FILE *` argument cannot be used.

To customize your library and rebuild it, see *Customizing and building your own runtime library*, page 137.

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 which library configuration you are using, you get different levels 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. See *Runtime library configurations*, page 140.

The DLIB runtime library can be used in two main modes:

- Using a full library configuration that has a locale interface, which makes it possible to switch between different locales during runtime

  The application starts with the C locale. To use another locale, you must call the `setlocale` function or use the corresponding mechanisms in C++. The locales that the application can use are set up at linkage.
- Using a normal library configuration that does not have a locale interface, where the C locale is hardwired into the application.

Note: If multibytes are to be printed, you must make sure that the implementation of `__write` in the DLIB low-level I/O interface can handle them.

Specifying which locales that should be available in your application

Choose **Project>Options>General Options>Library Options 2>Locale support**.

Use the linker option `--keep` with the tag of the locale as the parameter, for example:

```
--keep _Locale_cs_CZ_iso8859_2
```

The available locales are listed in the file `SupportedLocales.json` in the `arm\config` directory, for example:

```
['Czech language locale for Czech Republic', 'iso8859-2',
'cs_CZ.iso8859-2', '_Locale_cs_CZ_iso8859_2'],
```

The line contains the full locale name, the encoding for the locale, the abbreviated locale name, and the tag to be used as parameter to the linker option `--keep`.

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 available encodings are ISO-8859-1, ISO-8859-2, ISO-8859-4, ISO-8859-5, ISO-8859-7, ISO-8859-8, ISO-8859-9, ISO-8859-15, CP932, and UTF-8.

For a complete list of the available locales and their respective encoding, see the file `SupportedLocales.json` in the `arm\config` directory.

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");
```

Managing a multithreaded environment

This section contains information about:

In a multithreaded environment, the standard library must treat all library objects according to whether they are global or local to a thread. If an object is a true global object, any updates of its state must be guarded by a locking mechanism to make sure that only one thread can update it at any given time. If an object is local to a thread, the static variables containing the object state must reside in a variable area local to that thread. This area is commonly named *thread-local storage* (TLS).

The low-level implementations of locks and TLS are system-specific, and is not included in the DLIB runtime environment. If you are using an RTOS, check if it provides some or all of the required functions. Otherwise, you must provide your own.

MULTITHREAD SUPPORT IN THE DLIB RUNTIME ENVIRONMENT

The DLIB runtime environment uses two kinds of locks—*system locks* and *file stream locks*. The file stream locks are used as guards when the state of a file stream is updated, and are only needed in the Full library configuration. The following objects are guarded with system locks:

- The heap (in other words when `malloc`, `new`, `free`, `delete`, `realloc`, or `calloc` is used).
- The C file system (only available in the Full library configuration), but not the file streams themselves. The file system is updated when a stream is opened or closed, in other words when `fopen`, `fclose`, `fdopen`, `fflush`, or `freopen` is used.
- The signal system (in other words when `signal` is used).
- The temporary file system (in other words when `tmpnam` is used).
- C++ dynamically initialized function-local objects with static storage duration.
- C++ locale facet handling
- C++ regular expression handling
- C++ terminate and unexpected handling

These library objects use TLS:

Library objects using TLS	When these functions are used
Error functions	`errno`, `strerror`
C++ exception engine	Not applicable

Table 9: Library objects using TLS

Note: If you are using `printf`/`scanf` (or any variants) with formatters, each individual formatter will be guarded, but the complete `printf`/`scanf` invocation will not be guarded.

If C++ is used in a runtime environment with multithread support, the compiler option `--guard_calls` must be used to make sure that function-static variables with dynamic initializers are not initialized simultaneously by several threads.

ENABLING MULTITHREAD SUPPORT

To configure multithread support for use with threaded applications:

1 To enable multithread support:

On the command line, use the linker option `--threaded_lib`.

If C++ is used, the compiler option `--guard_calls` should be used as well to make sure that function-static variables with dynamic initializers are not initialized simultaneously by several threads.

In the IDE, choose **Project>Options>General Options>Library Configuration>Enable thread support in the library**. This will invoke the linker option `--threaded_lib` and if C++ is used, the IDE will automatically use the compiler option `--guard_calls` to make sure that function-static variables with dynamic initializers are not initialized simultaneously by several threads.

2 To complement the built-in multithread support in the runtime library, you must also:

- Implement code for the library's system locks interface.
- If file streams are used, implement code for the library's file stream locks interface.
- Implement code that handles thread creation, thread destruction, and TLS access methods for the library.

You can find the required declaration of functions in the `DLib_Threads.h` file. There you will also find more information.

3 Build your project.

Note: If you are using a third-party RTOS, check their guidelines for how to enable multithread support with IAR Systems tools.

C++ EXCEPTIONS IN THREADS

Using exceptions in threads works as long as the `main` function for the thread has the `noexcept` exception specification. Otherwise non-caught exceptions will not correctly terminate the application.

SETTING UP THREAD-LOCAL STORAGE (TLS)

Thread-local storage (TLS) is supported in both C (via the `_Thread_local` type specifier introduced in C11) and C++ (via the `thread_local` type specifier introduced in C++11). TLS variables reside in the thread-local storage area, a memory area that must be set up when the thread is created. Any resources used must be returned when the thread is destroyed. In a C++ environment, any TLS object must be created after the thread-local storage area has been set up, and destroyed before the thread-local storage area is destroyed.

If you are using an operating system, refer to the relevant TLS documentation. Additional information can be found in the IAR library header file DLib_Threads.h. Information from such specific sources takes precedence over this general overview.

The main thread

If the linker option --threaded_lib has been specified, TLS is active. The regular system startup handles the initialization of the main thread's thread-local storage area. The initialized TLS variables in the main thread are placed in the linker section .tdata and the zero-initialized TLS variables are placed in the section .tbss. All other threads must set up their thread-local storage area when they are created. If --threaded_lib was not specified, content in the .tdata and .tbss sections is handled as if they were .data and .bss. However, accesses to such variables are still TLS accesses.

Acquiring memory for TLS

TLS variables must be placed in memory. Exactly how this is handled does not matter as long as the memory remains available for the duration of the thread's lifetime. The size of the thread-local storage area can be obtained by calling the function __iar_tls_size (declared in DLib_Threads.h).

Some options for acquiring memory for TLS are:

- Acquire memory from the OS
- Allocate heap memory
- Use space on the stack of a function that does not return until the thread is done
- Use space in a dedicated section.

Initializing TLS memory

To initialize the TLS memory, call the function __iar_tls_init (declared in DLib_Threads.h) with a pointer to the memory area.

The initialization function copies the contents of the linker section __iar_tls$$INIT_DATA to the memory, and then zero-initializes the remaining memory up to the size of the section __iar_tls$$DATA. In a C++ environment, the function __iar_call_tls_ctors is also called—it executes all constructors in the section __iar_tls$$PREINIT_ARRAY. When the initialization has been performed, the thread-local storage area is ready to use, all TLS variables have their initial values, and in a C++ environment all thread-local objects have been constructed.

Deallocating TLS memory

When it is time to destroy the thread, the thread-local storage area must also be handled. In a C++ environment, the thread-local objects must be destroyed before the memory itself is processed. This is achieved by calling the function `__call_thread_dtors` (declared in `DLib_Threads.h`). If the memory was acquired from a handler (like the heap or the OS), that memory must be returned.

As an example, this code snippet allocates the thread-local storage area on the heap. `tp` is a pointer to a thread-control object:

```
/* creating a new thread */
  ...
  /* initialize TLS */
  void * tls_mem = malloc(__iar_tls_size()); /* get memory */
  __iar_tls_init(tls_mem);                    /* init TLS in the */
                                              /* new memory */
  tp->tls_area = tls_mem;                     /* set the thread's */
                                    /* TLS area to the new memory */
  ...
  /* destroying a thread */
  ...
  /* destroy the TLS area */
  __call_thread_dtors();                   /* only if C++ is used */
  free(tp->tls_area);                      /* return memory */
  ...
```

Assembler language interface

- Mixing C and assembler
- Calling assembler routines from C
- Calling assembler routines from C++
- Calling convention
- Call frame information

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 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.

For more 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 causes some overhead in the form of function call and return instruction sequences, and the compiler will regard some registers as scratch registers. 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 question is discussed in the section *Calling assembler routines from C*, page 182. The following two are covered in the section *Calling convention*, page 185.

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 195.

The recommended method for mixing C or C++ and assembler modules is described in *Calling assembler routines from C*, page 182, and *Calling assembler routines from C++*, page 185, respectively.

INLINE ASSEMBLER

Inline assembler can be used for inserting assembler instructions directly into a C or C++ function. Typically, this can be useful if you need to:

- Access hardware resources that are not accessible in C (in other words, when there is no definition for an SFR or there is no suitable intrinsic function available).
- Manually write a time-critical sequence of code that if written in C will not have the right timing.
- Manually write a speed-critical sequence of code that if written in C will be too slow.

An inline assembler statement is similar to a C function in that it can take input arguments (input operands), have return values (output operands), and read or write to

C symbols (via the operands). An inline assembler statement can also declare *clobbered resources*, that is, values in registers and memory that have been overwritten.

Limitations

Most things you can to do in normal assembler language are also possible with inline assembler, with the following differences:

- Alignment cannot be controlled—this means, for example, that `DC32` directives might be misaligned.
- The only accepted register synonyms **in 32-bit mode** are `SP` (for `R13`), `LR` (for `R14`), and `PC` (for `R15`).
- The only accepted register synonyms **in 64-bit mode** are `IP0` (for `X16`), `IP1` (for `X17`), `FP` (for `X29`), and `LR` (for `X30`).
- In general, assembler directives will cause errors or have no meaning. However, data definition directives will work as expected.
- Resources used (registers, memory, etc) that are also used by the C compiler must be declared as operands or clobbered resources.
- If you do not want to risk that the inline assembler statement to be optimized away by the compiler, you must declare it `volatile`.
- Accessing a C symbol or using a constant expression requires the use of operands.
- Dependencies between the expressions for the operands might result in an error.
- The pseudo-instruction `LDR Rd, =`*`expr`* is not available from inline assembler.

Risks with inline assembler

Without operands and clobbered resources, inline assembler statements have no interface with the surrounding C source code. This makes the inline assembler code fragile, and might also become a maintenance problem if you update the compiler in the future. There are also several limitations to using inline assembler without operands and clobbered resources:

- The compiler's various optimizations will disregard any effects of the inline statements, which will not be optimized at all.
- Inlining of functions with assembler statements without declared side-effects will not be done.
- The inline assembler statement will be `volatile` and *clobbered memory* is not implied. This means that the compiler will not remove the assembler statement. It will simply be inserted at the given location in the program flow. The consequences or side-effects that 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.

The following example—for Arm mode—demonstrates the risks of using the `asm` keyword without operands and clobbers:

```
int Add(int term1, int term2)
{
  asm("adds r0,r0,r1");
  return term1;
}
```

In this example:

- The function `Add` assumes that values are passed and returned in registers in a way that they might not always be, for example, if the function is inlined.
- The `s` in the `adds` instruction implies that the condition flags are updated, which you specify using the `cc` clobber operand. Otherwise, the compiler will assume that the condition flags are not modified.

Inline assembler without using operands or clobbered resources is therefore often best avoided. The compiler will issue a remark for them.

Reference information for inline assembler

The `asm` and `__asm` keywords both insert inline assembler instructions. However, when you compile C source code, the `asm` keyword is not available when the option `--strict` is used. The `__asm` keyword is always available.

Syntax

The syntax of an inline assembler statement is (similar to the one used by GNU GCC):

```
asm [volatile]( string [assembler-interface])
```

A *`string`* can contain one or more operations, separated by `\n`. Each operation can be a valid assembler instruction or a data definition assembler directive prefixed by an optional label. There can be no whitespace before the label and it must be followed by `:`.

For example:

```
asm("label:nop\n"
    "b label");
```

Note: Any labels you define in the inline assembler statement will be local to that statement. You can use this for loops or conditional code.

If you define a label in an inline assembler statement using two colons—for example, `"`*`label`*`:: nop\n"`—instead of one, the label will be public, not only in the inline assembler statement, but in the module as well. This feature is intended for testing only.

An assembler statement without declared side-effects will be treated as a volatile assembler statement, which means it cannot be optimized at all. The compiler will issue a remark for such an assembler statement.

assembler-interface is:

```
: comma-separated list of output operands     /* optional */
: comma-separated list of input operands      /* optional */
: comma-separated list of clobbered resources /* optional */
```

Operands

An inline assembler statement can have one input and one output comma-separated list of operands. Each operand consists of an optional symbolic name in brackets, a quoted constraint, followed by a C expression in parentheses.

Syntax of operands

`[[ symbolic-name ]] "[modifiers]constraint" (expr)`

For example:

```
int Add(int term1, int term2)
{
  int sum;

  asm("add %0,%1,%2"
      : "=r"(sum)
      : "r" (term1), "r" (term2));

  return sum;
}
```

In this example, the assembler instruction uses one output operand, `sum`, two input operands, `term1` and `term2`, and no clobbered resources.

It is possible to omit any list by leaving it empty. For example:

```
int matrix[M][N];

void MatrixPreloadRow(int row)
{
  asm volatile ("pld [%0]" : : "r" (&matrix[row][0]));
}
```

Operand constraints

The operand constraints define how to pass an operand between inline assembler code and the surrounding C or C++ code.

These are the constraint codes **in 32-bit mode**:

Constraint	Description
`r`	Uses a general purpose register for the expression: `R0-R12`, `R14` (for Arm and Thumb2) `R0-R7` (for Thumb1)
`l`	`R0-R7` (only valid for Thumb1)
`Rp`	Uses a pair of general purpose registers, for example `R0`, `R1`.
`Te`	Uses an even-numbered general purpose register for the expression.
`To`	Uses an odd-numbered general purpose register for the expression.
`i`	An immediate integer operand with a constant value. Symbolic constants are allowed.
`j`	A 16-bit constant suitable for a `MOVW` instruction (valid for Arm and Thumb2).
`n`	An immediate operand, alias for `i`.
`I`	A constant valid for a data processing instruction (for Arm and Thumb2), or a constant in the range 0 to 255 (for Thumb1).
`J`	An immediate constant in the range -4095 to 4095 (for Arm and Thumb2), or a constant in the range -255 to -1 (for Thumb1).
`K`	An immediate constant that satisfies the `I` constraint if inverted (for Arm and Thumb2), or a constant that satisfies the `I` constraint multiplied by any power of 2 (for Thumb1).
`L`	An immediate constant that satisfies the `I` constraint if negated (for Arm and Thumb2), or a constant in the range -7 to 7 (for Thumb1).
`M`	An immediate constant that is a multiple of 4 in the range 0 to 1020 (only valid for Thumb1).
`N`	An immediate constant in the range 0 to 31 (only valid for Thumb1).
`O`	An immediate constant that is a multiple of 4 in the range -508 to 508 (only valid for Thumb1).
`t`	An `S` register.
`w`	A `D` register.
`q`	A `Q` register.
`Dv`	A 32-bit floating-point immediate constant for the `VMOV.F32` instruction.
`Dy`	A 64-bit floating-point immediate constant for the `VMOV.F64` instruction.

Table 10: Inline assembler operand constraints in 32-bit mode

Constraint	Description
v2S ... v4Q	A vector of 2, 3, or 4 consecutive S, D, or Q registers. For example, v4Q is a vector of four Q registers. The vectors do not overlap, so the available v4Q register vectors are Q0-Q3, Q4-Q7, Q8-Q11, and Q12-Q15.

Table 10: Inline assembler operand constraints in 32-bit mode (Continued)

These are the constraint codes **in 64-bit mode**:

Constraint	Description
r	Uses a 64-bit general purpose register for the expression: X0-X30. If you want the compiler to use the 32-bit general purpose registers W0–W31 instead, use the w operand modifier.
i	An immediate integer operand with a constant value. Symbolic constants are allowed.
n	An immediate operand, alias for i.
I	A constant in the range 0–4095, with an optional left shift by 12. The range that the ADD and SUB instructions accept.
J	A constant in the range –4095 to 0, with an optional left shift by 12.
K	An immediate constant that is valid for 32-bit logical instructions. For example, AND, ORR, EOR.
L	An immediate constant that is valid for 64-bit logical instructions. For example, AND, ORR, EOR.
M	An immediate constant that is valid for a MOV instruction with a destination of a 32-bit register. Valid values are all values that the K constraint accepts, plus the values that the MOVZ, MOVN, and MOVK instructions accept.
N	An immediate constant that is valid for a MOV instruction with a destination of a 64-bit register. Valid values are all values that the L constraint accepts, plus the values that the MOVZ, MOVN, and MOVK instructions accept.
w	Uses a SIMD or floating-point register, V0–V31. The b, h, s, d, and q operand modifiers can override this behavior.
x	The operand must be a 128-bit vector type. The compiler uses a low SIMD register, V0–V15.

Table 11: Inline assembler operand constraints in 64-bit mode

Constraint modifiers

Constraint modifiers can be used together with a constraint to modify its meaning. This table lists the supported constraint modifiers:

Modifier	Description
=	Write-only operand
+	Read-write operand
&	Early clobber output operand which is written to before the instruction has processed all the input operands.

Table 12: Supported constraint modifiers

Referring to operands

Assembler instructions refer to operands by prefixing their order number with `%`. The first operand has order number `0` and is referred to by `%0`.

If the operand has a symbolic name, you can refer to it using the syntax `%[`*`operand.name`*`]`. Symbolic operand names are in a separate namespace from C/C++ code and can be the same as a C/C++ variable names. Each operand name must however be unique in each assembler statement. For example:

```
int Add(int term1, int term2)
{
  int sum;

  asm("add %[Rd],%[Rn],%[Rm]"
      : [Rd]"=r"(sum)
      : [Rn]"r" (term1), [Rm]"r" (term2));

  return sum;
}
```

Input operands

Input operands cannot have any constraint modifiers, but they can have any valid C expression as long as the type of the expression fits the register.

The C expression will be evaluated just before any of the assembler instructions in the inline assembler statement and assigned to the constraint, for example, a register.

Output operands

Output operands must have `=` as a constraint modifier and the C expression must be an l-value and specify a writable location. For example, `=r` for a write-only general purpose register. The constraint will be assigned to the evaluated C expression (as an l-value) immediately after the last assembler instruction in the inline assembler statement. Input operands are assumed to be consumed before output is produced and the compiler may use the same register for an input and output operand. To prohibit this, prefix the output constraint with `&` to make it an early clobber resource, for example, `=&r`. This will ensure that the output operand will be allocated in a different register than the input operands.

Input/output operands

An operand that should be used both for input and output must be listed as an output operand and have the `+` modifier. The C expression must be an l-value and specify a writable location. The location will be read immediately before any assembler instructions and it will be written to right after the last assembler instruction.

This is an example of using a read-write operand:

```
int Double(int value)
{
  asm("add %0,%0,%0" : "+r"(value));

  return value;
}
```

In the example above, the input value for `value` will be placed in a general purpose register. After the assembler statement, the result from the `ADD` instruction will be placed in the same register.

Clobbered resources

An inline assembler statement can have a list of clobbered resources.

"*resource1*", "*resource2*", ...

Specify clobbered resources to inform the compiler about which resources the inline assembler statement destroys. Any value that resides in a clobbered resource and that is needed after the inline assembler statement will be reloaded.

Clobbered resources will not be used as input or output operands.

This is an example of how to use clobbered resources:

```
int Add(int term1, int term2)
{
  int sum;

  asm("adds %0,%1,%2"
      : "=r"(sum)
      : "r" (term1), "r" (term2)
      : "cc");

  return sum;
}
```

In this example, the condition codes will be modified by the `ADDS` instruction. Therefore, `"cc"` must be listed in the clobber list.

This table lists valid clobbered resources:

Clobber	Description
R0-R12, R14 for Arm mode and Thumb2 R0-R7, R12, R14 for Thumb1 X0-X30, W0-W30 for A64	General purpose registers
S0-S31, D0-D31, Q0-Q15 for Arm mode and Thumb2 V0-V31, B0-B31, H0-H31, S0-S31, D0-D31, Q0-Q31 for A64	Floating-point registers
cc	The condition flags (N, Z, V, and C)
memory	To be used if the instructions modify any memory. This will avoid keeping memory values cached in registers across the inline assembler statement.

Table 13: List of valid clobbers

Operand modifiers

An operand modifier is a single letter between the % and the operand number, which is used for transforming the operand.

In the example below, the modifiers L and H are used for accessing the least and most significant 16 bits, respectively, of an immediate operand:

```
int Mov32()
{
  int a;
  asm("movw %0,%L1 \n"
      "movt %0,%H1 \n" : "=r"(a) : "i"(0x12345678UL));
  return a;
}
```

Some operand modifiers can be combined, in which case each letter will transform the result from the previous modifier.

This table describes the transformation performed by each valid modifier **in 32-bit mode**:

Modifier	Description
L	The lowest-numbered register of a register pair, or the low 16 bits of an immediate constant.
H	The highest-numbered register of a register pair, or the high 16 bits of an immediate constant.

Table 14: Operand modifiers and transformations in 32-bit mode

Modifier	Description
c	For an immediate operand, an integer or symbol address without a preceding # sign. Cannot be transformed by additional operand modifiers.
B	For an immediate operand, the bitwise inverse of integer or symbol without a preceding # sign. Cannot be transformed by additional operand modifiers.
Q	The least significant register of a register pair.
R	The most significant register of a register pair.
M	For a register or a register pair, the register list suitable for `ldm` or `stm`. Cannot be transformed by additional operand modifiers.
a	Transforms a register `Rn` into a memory operand `[Rn,#0]` suitable for `pld`.
b	The low `S` register part of a `D` register.
p	The high `S` register part of a `D` register.
e	The low `D` register part of a `Q` register, or the low register in a vector of Neon registers.
f	The high `D` register part of a `Q` register, or the high register in a vector of Neon registers.
h	For a (vector of) `D` or `Q` registers, the corresponding list of `D` registers within curly braces. For example, `Q0` becomes `{D0,D1}`. Cannot be transformed by additional operand modifiers.
y	`S` register as indexed `D` register, for example `S7` becomes `D3[1]`. Cannot be transformed by additional operand modifiers.

Table 14: Operand modifiers and transformations in 32-bit mode (Continued)

This table describes the transformation performed by each valid modifier **in 64-bit mode**:

Modifier	Description
c	For an immediate operand, an integer, or symbol address without a preceding # sign. Cannot be transformed by additional operand modifiers.
a	The operand constraint must be `r`. Prints the register name surrounded by square brackets. Suitable for use as a memory operand.
n	For an immediate operand. Prints the arithmetic negation of the value without a preceding #.

Table 15: Operand modifiers and transformations in 64-bit mode

Modifier	Description
w	The operand constraint must be r. Prints the register using its 32-bit W name.
x	The operand constraint must be r. Prints the register using its 64-bit X name.
b	The operand constraint must be w or x. Prints the register using its 8-bit B name.
h	The operand constraint must be w or x. Prints the register using its 16-bit H name.
s	The operand constraint must be w or x. Prints the register using its 32-bit S name.
d	The operand constraint must be w or x. Prints the register using its 64-bit D name.
q	The operand constraint must be w or x. Prints the register using its 128-bit Q name.

Table 15: Operand modifiers and transformations in 64-bit mode (Continued)

AN EXAMPLE OF HOW TO USE CLOBBERED MEMORY

```
int StoreExclusive(unsigned long * location, unsigned long value)
{
  int failed;

  asm("strex %0,%2,[%1]"
      : "=&r"(failed)
      : "r"(location), "r"(value)
      : "memory");

  /* Note: 'strex' requires Armv6 (Arm) or Armv6T2 (THUMB) */

  return failed;
}
```

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: 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 SKELETON 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 . -On -e
```

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`. The `-On` option means that no optimization will be used and `-e` enables language extensions. In addition, make sure to use relevant compiler options, usually the same as you use for other C or C++ source files in your project.

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, to exclude the `CFI` directives from the list file, choose **Project>Options>C/C++ Compiler>List** and deselect the suboption **Include call frame information**.

On the command line, to exclude the `CFI` directives from the list file, use the option `-lB` instead of `-lA`.

Note: `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 195.

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 convention used by the compiler. These items are examined:

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

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

The calling convention used by the compiler adheres to the Procedure Call Standard for the Arm architecture, AAPCS, a part of AEABI, see *AEABI compliance*, page 230. AAPCS is not fully described here. For example, the use of floating-point coprocessor registers when using the VFP calling convention is not covered.

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.

In 32-bit mode, any of the registers `R0` to `R3`, and `R12`, can be used as a scratch register by the function. **In 64-bit mode**, the registers that can be used as scratch registers are the registers `X0` to `X15`.

Note: **In 32-bit mode**, `R12`, and **in 64-bit mode**, `X16` and `X17`, are also scratch registers when calling between assembler functions 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.

In 32-bit mode, the registers `R4` through to `R11` are preserved registers. They are preserved by the called function. **In 64-bit mode**, the registers `X18` to `X30` are preserved registers.

Special registers in 32-bit mode

For these 32-bit mode registers, you must consider these 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.

Special registers in 64-bit mode

For these 64-bit mode registers, you must consider certain prerequisites:

- The stack pointer register, `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 16-byte aligned. In the function, the stack pointer must always be word aligned. At exit, `SP` must have the same value that it had at entry.
- The link register, `LR/X30`, holds the return address at the entrance of the function.

FUNCTION ENTRANCE

Parameters can be passed to a function using one of these basic methods:

- In registers
- 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 182.

Register parameters in 32-bit mode

The registers available in 32-bit mode for passing parameters are:

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 the first available register pair: `R0:R1` or `R2:R3`

Table 16: Registers used in 32-bit mode 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`.

Register parameters in 64-bit mode

The registers available in 64-bit mode for passing parameters are:

Parameters	Passed in registers
Integers, pointers, small structures (up to 8 bytes)	Passed using the first free register: `X0-X7`
Small structures (9–16 bytes)	Passed using the first free register pair: `X0-X7`
Floating-point values	Passed using the first free register: `V0-V7`
Homogeneous structures (1–4 elements of the same floating-point or vector type)	Passed using the first free registers: `V0-V7` (one element in each register)
Large structures	Pointer is passed using the first free register: `X0-X7`

Table 17: Registers used in 64-bit mode for passing parameters

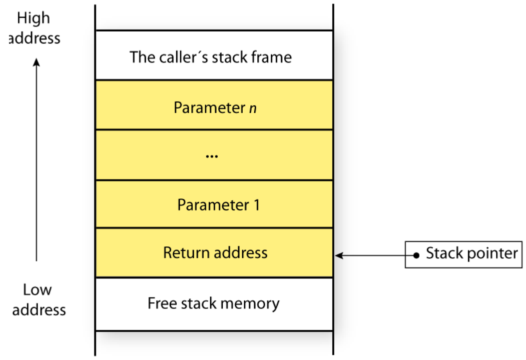
The assignment of registers to parameters is a straightforward process. Traversing the parameters from left to right, the first parameter is assigned to the available register or registers. Should there be no more available registers, the parameter is passed on the stack in reverse order.

In 64-bit mode, only the bits that are consistent with a parameter's size can be accessed. Therefore, the called function normally sign- or zero-extends parameters that have a size smaller than 32 bits.

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.

This figure illustrates how parameters are stored on the stack:



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 in 32-bit mode for returning values

The registers available in 32-bit mode for returning values are `R0` and `R0:R1`.

Return values	Passed in registers
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 18: Registers used in 32-bit mode for returning values

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

Registers used in 64-bit mode for returning values

Return values	Passed in registers
Integers, pointers, small structures (up to 8 bytes)	X0
Small structures (9–16 bytes)	X0-X1
Floating-point values	V0
Homogeneous structures (1–4 elements of the same floating-point or vector type)	V0-V3 (one element in each register)
Large structures	Pointer is passed by caller in X8

Table 19: Registers used in 64-bit mode for returning values

Only the bits of the return value that are consistent with the size of the return value can be accessed.

Stack layout at function exit

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

32-bit mode: 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:

```
name    call
section .text:CODE
extern  func

push    {r4-r6,lr}    ; Preserve stack alignment 8
bl      func

; Do something here.

pop     {r4-r6,pc}    ;  return

end
```

64-bit mode: 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 on the stack. At function exit, all these registers must be restored from the stack.

The following example shows what this can look like:

```
name    call
section .text:CODE
extern  func

strp    x9, lr, [sp, #16]!  ; Preserve stack alignment 16
bl      func

; Do something here.

ldrp    x9, x7, [sp, #16]
ret

end
```

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);
```

In 32-bit mode, 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:

```
name    return
section .text:CODE
add     r0, r0, #1
bx      lr
end
```

In 64-bit mode, the function takes one parameter in register X0, and the return value is passed back to its caller in register X0. A corresponding assembler routine that is compatible with the declaration looks like this:

```
name    return
section .text:CODE
add     x0, x0, #1
ret
end
```

Example 2

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

```
struct MyStruct
{
  short a;
  short b;
  short c;
  short d;
  short e;
};

int MyFunction(struct MyStruct x, int y);
```

In 32-bit mode, 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.

In 64-bit mode, the value of x is passed in X0 and X1, and y is passed in X2. The return value is passed in X0.

Example 3

The function below will return a structure of type struct MyStruct.

```
struct MyStruct
{
  int mA[20];
};

struct MyStruct MyFunction(int x);
```

It is the responsibility of the calling function to allocate a memory location for the return value and pass a pointer to it as a hidden first parameter. **In 32-bit mode**, the pointer to the location where the return value should be stored is passed in R0. The parameter x is passed in R1. **In 64-bit mode**, the pointer to the location where the return value should be stored is passed in X8. The parameter x is passed in X0.

Assume that the function instead was declared to return a pointer to the structure:

```
struct MyStruct *MyFunction(int x);
```

In this case, the return value is a scalar, so there is no hidden parameter. **In 32-bit mode**, the parameter x is passed in R0, and the return value is returned in R0. **In 64-bit mode**, the parameter x is passed in X0, and the return value is returned in X0.

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 *IAR Assembler User Guide for Arm*.

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.

32-bit mode call frame information resources:

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
D0–D31	Vector Floating Point (VFP) 64-bit coprocessor register
CPSR	Current program status register
SPSR	Saved program status register

Table 20: 32-bit mode call frame information resources defined in a names block

64-bit mode call frame information resources:

Resource	Description
X0–X29	Processor general-purpose 64-bit registers
X30	Link register, LR
SP	Stack pointer
CFA SP	The call frames of the stack
ELR_mode	Exception level
V0–V31	Vector Floating Point (VFP) 64-bit registers (in reality, they are 128 bits, but the ABI cannot handle this)

Table 21: 64-bit mode 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 **in 32-bit mode** looks like this.

```
        NAME Cfi

        RTMODEL "__SystemLibrary", "DLib"

        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,PC}          ;; return
        CFI EndBlock cfiBlock0

        END
```

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

Using C

- C language overview
- Extensions overview
- IAR C language extensions

C language overview

The IAR C/C++ Compiler for Arm supports the INCITS/ISO/IEC 9899:2018 standard, also known as C18. C18 addresses defects in C11 (INCITS/ISO/IEC 9899:2012) without introducing any new language features. This means that the C11 standard is also supported. In this guide, the C18 standard is referred to as *Standard C* and is the default standard used in the compiler. This standard is stricter than C89.

The compiler will accept source code written in the C18 standard or a superset thereof.

In addition, the compiler also supports the ISO 9899:1990 standard (including all technical corrigenda and addenda), also known as C94, C90, C89, and ANSI C. In this guide, this standard is referred to as *C89*. Use the `--c89` compiler option to enable this standard.

With Standard C enabled, the IAR C/C++ Compiler for Arm can compile all C18/C11 source code files, except for those that depend on thread-related system header files.

The floating-point standard that Standard C binds to is IEC 60559—known as ISO/IEC/IEEE 60559—which is nearly identical to the IEEE 754 format.

Annex K (*Bounds-checking interfaces*) of the C standard is supported. See *Bounds checking functionality*, page 139.

For an overview of the differences between the various versions of the C standard, see the Wikipedia articles *C18 (C standard revision)*, *C11 (C standard revision)*, or *C99*.

Extensions overview

The compiler offers the features of Standard 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.

This is an overview of the available extensions:

- *IAR C language extensions*

  For information about available language extensions, see *IAR C language extensions*, page 201. For more 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 Standard C 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 information about available pragma directives, see the chapter *Pragma directives*.

- *Preprocessor extensions*

  The preprocessor of the compiler adheres to Standard C. 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 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. For more information about using intrinsic functions, see *Mixing C and assembler*, page 171. For information about available functions, see the chapter *Intrinsic functions*.

- *Library functions*

  The DLIB runtime environment provides the C and C++ library definitions in the C/C++ standard library that apply to embedded systems. For more information, see *DLIB runtime environment—implementation details*, page 491.

Note: Any use of these extensions, except for the pragma directives, makes your source code inconsistent with Standard C.

ENABLING LANGUAGE EXTENSIONS

You can choose different levels of language conformance by means of project options:

Command line	IDE*	Description
`--strict`	**Strict**	All IAR C language extensions are disabled—errors are issued for anything that is not part of Standard C.
None	**Standard**	All *relaxations to Standard C* are enabled, but no *extensions for embedded systems programming*. For information about extensions, see *IAR C language extensions*, page 201.
`-e`	**Standard with IAR extensions**	All *IAR C language extensions* are enabled.

Table 22: Language extensions

* In the IDE, choose **Project>Options>C/C++ Compiler>Language 1>Language conformance** and select the appropriate option. Note that language extensions are enabled by default.

IAR C language extensions

The compiler provides a wide set of C language extensions. To help you to find the extensions required by your application, they are grouped like this in this section:

- *Extensions for embedded systems programming*—extensions specifically tailored for efficient embedded programming for the specific core you are using, typically to meet memory restrictions
- *Relaxations to Standard C*—that is, the relaxation of some minor Standard C issues and also some useful but minor syntax extensions, see *Relaxations to Standard C*, page 203.

EXTENSIONS FOR EMBEDDED SYSTEMS PROGRAMMING

The following language extensions are available both in the C and the C++ programming languages and they 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 features, see *Controlling data and function placement in memory*, page 242, and *location*, page 414.

- *Alignment control*

  Each data type has its own alignment. For more information, see *Alignment*, page 367. If you want to change the alignment, the `__packed` data type attribute, the `#pragma pack` directive, and the `#pragma data_alignment` directive are available. If you want to check the alignment of an object, use the `__ALIGNOF__()` operator.

  The `__ALIGNOF__` operator is used for accessing the alignment of an object. It takes one of two forms:

  - `__ALIGNOF__ (`*`type`*`)`
  - `__ALIGNOF__ (`*`expression`*`)`

  In the second form, the expression is not evaluated.

  See also the Standard C file `stdalign.h`.

- *Bitfields and non-standard types*

  In Standard C, a bitfield must be of the type `int` or `unsigned int`. Using IAR C language extensions, any integer type or enumeration can be used. The advantage is that the struct will sometimes be smaller. For more information, see *Bitfields*, page 370.

Dedicated section operators

The compiler supports getting the start address, end address, and size for a section with these built-in section operators:

`__section_begin`	Returns the address of the first byte of the named section or block.
`__section_end`	Returns the address of the first byte *after* the named section or block.
`__section_size`	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.

The operators can be used on named sections or on named blocks defined in the linker configuration file.

These operators behave syntactically as if declared like:

```
void * __section_begin(char const * section)
void * __section_end(char const * section)
size_t __section_size(char const * section)
```

When you use the @ operator or the #pragma location directive to place a data object or a function in a user-defined section, or when you use named blocks in the linker configuration file, the section operators can be used for getting the start and end address of the memory range where the sections or blocks were placed.

The named *section* must be a string literal and it 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 23: Section operators and their symbols

Note: The linker will not necessarily place sections with the same name consecutively 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 consecutively, 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 422, and *location*, page 414.

RELAXATIONS TO STANDARD C

This section lists and briefly describes the relaxation of some Standard C 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 extensions allow you to first declare the name of an `enum` and later resolve it by specifying the brace-enclosed list.
- Accepting missing semicolon at the end of a `struct` or `union` specifier

  A warning—instead of an error—is issued if the semicolon at the end of a `struct` or `union` specifier is missing.
- Null and `void`

  In operations on pointers, a pointer to `void` is always implicitly converted to another type if necessary, and a null pointer constant is always implicitly converted to a null pointer of the right type if necessary. In Standard C, some operators allow this kind of behavior, while others do not allow it.
- 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 377.
- Taking the address of a register variable

  In Standard C, it is illegal to take the address of a variable specified as a register variable. The compiler allows this, but a warning is issued.
- `long float` means `double`

  The type `long float` is accepted as a synonym for `double`.
- Binary integer literals (`0b`...) are supported.
- 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-`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 the strict Standard C 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 Standard C mode, a warning is issued.

- A label preceding a `}`

  In Standard 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 allows this, but issues a warning. Note that 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

  Standard 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;
}
```

- Static functions in function and block scopes

  Static functions may be declared in function and block scopes. Their declarations are moved to the file scope.

- Numbers scanned according to the syntax for numbers

  Numbers are scanned according to the syntax for numbers rather than the `pp-number` syntax. Therefore, `0x123e+1` is scanned as three tokens instead of one valid token. (If the `--strict` option is used, the `pp-number` syntax is used instead.)
- Empty translation unit

  A translation unit (input file) might be empty of declarations.
- Assignment of pointer types

  Assignment of pointer types is allowed in cases where the destination type has added type qualifiers that are not at the top level, for example, `int **` to `const int **`. Comparisons and pointer difference of such pairs of pointer types are also allowed. A warning is issued.
- Pointers to different function types

  Pointers to different function types might be assigned or compared for equality (`==`) or inequality (`!=`) without an explicit type cast. A warning is issued. This extension is not allowed in C++ mode.
- Assembler statements

  Assembler statements are accepted. This is disabled in strict C mode because it conflicts with the C standard for a call to the implicitly declared `asm` function.
- `#include_next`

  The non-standard preprocessing directive `#include_next` is supported. This is a variant of the `#include` directive. It searches for the named file only in the directories on the search path that follow the directory in which the current source file (the one containing the `#include_next` directive) is found. This is an extension found in the GNU C compiler.
- `#warning`

  The non-standard preprocessing directive `#warning` is supported. It is similar to the `#error` directive, but results in a warning instead of a catastrophic error when processed. This directive is not recognized in strict mode. This is an extension found in the GNU C compiler.
- Concatenating strings

  Mixed string concatenations are accepted.

  ```
  wchar_t * str="a" L "b";
  ```
- GNU style statement expressions are accepted.
- GNU style case ranges are accepted (`case 1..5:`).
- GNU style designated initializer ranges are accepted.

  Example: `int widths[] = {[0...9] = 1, [10...99] = 2, [100] = 3};`

Using C++

- Overview—Standard C++
- Enabling support for C++
- C++ feature descriptions
- C++ language extensions
- Porting code from EC++ or EEC++

Overview—Standard C++

The IAR C++ implementation fully complies with the ISO/IEC 14882:2014 C++ standard, except for source code that depends on thread-related system headers. The ISO/IEC 14882:2014 C++ standard is also known as C++14. In this guide, this standard is referred to as Standard C++.

The IAR C/C++ compiler accepts source code written in the C++14 standard or a superset thereof. The compiler also supports the added language features in ISO/IEC 14882:2017 (C++17), but not the added library features.

Atomic operations are available for cores where the instruction set supports them. See *Atomic operations*, page 497.

For an overview of the differences between the various versions of the C++ standard, see the Wikipedia articles *C++17*, *C++14*, *C++11*, or *C++* (for information about C++98).

Note: There is an alternate version of the C++ Standard Template Library (STL) that was used in earlier versions of IAR Embedded Workbench for Arm. This version of STL is simpler and can under favorable circumstances produce smaller code than the current version. This version of STL can be used in conjunction with the ordinary C++ library as long as accesses to the containers are not mixed. See the documentation for it in the file `arm/doc/HelpDLIB5.html`.

MODES FOR EXCEPTIONS AND RTTI SUPPORT

Both exceptions and runtime type information result in increased code size simply by being included in your application. You might want to disable either or both of these features to avoid this increase:

- Support for runtime type information constructs can be disabled by using the compiler option `--no_rtti`
- Support for exceptions can be disabled by using the compiler option `--no_exceptions`

Even if support is enabled while compiling, the linker can avoid including the extra code and tables in the final application. If no part of your application actually throws an exception, the code and tables supporting the use of exceptions are not included in the application code image. Also, if dynamic runtime type information constructs (`dynamic_cast`/`typeid`) are not used with polymorphic types, the objects needed to support them are not included in the application code image. To control this behavior, use the linker options `--no_exceptions`, `--force_exceptions`, and `--no_dynamic_rtti_elimination`.

Disabling exception support

When you use the compiler option `--no_exceptions`, the following will generate a compiler error:

- `throw` expressions
- `try-catch` statements
- Exception specifications on function definitions.

In addition, the extra code and tables needed to handle destruction of objects with auto storage duration when an exception is propagated through a function will not be generated when the compiler option `--no_exceptions` is used.

All functionality in system header files not directly involving exceptions is supported when the compiler option `--no_exceptions` is used.

The linker will produce an error if you try to link C++ modules compiled with exception support with modules compiled without exception support

For more information, see *--no_exceptions*, page 304.

Disabling RTTI support

When you use the compiler option `--no_rtti`, the following will generate a compiler error:

- The `typeid` operator

- The dynamic_cast operator.

Note: If --no_rtti is used but exception support is enabled, most RTTI support is still included in the compiler output object file because it is needed for exceptions to work.

For more information, see *--no_rtti*, page 306.

EXCEPTION HANDLING

Exception handling can be divided into three parts:

- *Exception raise mechanisms*—in C++ they are the throw and rethrow expressions.
- *Exception catch mechanisms*—in C++ they are the try–catch statements, the exception specifications for a function, and the implicit catch to prevent an exception leaking out from main.
- *Information about currently active functions*—if they have try–catch statements and the set of auto objects whose destructors need to be run if an exception is propagated through the function.

When an exception is raised, the function call stack is unwound, function by function, block by block. For each function or block, the destructors of auto objects that need destruction are run, and a check is made whether there is a catch handler for the exception. If there is, the execution will continue from that catch handler.

An application that mixes C++ code with assembler and C code, and that throws exceptions from one C++ function to another via assembler routines and C functions must use the linker option --exception_tables with the argument unwind.

The implementation of exceptions

Exceptions are implemented using a table method. For each function, the tables describe:

- How to unwind the function, that is, how to find its caller on the stack and restore registers that need restoring
- Which catch handlers that exist in the function
- Whether the function has an exception specification and which exceptions it allows to propagate
- The set of auto objects whose destructors must be run.

When an exception is raised, the runtime will proceed in two phases. The first phase will use the exception tables to search the stack for a function invocation containing a catch handler or exception specification that would cause stack unwinding to halt at that point. Once this point is found, the second phase is entered, doing the actual unwinding, and running the destructors of auto objects where that is needed.

The table method results in virtually no overhead in execution time or RAM usage when an exception is not actually thrown. It does incur a significant penalty in read-only memory usage for the tables and the extra code, and throwing and catching an exception is a relatively expensive operation.

The destruction of auto objects when the stack is being unwound as a result of an exception is implemented in code separated from the code that handles the normal operation of a function. This code, together with the code in catch handlers, is placed in a separate section (`.exc.text`) from the normal code (normally placed in `.text`). In some cases, for instance when there is fast and slow ROM memory, it can be advantageous to select on this difference when placing sections in the linker configuration file.

Enabling support for C++

In the compiler, the default language is C.

To compile files written in Standard C++, use the `--c++` compiler option. See --*c++*, page 286.

To enable C++ in the IDE, choose **Project>Options>C/C++ Compiler>Language 1>Language>C++**.

C++ 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.

USING IAR ATTRIBUTES WITH CLASSES

Static data members of C++ classes are treated the same way global variables are, and can have any applicable IAR type and object attribute.

Member functions are in general treated the same way free functions are, and can have any applicable IAR type and object attributes. Virtual member functions can only have attributes that are compatible with default function pointers, and constructors and destructors cannot have any such attributes.

The location operator `@` and the `#pragma location` directive can be used on static data members and with all member functions.

TEMPLATES

C++ supports templates according to the C++ standard. 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.

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
}
```

USING STATIC CLASS OBJECTS IN INTERRUPTS

If interrupt functions use static class objects that need to be constructed (using constructors) or destroyed (using destructors), your application will not work properly if the interrupt occurs before the objects are constructed, or, during or after the objects are destroyed.

To avoid this, make sure that these interrupts are not enabled until the static objects have been constructed, and are disabled when returning from `main` or calling `exit`. For information about system startup, see *System startup and termination*, page 150.

Function local static class objects are constructed the first time execution passes through their declaration, and are destroyed when returning from `main` or when calling `exit`.

USING NEW HANDLERS

To handle memory exhaustion, you can use the `set_new_handler` function.

New handlers in Standard C++ with exceptions enabled

If you do not call `set_new_handler`, or call it with a NULL `new` handler, and `operator new` fails to allocate enough memory, `operator new` will throw `std::bad_alloc` if exceptions are enabled. If exceptions are not enabled, `operator new` will instead call `abort`.

If you call `set_new_handler` with a non-NULL `new` handler, the provided `new` handler will be called by `operator new` if the `operator new` fails to allocate enough memory. The `new` handler must then make more memory available and return, or abort execution in some manner. If exceptions are enabled, the `new` handler can also throw a `std::bad_alloc` exception. The `nothrow` variant of `operator new` will only return NULL in the presence of a `new` handler if exceptions are enabled and the `new` handler throws `std::bad_alloc`.

New handlers in Standard C++ with exceptions disabled

If you do not call `set_new_handler`, or call it with a NULL `new` handler, and `operator new` fails to allocate enough memory, it will call `abort`. The `nothrow` variant of the `new` operator will instead return NULL.

If you call `set_new_handler` with a non-NULL `new` handler, the provided `new` handler will be called by `operator new` if `operator new` fails to allocate memory. The `new` handler must then make more memory available and return, or abort execution in some manner. The `nothrow` variant of `operator new` will never return NULL in the presence of a `new` handler.

This is the same behavior as using the `nothrow` variants of `new`.

DEBUG SUPPORT IN C-SPY

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.

Using C++, you can make C-SPY stop at a `throw` statement or if a raised exception does not have any corresponding `catch` statement.

For more information, see the *C-SPY® Debugging Guide for Arm*.

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 the standard
};
```

- 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 the 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 the 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.

- An anonymous union can be introduced into a containing class by a `typedef` name. It is not necessary to first declare the union. For example:

```
typedef union
{
  int i,j;
} U;  // U identifies a reusable anonymous union.

class A
{
public:
  U;  // OK -- references to A::i and A::j are allowed.
};
```

In addition, this extension also permits *anonymous classes* and *anonymous structs*, as long as they have no C++ features—for example, no static data members or member functions, and no non-public members—and have no nested types other than other anonymous classes, structs, or unions. For example:

```
struct A
{
  struct
  {
    int i,j;
  }; // OK -- references to A::i and A::j are allowed.
};
```

- The `friend` class syntax allows non-class types as well as class types expressed through a `typedef` without an elaborated type name. For example:

```
typedef struct S ST;

class C
{
public:
  friend S;  // Okay (requires S to be in scope)
  friend ST; // Okay (same as "friend S;")
  // friend S const; // Error, cv-qualifiers cannot
                     // appear directly
};
```

- It is allowed to specify an array with no size or size 0 as the last member of a struct. For example:

  ```
  typedef struct
  {
    int i;
    char ir[0]; // Zero-length array
  };

  typedef struct
  {
    int i;
    char ir[];  // Zero-length array
  };
  ```

- Arrays of incomplete types

  An array can have an incomplete `struct`, `union`, `enum`, or `class` 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.

- Concatenating strings

  Mixed string literal concatenations are accepted.

  ```
  wchar_t * str = "a" L "b";
  ```

- Trailing comma

  A trailing comma in the definition of an enumeration type is silently accepted.

Except where noted, all of the extensions described for C are also allowed in C++ mode.

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

Porting code from EC++ or EEC++

Apart from the fact that Standard C++ is a much larger language than EC++ or EEC++, there are two issues that might prevent EC++ and EEC++ code from compiling:

- The library is placed in `namespace std`.

  There are two remedy options:

  - Prefix each used library symbol with `std::`.
  - Insert `using namespace std;` after the last include directive for a C++ system header file.

- Some library symbols have changed names or parameter passing.

  To resolve this, look up the new names and parameter passing.

Application-related considerations

- Output format considerations
- Stack considerations
- Heap considerations
- Interaction between the tools and your application
- Checksum calculation for verifying image integrity
- AEABI compliance
- CMSIS integration (32-bit mode)
- Arm TrustZone®
- Patching symbol definitions using $Super$$ and $Sub$$

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.

For a complete list of supported output formats, run `ielftool` without options.

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 557.

Stack considerations

To make your application use stack memory efficiently, there are some considerations to be made.

STACK SIZE CONSIDERATIONS

The required stack size depends heavily on the application's behavior. If the given stack size is too large, RAM will be wasted. If the given stack size is too small, one of two things can happen, depending on where in memory you located your stack:

- Variable storage will be overwritten, leading to undefined behavior
- The stack will fall outside of the memory area, leading to an abnormal termination of your application.

Both alternatives are likely to result in application failure. 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 stack memory*, page 118, and *Saving stack space and RAM memory*, page 253.

STACK ALIGNMENT

In 32-bit mode, the default `cstartup` code automatically initializes all stacks to an 8-byte aligned address.

In 64-bit mode, the default `cstartup` code automatically initializes all stacks to a 16-byte aligned address.

For more information about aligning the stack, see *Calling convention*, page 185 and more specifically *Special registers in 32-bit mode*, page 188 and *Stack parameters and layout*, page 190.

EXCEPTION STACK

64-bit Arm cores and Cortex-M do not have individual exception stacks. By default, all exception stacks are placed in the `CSTACK` section.

The Arm7/9/11, Cortex-A, and Cortex-R devices support 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`	Operation 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 24: Exception stacks for Arm7/9/11, Cortex-A, and Cortex-R

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.

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 a corresponding function) or the C++ operator `new`.

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

- The use of basic, advanced, and no-free heap memory allocation
- Linker sections used for the heap
- Allocating the heap size, see *Setting up heap memory*, page 118.

HEAP MEMORY HANDLERS

The system library contains three separate heap memory handlers—the *basic*, the *advanced*, and the *no-free* heap handler.

- If there are calls to heap memory allocation routines in your application, but no calls to heap deallocation routines, the linker automatically chooses the no-free heap.
- If there are calls to heap memory allocation routines in your application, the linker automatically chooses the advanced heap.
- If there are calls to heap memory allocation routines in, for example, the library, the linker automatically chooses the basic heap.

Note: If your product has a size-limited KickStart license, the basic heap is automatically chosen.

You can use a linker option to explicitly specify which handler you want to use:

- The basic heap (`--basic_heap`) is a simple heap allocator, suitable for use in applications that do not use the heap very much. In particular, it can be used in applications that only allocate heap memory and never free it. The basic heap is not particularly speedy, and using it in applications that repeatedly free memory is quite likely to lead to unneeded fragmentation of the heap. The code for the basic heap is significantly smaller than that for the advanced heap. See *--basic_heap*, page 332.
- The advanced heap (`--advanced_heap`) provides efficient memory management for applications that use the heap extensively. In particular, applications that repeatedly allocate and free memory will likely get less overhead in both space and time. The code for the advanced heap is significantly larger than that for the basic heap. See *--advanced_heap*, page 331. For information about the definition, see *iar_dlmalloc.h*, page 498.
- The no-free heap (`--no_free_heap`) is the smallest possible heap implementation. This heap does not support `free` or `realloc`. See *--no_free_heap*, page 353.

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.

HEAP ALIGNMENT

In 32-bit mode, the heap is aligned to an 8-byte aligned address.

In 64-bit mode, the heap is aligned to a 16-byte aligned address.

For more information about aligning the heap, see *Setting up heap memory*, page 118.

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 linker command line option `--define_symbol`. The linker will create a public absolute constant symbol that the application can use as a label, as a size, as setup for a debugger, etc.
- 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 the linker about the start label of the application. It is used by the linker as a root symbol and to inform the debugger where to start execution.

The following lines illustrate how to use `-D` to create a symbol. If you need to use this mechanism, add these options to your command line like this:

```
--define_symbol NrOfElements=10
--config_def HEAP_SIZE=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 MY_HEAP_SIZE;

/* Setup a heap area with a size defined by an ILINK option */
define block MyHEAP with size = MY_HEAP_SIZE, 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. The value takes the form
of a label.
 */
extern int NrOfElements;

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

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

/* Declare the section that contains the 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) &MY_HEAP_SIZE; ++i)
  {
    p[i] = 0;
  }
  return p;
}
```

Checksum calculation for verifying image integrity

This section contains information about checksum calculation:

- *Briefly about checksum calculation*, page 223
- *Calculating and verifying a checksum*, page 224
- *Troubleshooting checksum calculation*, page 229

For more information, see also *The IAR ELF Tool—ielftool*, page 557.

BRIEFLY ABOUT CHECKSUM CALCULATION

You can use a checksum to verify that the image is the same at runtime as when the image's original checksum was generated. In other words, to verify that the image has not been corrupted.

This works as follows:

- You need an initial checksum.

  You can either use the IAR ELF Tool—`ielftool`—to generate an initial checksum or you might have a third-party checksum available.

- You must generate a second checksum during runtime.

  You can either add specific code to your application source code for calculating a checksum during runtime or you can use some dedicated hardware on your device for calculating a checksum during runtime.

- You must add specific code to your application source code for comparing the two checksums and take an appropriate action if they differ.

  If the two checksums have been calculated in the same way, and if there are no errors in the image, the checksums should be identical. If not, you should first suspect that the two checksums were not generated in the same way.

No matter which solutions you use for generating the two checksum, you must make sure that both checksums are calculated *in the exact same way*. If you use `ielftool` for the initial checksum and use a software-based calculation during runtime, you have full control of the generation for both checksums. However, if you are using a third-party checksum for the initial checksum or some hardware support for the checksum calculation during runtime, there might be additional requirements that you must consider.

For the two checksums, there are some choices that you must always consider and there are some choices to make only if there are additional requirements. Still, all of the details must be the same for both checksums.

Always consider:

- *Checksum range*

  The memory range (or ranges) that you want to verify by means of checksums. Typically, you might want to calculate a checksum for all ROM memory. However, you might want to calculate a checksum only for specific ranges. Remember that:

  - It is OK to have several ranges for one checksum.
  - 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, for example, `0x100-0x1FF,0x400-0x4FF` is not the same as `0x400-0x4FF,0x100-0x1FF`.

- If several checksums are used, you should place them in sections with unique names and use unique symbol names.
- A checksum should never be calculated on a memory range that contains a checksum or a software breakpoint.

- *Algorithm and size of checksum*

  You should consider which algorithm is most suitable in your case. There are two basic choices, Sum—a simple arithmetic algorithm—or CRC—which is the most commonly used algorithm. For CRC there are different sizes to choose for the checksum, 2, 4, or 8 bytes where the predefined polynomials are wide enough to suit the size, for more error detecting power. The predefined polynomials work well for most, but possibly not for all data sets. If not, you can specify your own polynomial. If you just want a decent error detecting mechanism, use the predefined CRC algorithm for your checksum size, typically CRC16 or CRC32.

  Note: For an `n`-bit polynomial, the `n`:th bit is always considered to be set. For a 16-bit polynomial—for example, CRC16—this means that `0x11021` is the same as `0x1021`.

  For more information about selecting an appropriate polynomial for data sets with non-uniform distribution, see for example section 3.5.3 in *Tannenbaum, A.S., Computer Networks, Prentice Hall 1981, ISBN: 0131646990.*

- *Fill*

  Every byte in the checksum range must have a well-defined value before the checksum can be calculated. Typically, bytes with unknown values are *pad bytes* that have been added for alignment. This means that you must specify which fill pattern to be used during calculation, typically `0xFF` or `0x00`.

- *Initial value*

  The checksum must always have an explicit initial value.

In addition to these mandatory details, there might be other details to consider. Typically, this might happen when you have a third-party checksum, you want the checksum be compliant with the Rocksoft™ checksum model, or when you use hardware support for generating a checksum during runtime. `ielftool` also provides support for controlling alignment, complement, bit order, byte order within words, and checksum unit size.

CALCULATING AND VERIFYING A CHECKSUM

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

1. If you are using `ielftool` from the command line, you must first allocate a memory location for the calculated checksum.

Note: If you instead are using the IDE (and not the command line), the `__checksum`, `__checksum_begin`, and `__checksum_end` symbols, and the `.checksum` section are *automatically* allocated when you calculate the checksum, which means that you can skip this step.

You can allocate the memory location in two ways:

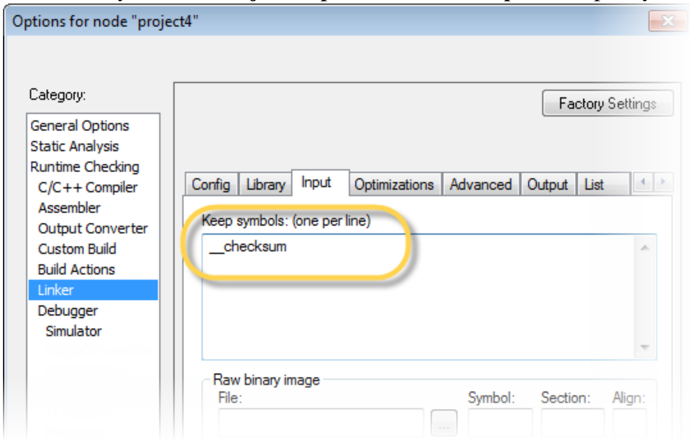
- By creating a global C/C++ or assembler constant symbol with a proper size, residing in a specific section—in this example, `.checksum`
- By using the linker option `--place_holder`.

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

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

2 The `.checksum` section will only be included in your application if the section appears to be needed. If the checksum is not needed by the application itself, use the linker option `--keep=__checksum` (or the linker directive `keep`) to force the section to be included.

Alternatively, choose **Project>Options>Linker>Input** and specify `__checksum`:



3 To control the placement of the `.checksum` section, you must modify the linker configuration file. For example, it can look like this (note the handling of the block `CHECKSUM`):

```
define block CHECKSUM    { ro section .checksum };
place in ROM_region  { ro, first block CHECKSUM };
```

Note: It is possible to skip this step, but in that case the `.checksum` section will automatically be placed with other read-only data.

4 When configuring `ielftool` to calculate a checksum, there are some basic choices to make:

- Checksum algorithm

  Choose which checksum algorithm you want to use. In this example, the CRC16 algorithm is used.

- Memory range

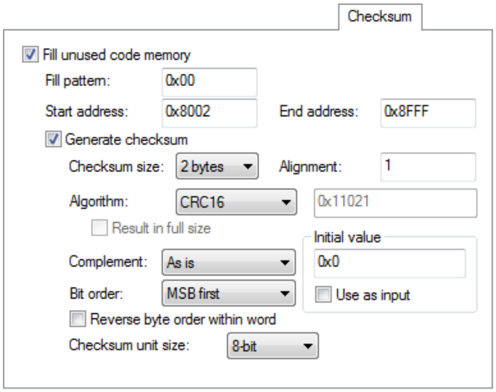
  Using the IDE, you can specify one memory range for which the checksum should be calculated. From the command line, you can specify any ranges.

- Fill pattern

  Specify a fill pattern—typically `0xFF` or `0x00`—for bytes with unknown values. The fill pattern will be used in all checksum ranges.

For more information, see *Briefly about checksum calculation*, page 223.

To run `ielftool` from the IDE, choose **Project>Options>Linker>Checksum** and make your settings, for example:



In the simplest case, you can ignore (or leave with default settings) these options: **Complement**, **Bit order**, **Reverse byte order within word**, and **Checksum unit size**.

To run `ielftool` from the command line, specify the command, for example, like this:

```
ielftool --fill=0x00;0x8002-0x8FFF
--checksum=__checksum:2,crc16;0x8002-0x8FFF sourceFile.out
destinationFile.out
```

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

The checksum will be created later on when you build your project and will be automatically placed in the specified symbol `__checksum` in the section `.checksum`.

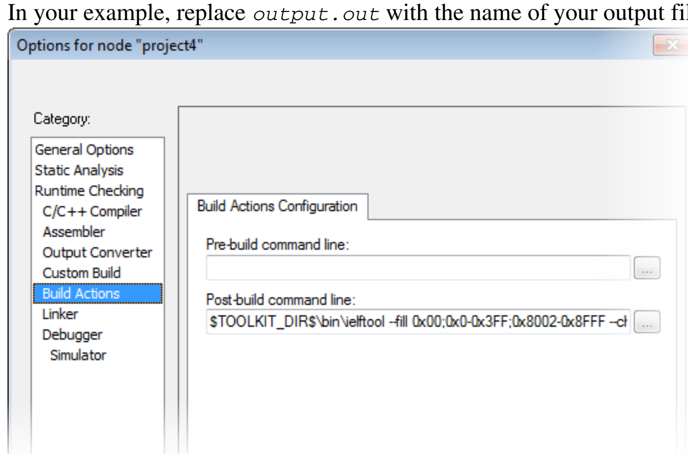
5 You can specify several ranges instead of only one range.

If you are using the IDE, perform these steps:

- Choose **Project>Options>Linker>Checksum** and make sure to deselect **Fill unused code memory**.
- Choose **Project>Options>Build Actions** and specify the ranges together with the rest of the required commands in the **Post-build command line** text field, for example like this:

```
$TOOLKIT_DIR$\bin\ielftool "$TARGET_PATH$" "$TARGET_PATH$"
--fill 0x00;0x0-0x3FF;0x8002-0x8FFF
--checksum=__checksum:2,crc16;0x0-0x3FF;0x8002-0x8FFF
```

In your example, replace *output.out* with the name of your output file.



If you are using the command line, specify the ranges, for example like this:

```
ielftool output.out output.out
--fill 0x00;0x0-0x3FF;0x8002-0x8FFF
--checksum=__checksum:2,crc16;0x0-0x3FF;0x8002-0x8FFF
```

In your example, replace *output.out* with the name of your output file.

6 Add a function for checksum calculation to your source code. Make sure that the function uses the same algorithm and settings as for the checksum calculated by `ielftool`. For example, a slow variant of the crc16 algorithm but with small memory footprint (in contrast to the fast variant that uses more memory):

```
unsigned short SmallCrc16(uint16_t
 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 this checksum algorithm in the `arm\src\linker` directory of your product installation.

7 Make sure that your application also contains a call to the function that calculates the checksum, compares the two checksums, and takes appropriate action if the checksum values do not match.

This code gives an example of how the checksum can be calculated for your application and to be compared with the `ielftool` generated checksum:

```
/* The calculated checksum */

/* Linker generated symbols */
extern unsigned short const __checksum;
extern int __checksum_begin;
extern int __checksum_end;


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

  /* Run the checksum algorithm */
  calc = SmallCrc16(0,
                (unsigned char *) &__checksum_begin,
                ((unsigned char *) &__checksum_end -
                ((unsigned char *) &__checksum_begin)+1));

  /* Fill the end of the byte sequence with zeros. */
  calc = SmallCrc16(calc, zeros, 2);

  /* Test the checksum */
  if (calc != __checksum)
  {
    printf("Incorrect checksum!\n");
    abort();   /* Failure */
  }

  /* Checksum is correct */
}
```

8 Build your application project and download it.

During the build, `ielftool` creates a checksum and places it in the specified symbol `__checksum` in the section `.checksum`.

9 Choose **Download and Debug** to start the C-SPY debugger.

During execution, the checksum calculated by `ielftool` and the checksum calculated by your application should be identical.

TROUBLESHOOTING CHECKSUM CALCULATION

If the two checksums do not match, there are several possible causes. These are some troubleshooting hints:

- If possible, start with a small example when trying to get the checksums to match.

- Verify that the exact same memory range or ranges are used in both checksum calculations.

  To help you do this, `ielftool` lists the ranges for which the checksum is calculated on `stdout` about the exact addresses that were used and the order in which they were accessed.
- Make sure that all checksum symbols are excluded from all checksum calculations.

  Compare the checksum placement with the checksum range and make sure they do not overlap. You can find information in the **Build** message window after `ielftool` has generated a checksum.
- Verify that the checksum calculations use the same polynomial.
- Verify that the bits in the bytes are processed in the same order in both checksum calculations, from the least to the most significant bit or the other way around. You control this with the **Bit order** option (or from the command line, the `-m` parameter of the `--checksum` option).
- If you are using the small variant of CRC, check whether you need to feed additional bytes into the algorithm.

  The number of zeros to add at the end of the byte sequence must match the size of the checksum, in other words, one zero for a 1-byte checksum, two zeros for a 2-byte checksum, four zeros for a 4-byte checksum, and eight zeros for an 8-byte checksum.
- Any breakpoints in flash memory change the content of the flash. This means that the checksum which is calculated by your application will no longer match the initial checksum calculated by `ielftool`. To make the two checksums match again, you must disable all your breakpoints in flash and any breakpoints set in flash by C-SPY internally. The stack plugin and the debugger option **Run to** both require C-SPY to set breakpoints. Read more about possible breakpoint consumers in the *C-SPY® Debugging Guide for Arm*.
- 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 different than the size of an `int`, 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 32-bit Arm architecture
CPPABI	C++ ABI for the 32-bit Arm architecture
AAELF	ELF for the 32-bit Arm architecture
AADWARF	DWARF for the 32-bit Arm architecture
RTABI	Runtime ABI for the 32-bit Arm architecture
CLIBABI	C library ABI for the 32-bit Arm architecture
AAPCS64	Procedure Call Standard for the 64-bit Arm architecture
VFABIA64	Vector function application binary interface specification for the 64-bit Arm architecture
ELF64	ELF for the 64-bit Arm architecture
DWARF64	DWARF for the 64-bit Arm architecture
CPPABI64	C++ ABI for the 64-bit 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:

- The AEABI is specified for C89 only
- The AEABI does not specify C++ library compatibility
- Neither the size of an `enum` or of `wchar_t` is constant in the AEABI.
- 64-bit Arm has no runtime ABI or a C ABI. Therefore, the compiler option `--aeabi` has no effect **in 64-bit mode**.

If AEABI compliance is enabled, 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:

- Include at least one module built with the IAR C/C++ Compiler for Arm.
- 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

Potential incompatibilities include but are not limited to:

- The size of `enum`
- The size of `wchar_t`
- The calling convention
- The instruction set used.

When linking AEABI-compliant modules, also consider the information in the chapters *Linking using ILINK* and *Linking your application*.

LINKING AEABI-COMPLIANT MODULES USING A THIRD-PARTY LINKER

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 232.

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`, `__svc`, `__irq`, `__fiq`, and `__nested`.

ENABLING AEABI COMPLIANCE IN THE COMPILER

You can enable AEABI compliance in the compiler by setting the `--aeabi` option. In this case, you must also use the `--guard_calls` option.

In the IDE, use the **Project>Options>C/C++ Compiler>Extra Options** page to specify the `--aeabi` and `--guard_calls` options.

On the command line, use the options `--aeabi` and `--guard_calls` 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
```

CMSIS integration (32-bit mode)

This mechanism is deprecated and might be removed in future versions. To set up new projects with CMSIS support, use the CMSIS Manager.

The `arm\CMSIS` subdirectory contains CMSIS (Cortex Microcontroller Software Interface Standard) and CMSIS DSP header and library files, and documentation. For more information, see **developer.arm.com/tools-and-software/embedded/cmsis**.

The special header file `inc\c\cmsis_iar.h` is provided as a CMSIS adaptation of the current version of the IAR C/C++ Compiler.

Note: CMSIS is not supported **in 64-bit mode**.

CMSIS DSP LIBRARY

IAR Embedded Workbench comes with prebuilt CMSIS DSP libraries in the `arm\CMSIS\Lib\IAR` directory.

The names of the library files for Armv7-M MCUs are constructed in this way:

```
iar_cortexM{0|3|4|7}{l|b}[s|f]_math.a
```

where `{0|3|4|7}` selects the Cortex-M variant, `{l|b}` selects the byte order, and `[s|f]` indicates that the library is built for a single/double precision FPU (Cortex-M4 and Cortex-M7 only).

The names of the library files for Armv8-M MCUs are constructed in this way:

```
iar_MCUl[d][fsp|fdp]_math.a
```

where *MCU* selects MCU variant (ARMv8MBL (M23) or ARMv8MML (M33/M35P), `l` indicates little-endian, `[d]` selects support for DSP instructions, and `[fsp|fdp]` indicates that the library is built for a single/double precision FPU.

Note: The Armv81 (M55) MCU is not supported by this mechanism.

CUSTOMIZING THE CMSIS DSP LIBRARY

The source code of the CMSIS DSP library is provided in the `arm\CMSIS\DSP_Lib\Source` directory. You can find an IAR Embedded Workbench project which is prepared for building a customized DSP library in the `arm\CMSIS\DSP_Lib\Source\IAR` directory.

BUILDING WITH CMSIS ON THE COMMAND LINE

This section contains examples of how to build your CMSIS-compatible application on the command line.

CMSIS only (that is without the DSP library)

```
iccarm -I $EW_DIR$\arm\CMSIS\Include
```

With the DSP library, for Cortex-M4, little-endian, and with FPU

```
iccarm --endian=little --cpu=Cortex-M4 --fpu=VFPv4_sp -I
$EW_DIR$\arm\CMSIS\Include $EW_DIR$\arm\CMSIS\DSP\Include -D
ARM_MATH_CM4

ilinkarm $EW_DIR$\arm\CMSIS\Lib\IAR\iar_cortexM3l_math.a
```

BUILDING WITH CMSIS IN THE IDE

Choose **Project>Options>General Options>Library Configuration** to enable CMSIS support.

When enabled, CMSIS include paths and the DSP library will automatically be used. For more information, see the *IDE Project Management and Building Guide for Arm*.

Arm TrustZone®

The Arm TrustZone® technology is a System on Chip (SOC) and CPU system-wide approach to security.

Arm TrustZone was introduced in Armv6KZ and is supported also in Armv7-A and Armv8-A. It does not require any specific tool support. Similar capabilities were introduced for Cortex-M as Arm TrustZone for Armv8-M, also known as CMSE

(Cortex-M Security Extension). CMSE does require tool support, and there is a standard interface for development tools that target CMSE. This extension includes two modes of execution—secure and non-secure. It also adds memory protection and instructions for validating memory access and controlled transition between the two modes.

IN 32-BIT MODE

To use TrustZone for Armv8-M, build two separate images—one for secure mode and one for non-secure mode. The secure image can export function entries that can be used by the non-secure image.

The IAR build tools support TrustZone by means of intrinsic functions, linker options, compiler options, predefined preprocessor symbols, extended keywords, and the section `Veneer$$CMSE`.

You can find the data types and utility functions needed for working with TrustZone in the header file `arm_cmse.h`.

The function type attributes `__cmse_nonsecure_call` and `__cmse_nonsecure_entry` add code to clear the used registers when calling from secure code to non-secure code.

The IAR build tools follow the standard interface for development tools targeting Cortex-M Security Extensions (CMSE), with the following exceptions:

- Variadic secure entry functions are not allowed.
- Secure entry functions with parameters or return values that do not fit in registers are not allowed.
- Non-secure calls with parameters or return values that do not fit in registers are not allowed.
- Non-secure calls with parameters or return values in floating-point registers.
- The compiler option `--cmse` requires the architecture Armv8-M with security extensions, and is not supported when building ROPI (read-only position-independent) images or RWPI (read-write position-independent) images.

For more information about Arm TrustZone, see **www.arm.com**.

An example using the Armv8-M Security Extensions (CMSE)

In the `arm\src\ARMv8M_Secure` directory, you can find an example project that demonstrates the use of Arm TrustZone and CMSE.

The example consists of two projects:

- `hello_s`: The secure part of the application
- `hello_ns`: The non-secure part of the application

Note: You must build the secure project before building the non-secure project.

There are two entry functions in `hello_s`, available to `hello_ns` via secure gateways in a non-secure callable region:

- `secure_hello`: Prints a greeting, in the style of the classic `Hello world` example.
- `register_secure_goodbye`: A callback that returns a string printed on exiting the secure part.

The linker will automatically generate the code for the needed secure gateways and place them in the section `Veneers$$CMSE`.

To set up and build the example:

1. Open the example workspace `hello_s.eww` located in `arm\src\ARMv8M_Secure\Hello_Secure`.
2. Set up the project `hello_s` to run in secure mode by choosing **Project>Options>General Options>32-bit** and then selecting the options **TrustZone** and **Mode: Secure**.
3. Set up the project `hello_ns` to run in non-secure mode by choosing **Project>Options>General Options>32-bit** and then selecting the options **TrustZone** and **Mode: Non-secure**.

   The non-secure part must populate a small vector at `0x200000` with addresses to the initialization routine, non-secure top of stack, and non-secure `main`. This vector is used by the secure part to set up and interact with the non-secure part. In this example, this is done with the following code in `nonsecure_hello.c`:

```
/* Interface towards the secure part */
#pragma location=NON_SECURE_ENTRY_TABLE
__root const non_secure_init_t init_table =
{
  __iar_data_init3,         /* initialization function */
  __section_end("CSTACK"),  /* non-secure stack */
  main_ns                   /* non-secure main */
};
```

4. When the secure project is built, the linker will automatically generate an import library file for the non-secure part that only includes references to functions in the secure part that can be called from the non-secure part. Specify this file by using **Project>Options>Linker>Output>TrustZone import library**.
5. Build the secure project.

6 Include the TrustZone import library file manually in the project `hello_ns` by specifying an additional library: **Project>Options>Linker>Library>Additional libraries**.

7 Build the non-secure project.

8 The secure project must specify the non-secure project output file as an extra image that should be loaded by the debugger. To do this, use **Project>Options>Debugger>Images>Download extra images**.

To debug the example:

1 To debug in the simulator, set the `hello_s` project as the active project by right-clicking on the project and choosing **Set as Active**.

2 Choose **Project>Options>Debugger>Driver** and select **Simulator**.

3 Choose **Simulator>Memory Configuration**. Make sure that the option **Use ranges based on** is deselected.

4 Select **Use manual ranges** and add the following new ranges:

Access type	Start address	End address
RAM	`0x0000'0000`	`0x003F'FFFF`
RAM	`0x2000'0000`	`0x203F'FFFF`
SFR	`0x4000'0000`	`0x5FFF'FFFF`
SFR	`0xE000'0000`	`0xE00F'FFFF`

Table 25: Memory ranges for TrustZone example

5 Click **OK** to close the **Memory Configuration** dialog box.

6 Start C-SPY by choosing **Project>Download and Debug**.

7 Choose **View>Terminal I/O** to open the **Terminal I/O** window.

8 Choose **Debug>Go** to start the execution.

9 The **Terminal I/O** window should now print this text:

```
Hello from secure World!
Hello from non-secure World!
Goodbye, for now.
```

IN 64-BIT MODE

TrustZone support is automatic **in 64-bit mode**.

Patching symbol definitions using $Super$$ and $Sub$$

Using the `$Sub$$` and `$Super$$` special patterns, you can patch existing symbol definitions in situations where you would otherwise not be able to modify the symbol, for example, when a symbol is located in an external library or in ROM code.

The `$Super$$` special pattern identifies the original unpatched function used for calling the original function directly.

The `$Sub$$` special pattern identifies the new function that is called instead of the original function. You can use the `$Sub$$` special pattern to add processing before or after the original function.

AN EXAMPLE USING THE $SUPER$$ AND $SUB$$ PATTERNS

The following example shows how to use the `$Super$$` and `$Sub$$` patterns to insert a call to the function `ExtraFunc()` before the call to the legacy function `foo()`.

```
extern void ExtraFunc(void);
extern void $Super$$foo(void);

/* this function is called instead of the original foo() */
void $Sub$$foo(void)
{
  ExtraFunc();    /* does some extra setup work */
  $Super$$foo();  /* calls the original foo() function */
                  /* To avoid calling the original foo() function
                   * omit the $Super$$foo(); function call.
                   */
}
```

Efficient coding for embedded applications

- Selecting data types
- Controlling data and function placement in memory
- Controlling compiler optimizations
- Facilitating good code generation

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.
- **In 32-bit mode**, 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 co-processor is 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 information 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 inefficient, both in terms of code size and execution speed. Therefore, you should consider replacing code that uses floating-point operations with code that uses integers, because these are more efficient.

The compiler supports three floating-point formats—16, 32, and 64 bits. The 32-bit floating-point type `float` is more efficient in terms of code size and execution speed. The 64-bit format `double` supports higher precision and larger numbers. The 16-bit format is mainly useful for some specific situations.

In the compiler, the floating-point type `float` always uses the 32-bit format, and the type `double` always uses the 64-bit format.

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`:

```
double 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:

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

For more information about floating-point types, see *Basic data types—floating-point types*, page 374.

ALIGNMENT OF ELEMENTS IN A STRUCTURE

Some Arm cores require that when accessing data in memory, the data 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 situations when this can be a problem:

- There are 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 367.

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.

Alternatively, 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 more information about the `#pragma pack` directive, see *pack*, page 418.

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.

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;



/* The variables are used here. */
void Test(void)
{
  IOPORT = 0;
  way = 1;
  out = 1;
}
```

This declares an I/O register byte `IOPORT` at address `0x1000`. The I/O register has 2 bits declared, `Way` and `Out`–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 and know which one is best suited for different situations. You can use:

- The `@` operator and the `#pragma location` directive for absolute placement.

  Using the `@` operator or the `#pragma location` directive, you can place individual global and static variables at absolute addresses. Note that it is not possible to use this notation for absolute placement of individual functions. For more information, see *Data placement at an absolute location*, page 243.

- The @ operator and the #pragma location directive for section placement.

  Using the @ operator or the #pragma location directive, you can place individual functions, variables, and constants in named sections. The placement of these sections can then be controlled by linker directives. For more information, see *Data and function placement in sections*, page 244.
- The @ operator and the #pragma location directive for register placement

  Use the @ operator or the #pragma location directive to place individual global and static variables in registers. The variables must be declared __no_init. This is useful for individual data objects that must be located in a specific register.
- Using the --section option, you can set the default segment for functions, variables, and constants in a particular module. For more information, see *--section*, page 318.

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.

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: All declarations of __no_init variables placed at an absolute address are *tentative definitions*. Tentatively defined variables are only kept in the output from the compiler if they are needed in the module being compiled. Such variables will be defined in all modules in which they are used, which will work as long as they are defined in the same way. The recommendation is to place all such declarations in header files that are included in all modules that use the variables.

Other variables placed at an absolute address use the normal distinction between declaration and definition. For these variables, you must provide the definition in only one module, normally with an initializer. Other modules can refer to the variable by using an extern declaration, with or without an explicit address.

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 @ 0xFF2000;/* OK */
```

The next example contains two const declared objects. The first one is not initialized, and the second one is initialized to a specific value. (The first case is useful for configuration parameters, because they are accessible from an external interface.) Both objects are placed in ROM. Note that in the second case, the compiler is not obliged to actually read from the variable, because the value is known.

```
#pragma location=0xFF2002
__no_init const int beta;                /* OK */

const int gamma @ 0xFF2004 = 3;          /* OK */
```

In the first case, the value is not initialized by the compiler—the value must be set by other means. The typical use is for configurations where the values are loaded to ROM separately, or for special function registers that are read-only.

```
__no_init int epsilon @ 0xFF2007;  /* 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 method 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 examples, a data object is placed in a user-defined section. Note that you must as always ensure that the section is placed in the appropriate memory area when linking.

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

#pragma location="MY_CONSTANTS"
const int beta = 42;                    /* OK */

const int gamma @ "MY_CONSTANTS" = 17;  /* OK */
int theta @ "MY_ZEROS";                 /* OK */
int phi @ "MY_INITED" = 4711;           /* OK */
```

The linker will normally arrange for the correct type of initialization for each variable. If you want to control or suppress automatic initialization, you can use the `initialize` and `do not initialize` directives in the linker configuration file.

Examples of placing functions in named sections

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

void g(void) @ "MY_FUNCTIONS"
{
}

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

DATA PLACEMENT IN REGISTERS (32-BIT MODE)

In 32-bit mode, the `@` operator, alternatively the `#pragma location` directive, can be used for placing global and static variables in a register.

To place a variable in a register, the argument to the `@` operator and the `#pragma location` directive should be an identifier that corresponds to an Arm core register in the range R4–R11 (R9 cannot be specified in combination with the --rwpi command line option).

A variable can be placed in a register only if it is declared as __no_init, has file scope, and its size is four bytes. A variable placed in a register does not have a memory address, so the address operator & cannot be used.

Within a module where a variable is placed in a register, the specified register will only be used for accessing that variable. The value of the variable is preserved across function calls to other modules because the registers R4–R11 are callee saved, and as such they are restored when execution returns. However, the value of a variable placed in a register is not always preserved as expected:

- In an exception handler or library callback routine (such as the comparator function passed to qsort) the value might not be preserved. The value will be preserved if the command line option --lock_regs is used for locking the register in all modules of the application, including library modules.
- In a fast interrupt handler, the value of a variable in R8–R11 is not preserved from outside the handler, because these registers are banked.
- The longjmp function and C++ exceptions might restore variables placed in registers to old values, unlike other variables with static storage duration which are not restored.

The linker does not prevent modules from placing different variables in the same register. Variables in different modules can be placed in the same register, and another module could use the register for other purposes.

Note: A variable placed in a register should be defined in an include file, to be included in every module that uses the variable. An unused definition in a module will cause the register to not be used in that module.

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 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. See *optimize*, page 417, 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 301.

Note: Only one object file is generated, and therefore all symbols will be part of that object file.

If the whole application is compiled as one compilation unit, it is 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 291.

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 Dead code elimination Redundant label elimination Redundant branch elimination
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 Common subexpression elimination Code motion Static clustering
High (Balanced)	Same as above, and: Instruction scheduling Cross jumping Advanced register content analysis and optimization Loop unrolling Function inlining Type-based alias analysis

Table 26: Compiler optimization levels

Note: Some of the performed optimizations can be individually enabled or disabled. For more information, see *Fine-tuning enabled transformations*, page 249.

A high level of optimization might result in increased compile time, and will also most likely 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, or even occasionally display an incorrect value. 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.

If you use the optimization level High speed, the `--no_size_constraints` compiler option relaxes the normal restrictions for code size expansion and enables more aggressive optimizations.

You can choose an optimization goal for each module, or even individual functions, using command line options and pragma directives (see *-O*, page 311 and *optimize*, page 417). For a small embedded application, this makes it possible to achieve acceptable speed performance while minimizing the code size: Typically, only a few places in the application need to be fast, such as the most frequently executed inner loops, or the interrupt handlers.

Rather than compiling the whole application with High (Balanced) optimization, you can use High (Size) in general, but override this to get High (Speed) optimization only for those functions where the application needs to be fast.

Note: Because of the unpredictable way in which different optimizations interact, where one optimization can enable other optimizations, sometimes a function becomes smaller when compiled with High (Speed) optimization than if High (Size) is used. Also, using multi-file compilation (see *--mfc*, page 301) can enable many optimizations to improve both speed and size performance. It is recommended that you experiment with different optimization settings so that you can pick the best ones for your project.

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
- Vectorization

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.

For more information about the command line option, see *--no_cse*, page 303.

Loop unrolling

Loop unrolling means that the code body of a loop, whose number of iterations can be determined at compile time, is duplicated. Loop unrolling reduces the loop overhead by amortizing it over several iterations.

This optimization is most efficient for smaller loops, where the loop overhead can be a substantial part of the total loop body.

Loop unrolling, 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. Only relatively small loops where the loop overhead reduction is noticeable will be unrolled. 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 disable loop unrolling, use the command line option `--no_unroll`, see *--no_unroll*, page 310.

Function inlining

Function inlining means that a 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 normally reduces execution time, but might increase the code size.

For more information, see *Inlining functions*, page 90.

To disable function inlining, use the command line option `--no_inline`, see *--no_inline*, page 304.

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 Medium and above, 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 below Medium.

For more information about the command line option, see --*no_code_motion*, page 302.

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 application code conforming to standard C or C++ application code, this optimization can reduce code size and execution time. However, non-standard 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.

For more information about the command line option, see --*no_tbaa*, page 308.

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. Therefore, 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.

For more information about the command line option, see *--no_clustering*, page 302.

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.

For more information about the command line option, see *--no_scheduling*, page 307.

Vectorization

Vectorization transforms sequential loops into NEON vector operations, without the need to write assembler code or use intrinsic functions. This enhances portability. Loops will only be vectorized if the target processor has NEON capability and auto-vectorization is enabled. Auto-vectorization is not supported **in 64-bit mode**.

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

Note: This option has no effect at optimization levels None, Low, and Medium, or for High Balanced or High Size. To disable vectorization for individual functions, use one of the pragma directives `optimize` or `vectorize`, see *optimize*, page 417 and *vectorize*, page 426.

For information about the command line option, see *--vectorize*, page 324.

Facilitating good code generation

This section contains hints on how to help the compiler generate good code:

- *Writing optimization-friendly source code*, page 253
- *Saving stack space and RAM memory*, page 253
- *Function prototypes*, page 253
- *Integer types and bit negation*, page 254
- *Protecting simultaneously accessed variables*, page 255
- *Accessing special function registers*, page 255
- *Passing values between C and assembler objects*, page 257
- *Non-initialized variables*, page 257

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 therefore 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 (non-static). Also, avoid taking the address of frequently accessed static variables.
- The compiler is capable of inlining functions, see *Function inlining*, page 250. To maximize the effect of the inlining transformation, it is good practice to place the definitions of small functions called from more than one module in the header file rather than in the implementation file. Alternatively, you can use multi-file compilation. For more information, see *Multi-file compilation units*, page 247.
- Avoid using inline assembler without operands and clobbered resources. Instead, use SFRs or intrinsic functions if available. Otherwise, use inline assembler *with* operands and clobbered resources or write a separate module in assembler language. For more information, see *Mixing C and assembler*, page 171.

SAVING STACK SPACE AND RAM MEMORY

The following is a list of programming techniques that 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 valid C, however it is strongly recommended to use the prototyped style, and provide a prototype declaration for each public function in a header that is included both in the compilation unit defining the function and in all compilation units using it.

The compiler will not perform type checking on parameters passed to functions declared using K&R style. Using prototype declarations will also result in more efficient code in some cases, as there is no need for type promotion for these functions.

To make the compiler require that all function definitions use the prototyped style, and that all public functions have been declared before being defined, use the **Project>Options>C/C++ Compiler>Language 1>Require prototypes** compiler option (`--require_prototypes`).

Prototyped style

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

```
int Test(char, int); /* Declaration */

int Test(char ch, int i) /* Definition */
{
  return i + ch;
}
```

Kernighan & Ritchie style

In K&R style—pre-Standard 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 in the range 0–255, which can never be equal to `0xFFFFFF7F`. Furthermore, it 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.

For more information about the `volatile` type qualifier and the rules for accessing volatile objects, see *Declaring objects volatile*, page 380.

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`:

```
__no_init volatile union
{
  unsigned short mwctl2;
  struct
  {
    unsigned short edr: 1;
    unsigned short edw: 1;
    unsigned short lee: 2;
    unsigned short lemd: 2;
    unsigned short lepl: 2;
  } mwctl2bit;
} @ 0x1000;


/* By including the appropriate include file in your code,
 * it is possible to access either the whole register or any
 * individual bit (or bitfields) from C code as follows.
 */

void Test()
{
  /* Whole register access */
  mwctl2 = 0x1234;

  /* Bitfield accesses */
  mwctl2bit.edw  = 1;
  mwctl2bit.lepl = 3;
}
```

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

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:

```
static unsigned long get_APSR( void )
{
  unsigned long value;
  asm volatile( "MRS %0, APSR" : "=r"(value) );
  return value;
}

static void set_APSR( unsigned long value)
{
  asm volatile( "MSR APSR, %0" :: "r"(value) );
}
```

The general purpose register is used for getting and setting the value of the special purpose register `APSR`. The same method can also be used for accessing other special purpose registers and specific instructions.

To read more about inline assembler, see *Inline assembler*, page 172.

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 more information, see *__no_init*, page 393.

Note: To use this keyword, language extensions must be enabled, see *-e*, page 293. For more information, see *object_attribute*, page 416.

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
- Extended keywords
- Pragma directives
- Intrinsic functions
- The preprocessor
- C/C++ standard library functions
- The linker configuration file
- Section reference
- The stack usage control file
- IAR utilities
- Implementation-defined behavior for Standard C++
- Implementation-defined behavior for Standard C
- Implementation-defined behavior for C89

External interface details

- Invocation syntax
- Include file search procedure
- Compiler output
- ILINK output
- Text encodings
- Reserved identifiers
- Diagnostics

Invocation syntax

You can use the compiler and linker either from the IDE or from the command line. See the *IDE Project Management and Building 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 as 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 261.
- Via environment variables

  The compiler and linker automatically append the value of the environment variables to every command line, see *Environment variables*, page 263.
- Via a text file, using the `-f` option, see *-f*, page 295.

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

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:\my_programs\embedded workbench 8.`*n*`\arm\inc;c:\headers`
`QCCARM`	Specifies command line options, for example: `QCCARM=-lA asm.lst`

Table 27: 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 28: ILINK environment variables

Include file search procedure

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

- The string found between the `""` and `<>` in the `#include` directive is used verbatim as a source file name.
- If the name of the `#include` file is an absolute path specified in angle brackets or double quotes, 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 298.
  2. The directories specified using the `C_INCLUDE` environment variable, if any, see *Environment variables*, page 263.
  3. The automatically set up library system include directories. See *--dlib_config*, page 291.

- 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.

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

For more information, see *Overview of the preprocessor*, page 471.

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 298. 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. The file will have the filename extension `i`, by default.

- Diagnostic messages

  Diagnostic messages are directed to the standard error stream and displayed on the screen, and printed in an optional list file. For more information about diagnostic messages, see *Diagnostics*, page 268.

- 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 265.

- 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 29: 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 350. 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. For more information about diagnostic messages, see *Diagnostics*, page 268.
- Error return codes

  ILINK returns status information to the operating system which can be tested in a batch file, see *Error return codes*, page 265.
- 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.
- An import library for use when building a non-secure image, a relocatable ELF object module containing symbols and their addresses. See the linker option *--import_cmse_lib_out*, page 347.

Text encodings

Text files read or written by IAR tools can use a variety of text encodings:

- Raw

  This is a backward-compatibility mode for C/C++ source files. Only 7-bit ASCII characters can be used in symbol names. Other characters can only be used in comments, literals, etc. This is the default source file encoding if there is no Byte Order Mark (BOM).

- The system default locale

  The locale that you have configured your Windows OS to use.

- UTF-8

  Unicode encoded as a sequence of 8-bit bytes, with or without a Byte Order Mark.

- UTF-16

  Unicode encoded as a sequence of 16-bit words using a big-endian or little-endian representation. These files always start with a Byte Order Mark.

In any encoding other than Raw, you can use Unicode characters of the appropriate kind (alphabetic, numeric, etc) in the names of symbols.

When an IAR tool reads a text file with a Byte Order Mark, it will use the appropriate Unicode encoding, regardless of the any options set for input file encoding.

For source files without a Byte Order Mark, the compiler will use the Raw encoding, unless you specify the compiler option `--source_encoding`. See *--source_encoding*, page 320.

For source files without a Byte Order Mark, the assembler will use the Raw encoding unless you specify the assembler option `--source_encoding`.

For other text input files, like the extended command line (`.xcl` files), without a Byte Order Mark, the IAR tools will use the system default locale unless you specify the compiler option `--utf8_text_in`, in which case UTF-8 will be used. See *--utf8_text_in*, page 324.

For compiler list files and preprocessor output, the same encoding as the main source file will be used by default. Other tools that generate text output will use the UTF-8 encoding by default. You can change this by using the compiler options `--text_out` and `--no_bom`. See *--text_out*, page 321 and *--no_bom*, page 301.

CHARACTERS AND STRING LITERALS

When you compile source code, characters (*x*) and string literals (*xx*) are handled as follows:

`'x'`, `"xx"`	Characters in untyped character and string literals are copied verbatim, using the same encoding as in the source file.
`u8"xx"`	Characters in UTF-8 string literals are converted to UTF-8.
`u'x'`, `u"xx"`	Characters in UTF-16 character and string literals are converted to UTF-16.

`U'x',U"xx"`	Characters in UTF-32 character and string literals are converted to UTF-32.
`L'x',L"xx"`	Characters in wide character and string literals are converted to UTF-32.

Reserved identifiers

Some identifiers are reserved for use by the implementation. Some of the more important identifiers that the C/C++ standards reserve for any use are:

- Identifiers that contain a double underscore (`__`)
- Identifiers that begin with an underscore followed by an uppercase letter

In addition to this, the IAR tools reserve for any use:

- Identifiers that contain a double dollar sign (`$$`)
- Identifiers that contain a question mark (`?`)

More specific reservations are in effect in particular circumstances, see the C/C++ standards for more information.

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:

level[*tag*]: *message filename linenumber* 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
filename	The name of the source file in which the issue was encountered
linenumber	The line number at which the compiler detected the issue

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 and 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 316.

Warning

A diagnostic message that is produced when the compiler or linker finds a potential problem which is of concern, but which does not prevent completion of the compilation or linking. Warnings can be disabled by use of the command line option `--no_warnings`, see *--no_warnings*, page 311.

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 produced when the compiler finds a condition that not only prevents code generation, but also 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.

For information about the compiler options that are available for setting severity levels, see the chapter *Compiler options*.

For information about the pragma directives that are available for setting severity levels for the compiler, see the chapter *Pragma directives*.

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

- Options syntax
- Summary of compiler options
- Descriptions of compiler options

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.

See the online help system 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 262.

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 (=), like this:

`--example_option=value`

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 take a file 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
`--aarch64`	Generates code using the A64 instruction set
`--abi`	Specifies a data model for generating code using the A64 instruction set
`--aeabi`	Enables AEABI-compliant code generation

Table 30: Compiler options summary

Command line option	Description
--align_sp_on_irq	Generates code to align SP on entry to __irq functions
--arm	Sets the default function mode to Arm
--c89	Specifies the C89 dialect
--char_is_signed	Treats char as signed
--char_is_unsigned	Treats char as unsigned
--cmse	Enables CMSE secure object generation
--cpu	Specifies a processor variant
--cpu_mode	Specifies the default CPU mode for functions
--c++	Specifies Standard C++
-D	Defines preprocessor symbols
--debug	Generates debug information
--dependencies	Lists file dependencies
--deprecated_feature_warnings	Enables/disables warnings for deprecated features
--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	Uses the system include files for the DLIB library and determines which configuration of the library to use
--do_explicit_zero_opt_in_named_sections	For user-named sections, treats explicit initializations to zero as zero initializations
-e	Enables language extensions
--enable_hardware_workaround	Enables a specific hardware workaround
--enable_restrict	Enables the Standard C keyword restrict
--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

Table 30: Compiler options summary (Continued)

Command line option	Description
-f	Extends the command line
--f	Extends the command line, optionally with a dependency.
--fpu	Selects the type of floating-point unit
--generate_entries_without_bounds	Generates extra functions for use from non-instrumented code. See the C-RUN documentation in the *C-SPY® Debugging Guide for Arm*.
--guard_calls	Enables guards for function static variable initialization
--header_context	Lists all referred source files and header files
-I	Specifies include file path
--ignore_uninstrumented_pointers	Disables checking of accesses via pointers from non-instrumented code. See the C-RUN documentation in the *C-SPY® Debugging Guide for Arm*.
-l	Creates a list file
--lock_regs	Prevents the compiler from using specified registers
--macro_positions_in_diagnostics	Obtains positions inside macros in diagnostic messages
--make_all_definitions_weak	Turns all variable and function definitions into weak definitions.
--max_cost_constexpr_call	Specifies the limit for constexpr evaluation cost
--max_depth_constexpr_call	Specifies the limit for constexpr recursion depth
--mfc	Enables multi-file compilation
--no_alignment_reduction	Disables alignment reduction for simple Thumb functions
--no_bom	Omits the Byte Order Mark for UTF-8 output files
--no_call_frame_info	Disables output of call frame information
--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

Table 30: Compiler options summary (Continued)

Command line option	Description
--no_default_fp_contract	Sets the default value for STDC FP_CONTRACT to OFF.
--no_exceptions	Disables C++ exception support
--no_fragments	Disables section fragment handling
--no_inline	Disables function inlining
--no_literal_pool	Generates code that should run from a memory region where it is not allowed to read data, only to execute code
--no_loop_align	Disables the alignment of labels in loops
--no_mem_idioms	Makes the compiler not optimize certain memory access patterns
--no_path_in_file_macros	Removes the path from the return value of the symbols __FILE__ and __BASE_FILE__
--no_rtti	Disables C++ RTTI support
--no_rw_dynamic_init	Disables runtime initialization of static C variables.
--no_scheduling	Disables the instruction scheduler
--no_size_constraints	Relaxes the normal restrictions for code size expansion when optimizing for speed.
--no_static_destruction	Disables destruction of C++ static variables at program exit
--no_system_include	Disables the automatic search for system include files
--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_uniform_attribute_syntax	Specifies the default syntax rules for IAR type attributes
--no_unroll	Disables loop unrolling
--no_var_align	Aligns variable objects based on the alignment of their type.
--no_warnings	Disables all warnings
--no_wrap_diagnostics	Disables wrapping of diagnostic messages

Table 30: Compiler options summary (Continued)

Command line option	Description
--nonportable_path_warnings	Generates a warning when the path used for opening a source header file is not in the same case as the path in the file system.
-O	Sets the optimization level
-o	Sets the object filename. Alias for --output.
--only_stdout	Uses standard output only
--output	Sets the object filename
--pending_instantiations	Sets the maximum number of instantiations of a given C++ template.
--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. Alias for --debug.
--relaxed_fp	Relaxes the rules for optimizing floating-point expressions
--remarks	Enables remarks
--require_prototypes	Verifies that functions are declared before they are defined
--ropi	Generates code that uses PC-relative references to address code and read-only data.
--ropi_cb	Makes all accesses to constant data, base-addressed relative to the register R8
--runtime_checking	Enables runtime error checking. See the C-RUN documentation in the *C-SPY® Debugging Guide for Arm*.
--rwpi	Generates code that uses an offset from the static base register to address-writable data.
--rwpi_near	Generates code that uses an offset from the static base register to address-writable data. Addresses max 64 Kbytes of memory.
--section	Changes a section name
--section_prefix	Adds a prefix to section names
--silent	Sets silent operation

Table 30: Compiler options summary (Continued)

Command line option	Description
--source_encoding	Specifies the encoding for source files
--stack_protection	Enables stack protection
--strict	Checks for strict compliance with Standard C/C++
--system_include_dir	Specifies the path for system include files
--text_out	Specifies the encoding for text output files
--thumb	Sets default function mode to Thumb
--uniform_attribute_syntax	Specifies the same syntax rules for IAR type attributes as for const and volatile
--use_c++_inline	Use C++ inline semantics in C
--use_paths_as_written	Use paths as written in debug information
--use_unix_directory_separators	Uses / as directory separator in paths
--utf8_text_in	Uses the UTF-8 encoding for text input files
--vectorize	Enables generation of NEON vector instructions
--version	Sends compiler output to the console and then exits.
--vla	Enables VLA support
--warn_about_c_style_casts	Makes the compiler warn when C-style casts are used in C++ source code
--warnings_affect_exit_code	Warnings affect exit code
--warnings_are_errors	Warnings are treated as errors

Table 30: Compiler options summary (Continued)

Descriptions of compiler options

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

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 software FPU is selected.
`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 unit is used.

Description

Use this option to specify the floating-point calling convention. **In 64-bit mode**, this option has no effect.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--aarch64

Syntax

`--aarch64`

Description

Use this option to generate code using the A64 instruction set in the AArch64 state for the assembler directive `CODE`.

Note: This option has the same effect as the `--cpu_mode=aarch64` option.

See also

--*abi*, page 281 and --*cpu_mode*, page 286.

To set this option, use **Project>Options>General Options>Target>Execution mode**

--abi

Syntax

`--abi={ilp32|lp64}`

Parameters

`ilp32`	Generates A64 code for the ILP32 data model. Defines the symbol `__ILP32__`
`lp64`	Generates A64 code for the LP64 data model. Defines the symbol `__LP64__`.

Description

Use this option to specify a data model for the generation of code using the A64 instruction set in the AArch64 state.

See also

--aarch64, page 281 and *--cpu_mode*, page 286.

To set related options, choose:

Project>Options>General Options>Target>Execution mode

and

Project>Options>General Options>64-bit>Data model

--aeabi

Syntax

```
--aeabi
```

Description

Use this option to generate AEABI-compliant object code. **In 64-bit mode**, this option has no effect.

Note: This option must be used together with the `--guard_calls` option.

Note: This option cannot be used together with C++ header files.

See also

AEABI compliance, page 230 and *--guard_calls*, page 297.

To set this option, use **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. **In 64-bit mode**, this option has no effect.

This is especially useful for nested interrupts, where the interrupted code uses the same `SP` as the interrupt handler. This means that the stack might only have 4-byte alignment, instead of the 8-byte alignment required by AEABI (and some instructions generated by the compiler for some cores).

See also

__irq, page 390.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--arm

Syntax
`--arm`

Description
Use this option to set default function mode to Arm (A32 in 32-bit mode). **In 64-bit mode**, this option has no effect.

Note: This option has the same effect as the `--cpu_mode=arm` option.

Project>Options>C/C++ Compiler>Code>Processor mode>Arm

--c89

Syntax
`--c89`

Description
Use this option to enable the C89 C dialect instead of Standard C.

See also
C language overview, page 199.

Project>Options>C/C++ Compiler>Language 1>C dialect>C89

--char_is_signed

Syntax
`--char_is_signed`

Description
By default, the compiler interprets the plain `char` type as unsigned. Use this option to make the compiler interpret the plain `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 and cannot be used with code that is compiled with this option.

Project>Options>C/C++ Compiler>Language 2>Plain 'char' is

--char_is_unsigned

Syntax
`--char_is_unsigned`

Description
Use this option to make the compiler interpret the plain `char` type as unsigned. This is the default interpretation of the plain `char` type.

Project>Options>C/C++ Compiler>Language 2>Plain 'char' is

--cmse

Syntax

`--cmse`

Description

This option enables language extensions for TrustZone for Armv8-M. **In 64-bit mode**, this option has no effect. Use this option for object files that are to be linked in a secure image. The option allows the use of instructions, keywords, and types that are not available for non-secure code:

- The function attributes `_cmse_nonsecure_call` and `_cmse_nonsecure_entry`.
- The functions for CMSE have names with the prefix `cmse_`, and are defined in the header file `arm_cmse.h`.

Note: To use this option, you must first select the option **Project>Options>General Options>32-bit>TrustZone**.

See also

Arm TrustZone®, page 234 and *ARMv8-M Security Extensions: Requirements on Development Tools* (ARM-ECM-0359818).

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--cpu

Syntax

`--cpu={core|list}`

Parameters

core	Specifies a specific processor variant
`list`	Lists all supported values for the option `--cpu`

Description

Use this option to select the architecture or processor variant for which the code is to be generated.

The default core is Cortex-M3.

In 32-bit mode: Some of the supported values for the `--cpu` option are:

`6-M`	`7-A`	`7E-M`
`7-M`	`7-R`	`7-S`

`8-A.AArch32`	`8-M.baseline`	`8-M.mainline`
`8-R.AArch32`	`Cortex-A5`	`Cortex-A7`
`Cortex-A8`	`Cortex-A9`	`Cortex-A12`
`Cortex-A15`	`Cortex-A17`	`Cortex-M0`
`Cortex-M0+`	`Cortex-M23`	`Cortex-M23.no_se` (core without support for TrustZone)
`Cortex-M3`	`Cortex-M33`	`Cortex-M33.no_dsp` (core without integer DSP extension)
`Cortex-M33.fp` (floating-point unit with support for single precision)	`Cortex-M33.no_se` (core without support for TrustZone)	`Cortex-M4`
`Cortex-M4F`	`Cortex-M55`	`Cortex-M7`
`Cortex-M7.fp.dp` (floating-point unit with support for double precision)	`Cortex-M7.fp.sp` (floating-point unit with support for single precision)	`Cortex-R4`
`Cortex-R5`	`Cortex-R52`	`Cortex-R52.no_neon`
`Cortex-R7`		

In 64-bit mode: Some of the supported values for the `--cpu` option are:

`8-a.AArch64`	`Cortex-A35`	`Cortex-A53`
`Cortex-A55`	`Cortex-A57`	`Cortex-A72`

See also

Processor variant, page 70.

Project>Options>General Options>Target>Processor variant

--cpu_mode

Syntax

```
--cpu_mode={arm|a|thumb|t|a64}
```

Parameters

`arm, a` (default)	Selects the A32 instruction set in 32-bit mode.
`thumb, t`	Selects the T32 or T instruction set in 32-bit mode.
`a64`	Selects the A64 instruction set in 64-bit mode.

Description

Use this option to select the default mode for functions.

See also

--*aarch64*, page 281.

Project>Options>General Options>Target>Processor variant

--c++

Syntax

```
--c++
```

Description

By default, the language supported by the compiler is C. If you use Standard C++, you must use this option to set the language the compiler uses to C++.

See also

Using C++, page 207.

Project>Options>C/C++ Compiler>Language 1>C++

-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:

`-D`*`symbol`*

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|n][s]] {`*`filename`*`|`*`directory`*`|+}`

Parameters

`i` (default)	Lists only the names of files
`m`	Lists in makefile style (multiple rules)
`n`	Lists in makefile style (one rule)
`s`	Suppresses system files
`+`	Gives the same output as `-o`, but with the filename extension `d`

See also *Rules for specifying a filename or directory as parameters*, page 274.

Description

Use this option to make the compiler list the names of all source and header files opened for input into a file with the default filename extension `i`.

Example

If `--dependencies` or `--dependencies=i` is used, the name of each opened input 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 is in makefile style. For each input 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 an input 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.

--deprecated_feature_warnings

Syntax

`--deprecated_feature_warnings=[+|-]`*feature*`[,[+|-]`*feature*`,...]`

Parameters

feature	A feature can be `attribute_syntax` or `segment_pragmas`.

Description

Use this option to disable or enable warnings for the use of a deprecated feature. The deprecated features are:

- `attribute_syntax`

  See *--uniform_attribute_syntax*, page 322, *--no_uniform_attribute_syntax*, page 310, and *Syntax for type attributes used on data objects*, page 384.

- `segment_pragmas`

  See the pragma directives `dataseg`, `constseg`, and `memory`. Use the `#pragma location` and `#pragma default_variable_attributes` directives instead.

Because the deprecated features will be removed in a future version of the IAR C/C++ compiler, it is prudent to remove the use of them in your source code. To do this, enable warnings for a deprecated feature. For each warning, rewrite your code so that the deprecated feature is no longer used.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--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 See *Rules for specifying a filename or directory as parameters*, page 274.

Description Use this option to list all possible diagnostic messages to 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.

Typically, this option cannot be given together with other options.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--discard_unused_publics

Syntax `--discard_unused_publics`

Description Use this option to discard unused public functions and variables when compiling with the `--mfc` compiler option.

Note: Do not use this option only on parts of the application, as necessary symbols might be removed from the generated output. Use the object attribute `__root` to keep symbols that are used from outside the compilation unit, for example, interrupt handlers. If the symbol does not have the `__root` attribute and is defined in the library, the library definition will be used instead.

See also *--mfc*, page 301 and *Multi-file compilation units*, page 247.

Project>Options>C/C++ Compiler>Discard unused publics

--dlib_config

Syntax `--dlib_config` *`filename`*`.h|`*`config`*

Parameters

`filename`	A DLIB configuration header file, see below the table.

config	The default configuration file for the specified configuration will be used. Choose between: `none`, no configuration will be used `normal`, the normal library configuration will be used (default) `full`, the full library configuration will be used.

See also *Rules for specifying a filename or directory as parameters*, page 274.

Description

Use this option to specify which library configuration to use, either by specifying an explicit file or by specifying a library configuration—in which case the default file for that library configuration will be used. Make sure that you specify a configuration that corresponds to the library you are using. If you do not specify this option, the default library configuration file will be used.

You can find the library object files in the directory `arm\lib` and the library configuration files in the directory `arm\inc\c`. For examples and information about prebuilt runtime libraries, see *Prebuilt runtime libraries*, page 141.

If you build your own customized runtime library, you can also create a corresponding customized library configuration file to specify to the compiler. For more information, see *Customizing and building your own runtime library*, page 137.

To set related options, choose:

Project>Options>General Options>Library Configuration

--do_explicit_zero_opt_in_named_sections

Syntax

`--do_explicit_zero_opt_in_named_sections`

Description

By default, the compiler treats static initialization of variables explicitly and implicitly initialized to zero the same, except for variables which are to be placed in user-named sections. For these variables, an explicit zero initialization is treated as a copy initialization, that is the same way as variables statically initialized to something other than zero.

Use this option to disable the exception for variables in user-named sections, and thus treat explicit initializations to zero as zero initializations, not copy initializations.

Example

```
int var1;                 // Implicit zero init -> zero inited
int var2 = 0;             // Explicit zero init -> zero inited
int var3 = 7;             // Not zero init      -> copy inited
int var4 @ "MYDATA";      // Implicit zero init -> zero inited
int var5 @ "MYDATA" = 0;  // Explicit zero init -> copy inited
                          // If option specified, then zero inited
int var6 @ "MYDATA" = 7;  // Not zero init      -> copy inited
```

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

-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` option cannot be used at the same time.

See also

Enabling language extensions, page 201.

Project>Options>C/C++ Compiler>Language 1>Standard with IAR extensions

Note: By default, this option is selected in the IDE.

--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 the compiler for a list of available parameters.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options.**

--enable_restrict

Syntax

`--enable_restrict`

Description

Enables the Standard C keyword `restrict` in C89 and C++. By default, `restrict` is recognized in Standard C and `__restrict` is always recognized.

This option can be useful for improving analysis precision during optimization.

To set this option, use **Project>Options>C/C++ Compiler>Extra options**

--endian

Syntax

`--endian={big|b|little|l}`

Parameters

`big, b`	Specifies big-endian as the default byte order. (This byte order cannot be used **in 64-bit mode**.)
`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. **In 64-bit mode**, this is the only byte order you can specify.

See also

Byte order (32-bit mode only), page 368.

Project>Options>General Options>32-bit>Byte order

--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. If you use this option when you compile a source file that uses a specific `enum` type, each source file that uses that `enum` type must be compiled using this option.

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 370.

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 See *Rules for specifying a filename or directory as parameters*, page 274.

Description 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.

If you use the compiler option `--dependencies`, extended command line files specified using `-f` will not generate a dependency, but those specified using `--f` will generate a dependency.

See also *--dependencies*, page 287 and *--f*, page 296.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--f

Syntax `--f` *`filename`*

Parameters See *Rules for specifying a filename or directory as parameters*, page 274.

Description 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.

If you use the compiler option `--dependencies`, extended command line files specified using `--f` will generate a dependency, but those specified using `-f` will not generate a dependency.

See also *--dependencies*, page 287 and *-f*, page 295.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--fpu

Syntax `--fpu={`*`name`*`|list|none}`

Parameters

`name`	The target FPU architecture.
`list`	Lists all supported values for the `--fpu` option.
`none` (default)	No FPU.

Description Use this option to generate code that performs floating-point operations using a Floating Point Unit (FPU). By selecting an FPU, you will override the use of the software floating-point library for all supported floating-point operations.

The name of a target FPU is constructed in one of these ways:

- `none`: No FPU (default)
- *`fp-architecture`*: Base variant of the specified architecture
- *`fp-architecture`*`-SP`: Single-precision variant

- *fp-architecture*_D16: Variant with 16 D registers
- *fp_architecture*_Fp16: Variant with half-precision extensions

The available combinations include:

- `{VFPv2|VFPv3|VFPv4|VFPv5}`
- `{VFPv3|FPv4|FPv5}_D16`
- `{FPv4|FPv5}-SP`
- `VFPv3_Fp16`
- `VFPv3_D16_Fp16`

Note: In 64-bit mode, this option has no effect. The AArch64 architecture always supports hardware floating-point operations.

See also

VFP and floating-point arithmetic, page 70.

Project>Options>General Options>32-bit>Floating-point settings

--guard_calls

Syntax

```
--guard_calls
```

Description

Use this option to enable guards for function static variable initialization. This option should be used in a threaded C++ environment.

See also

Managing a multithreaded environment, page 165.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--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 263.

Project>Options>C/C++ Compiler>Preprocessor>Additional include directories

-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.

See also *Rules for specifying a filename or directory as parameters*, page 274.

Description

Use this option to generate an assembler or C/C++ listing to a file.

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

To set related options, choose:

Project>Options>C/C++ Compiler>List

--lock_regs

Syntax

`--lock_regs=`*`register`*

Parameters

`registers`	A comma-separated list of register names and register intervals to be locked. **In 32-bit mode**: registers in the range `R4–R11`. **In 64-bit mode**: registers in the ranges `X9–X15`and `X18–X29`.

Description

Use this option to prevent the compiler from generating code that uses the specified registers.

Example

```
--lock_regs=R4
--lock_regs=R8–R11
--lock_regs=R4,R8–R11
```

To set this option, use **Project>Options>C/C++ Compiler>Extra Options.**

--macro_positions_in_diagnostics

Syntax

`--macro_positions_in_diagnostics`

Description

Use this option to obtain position references inside macros in diagnostic messages. This is useful for detecting incorrect source code constructs in macros.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--make_all_definitions_weak

Syntax

`--make_all_definitions_weak`

Description

Turns all variable and function definitions in the compilation unit into weak definitions.

Note: Normally, it is better to use extended keywords or pragma directives to turn individual variable and function definitions into weak definitions.

See also

`__`*weak*, page 399.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--max_cost_constexpr_call

Syntax

`--max_cost_constexpr_call=`*`limit`*

Parameters

`limit`	The number of calls and loop iterations. The default is 2000000.

Description

Use this option to specify an upper limit for the *cost* for folding a top-level `constexpr` call (function or constructor). The cost is a combination of the number of calls interpreted and the number of loop iterations preformed during the interpretation of a top-level call.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--max_depth_constexpr_call

Syntax

`--max_depth_constexpr_call=`*`limit`*

Parameters

`limit`	The depth of recursion. The default is 1000.

Description

Use this option to specify the maximum depth of recursion for folding a top-level `constexpr` call (function or constructor).

To set this option, use **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 enhances interprocedural optimizations.

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 291, *--output, -o*, page 313, and *Multi-file compilation units*, page 247.

Project>Options>C/C++ Compiler>Multi-file compilation

--no_alignment_reduction

Syntax

```
--no_alignment_reduction
```

Description

Some simple Thumb/Thumb2 functions can be 2-byte aligned. Use this option to keep those functions 4-byte aligned.

This option has no effect when compiling for Arm mode.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--no_bom

Syntax

```
--no_bom
```

Description

Use this option to omit the Byte Order Mark (BOM) when generating a UTF-8 output file.

See also

--text_out, page 321, and *Text encodings*, page 266.

Project>Options>C/C++ Compiler>Encodings>Text output file encoding

--no_call_frame_info

Syntax `--no_call_frame_info`

Description Normally, the compiler always generates call frame information in the output, to enable the debugger to display the call stack even in code from modules with no debug information. Use this option to disable the generation of call frame information.

See also *Call frame information*, page 195.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--no_clustering

Syntax `--no_clustering`

Description Use this option to disable static clustering optimizations.

Note: This option has no effect at optimization levels below Medium.

See also *Static clustering*, page 251.

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.

Note: This option has no effect at optimization levels below Medium.

See also *Code motion*, page 250.

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 367.

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.

See also

Common subexpression elimination, page 250.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Common subexpression elimination

--no_default_fp_contract

Syntax

```
--no_default_fp_contract
```

Description

The pragma directive `STDC FP_CONTRACT` specifies whether the compiler is allowed to contract floating-point expressions. The default for this pragma directive is `ON` (allowing contraction). Use this option to change the default to `OFF` (disallowing contraction).

See also

STDC FP_CONTRACT, page 423.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--no_exceptions

Syntax `--no_exceptions`

Description Use this option to disable exception support in the C++ language. Exception statements like `throw` and `try–catch`, and exception specifications on function definitions will generate an error message. Exception specifications on function declarations are ignored. The option is only valid when used together with the `--c++` compiler option.

If exceptions are not used in your application, it is recommended to disable support for them by using this option, because exceptions cause a rather large increase in code size.

See also *Exception handling*, page 209 and *__EXCEPTIONS*, page 483.

Project>Options>C/C++ Compiler>Language 1>C++

and deselect

Project>Options>C/C++ Compiler>Language 1>C++ options>Enable exceptions

--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 further minimize the size of the executable image. When you use this option, this information is not output in the object files.

See also *Keeping symbols and sections*, page 117.

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.

See also *Inlining functions*, page 90.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Function inlining

--no_literal_pool

Syntax

`--no_literal_pool`

Description

Use this option to generate code that should run from a memory region where it is not allowed to read data, only to execute code. **In 64-bit mode**, this option has no effect.

When this option is used, the compiler will construct addresses and large constants with the `MOV32` pseudo instruction instead of using a literal pool: switch statements are no longer translated using tables, and constant data is placed in the `.rodata` section.

This option also affects the automatic library selection performed by the linker. An IAR-specific ELF attribute is used for determining whether libraries compiled with this option should be used.

This option is only allowed for Armv6-M and Armv7 cores, and can be combined with the options `--ropi` or `--rwpi` only for Armv7 cores.

Note: For the M architecture profile (Cortex-M cores), this option is only available when you use the little-endian byte order.

See also

--*no_literal_pool*, page 354.

Project>Options>C/C++ Compiler>Code>No data reads in code memory

--no_loop_align

Syntax

`--no_loop_align`

Description

Use this option to disable the 4-byte alignment of labels in loops. This option is only useful in Thumb2 mode.

In Arm/Thumb1 mode, this option is enabled but does not perform anything.

See also

Alignment, page 367.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options.**

--no_mem_idioms

Syntax

```
--no_mem_idioms
```

Description

Use this option to make the compiler not optimize code sequences that clear, set, or copy a memory region. These memory access patterns (idioms) can otherwise be aggressively optimized, in some cases using calls to the runtime library. In principle, the transformation can involve more than a library call.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options.**

--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

Description of predefined preprocessor symbols, page 472.

This option is not available in the IDE.

--no_rtti

Syntax

```
--no_rtti
```

Description

Use this option to disable the runtime type information (RTTI) support in the C++ language. RTTI statements like `dynamic_cast` and `typeid` will generate an error message. This option is only valid when used together with the `--c++` compiler option.

See also

Using C++, page 207 and *__RTTI__*, page 485.

Project>Options>C/C++ Compiler>Language 1>C++

and deselect

Project>Options>C/C++ Compiler>Language 1>C++ options>Enable RTTI

--no_rw_dynamic_init

Syntax `--no_rw_dynamic_init`

Description Use this option to disable runtime initialization of static C variables.

C source code that is compiled with `--ropi` or `--rwpi` cannot have static pointer variables and constants initialized to addresses of objects that do not have a known address at link time. To solve this for writable static variables, the compiler generates code that performs the initialization at program startup (in the same way as dynamic initialization in C++).

See also --*ropi*, page 316 and --*rwpi*, page 317.

Project>Options>C/C++ Compiler>Code>No dynamic read/write/initialization

--no_scheduling

Syntax `--no_scheduling`

Description Use this option to disable the instruction scheduler.

Note: This option has no effect at optimization levels below High.

See also *Instruction scheduling*, page 252.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Instruction scheduling

--no_size_constraints

Syntax `--no_size_constraints`

Description Use this option to relax the normal restrictions for code size expansion when optimizing for high speed.

Note: This option has no effect unless used with `-Ohs`.

See also *Speed versus size*, page 249.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>No size constraints

--no_static_destruction

Syntax `--no_static_destruction`

Description Normally, the compiler emits code to destroy C++ static variables that require destruction at program exit. Sometimes, such destruction is not needed.

Use this option to suppress the emission of such code.

See also *Setting up the atexit limit*, page 118.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--no_system_include

Syntax `--no_system_include`

Description By default, the compiler automatically locates the system include files. Use this option to disable the automatic search for system include files. In this case, you might need to set up the search path by using the `-I` compiler option.

See also --*dlib_config*, page 291, and --*system_include_dir*, page 321.

Project>Options>C/C++ Compiler>Preprocessor>Ignore standard include directories

--no_tbaa

Syntax `--no_tbaa`

Description Use this option to disable type-based alias analysis.

Note: This option has no effect at optimization levels below High.

See also *Type-based alias analysis*, page 251.

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 = "My text string";
```

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, might 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.

For the architectures Armv7-M, Armv8-A, and Armv8-M.mainline, the hardware support for unaligned access can be controlled by software. There are variants of library routines for these architectures that are faster when unaligned access is supported in hardware (symbols with the prefix `__iar_unaligned_`). The IAR linker will not use these variants if any of the input modules does not allow unaligned access.

See also

Alignment, page 367.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options.**

--no_uniform_attribute_syntax

Syntax `--no_uniform_attribute_syntax`

Description Use this option to apply the default syntax rules to IAR type attributes specified before a type specifier.

See also *--uniform_attribute_syntax*, page 322 and *Syntax for type attributes used on data objects*, page 384.

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.

Note: This option has no effect at optimization levels below High.

See also *Loop unrolling*, page 250.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Loop unrolling

--no_var_align

Syntax `--no_var_align`

Description By default, the compiler uses alignment 4 for variable objects with a size of 4 bytes or more. Use this option to make the compiler align variable objects based on the alignment of their type.

For example, a `char` array will get alignment 1, because its elements of the type `char` have alignment 1. Using this option might save RAM space, possibly at the expense of processing speed.

See also *Alignment*, page 367 and *--no_const_align*, page 303.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options.**

--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.

--nonportable_path_warnings

Syntax

```
--nonportable_path_warnings
```

Description

Use this option to make the compiler generate a warning when characters in the path used for opening a source file or header file are lower case instead of upper case, or vice versa, compared with the path in the file system.

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

*All optimizations performed at level Low will be performed also at None. The only difference is that at level 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.

At high optimization levels, when favoring speed or size (`-Ohs` or `-Ohz`), the compiler will emit AEABI attributes indicating the requested optimization goal. This information can be used by the linker to select smaller or faster variants of DLIB library functions.

- If a module referencing a function is compiled with `-Ohs`, and the DLIB library contains a fast variant, that variant is used.
- If all modules referencing a function are compiled with `-Ohz`, and the DLIB library contains a small variant, that variant is used.

For example, using `-Ohz` for Cortex-M0 will result in the use of a smaller AEABI library routine for integer division.

See also

Controlling compiler optimizations, page 246.

Project>Options>C/C++ Compiler>Optimizations

--only_stdout

Syntax

`--only_stdout`

Description

Use this option to make the compiler use the standard output stream (`stdout`), and 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

See *Rules for specifying a filename or directory as parameters*, page 274.

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.

--pending_instantiations

Syntax

```
--pending_instantiations number
```

Parameters

number	An integer that specifies the limit, where `64` is default. If `0` is used, there is no limit.

Description

Use this option to specify the maximum number of instantiations of a given C++ template that is allowed to be in process of being instantiated at a given time. This is used for detecting recursive instantiations.

Project>Options>C/C++ Compiler>Extra Options

--predef_macros

Syntax

```
--predef_macros {filename|directory}
```

Parameters

See *Rules for specifying a filename or directory as parameters*, page 274.

Description

Use this option to list all symbols defined by the compiler or on the command line. (Symbols defined in the source code are not listed.) 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: 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

See *Rules for specifying a filename or directory as parameters*, page 274.

Description

Use this option to make the compiler read 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][s]] {filename|directory}`

Parameters

`c`	Include comments
`n`	Preprocess only
`s`	Suppress `#line` directives

See also *Rules for specifying a filename or directory as parameters*, page 274.

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.

--relaxed_fp

Syntax

```
--relaxed_fp
```

Description

Use this option to allow the compiler to relax the language rules and perform more aggressive optimization of floating-point expressions. This option improves performance for floating-point expressions that fulfill these conditions:

- The expression consists of both single and double-precision values
- The double-precision values can be converted to single precision without loss of accuracy
- The result of the expression is converted to single precision.

Note: Performing the calculation in single precision instead of double precision might cause a loss of accuracy.

Example

```
float F(float a, float b)
{
  return a + b * 3.0;
}
```

The C standard states that `3.0` in this example has the type `double` and therefore the whole expression should be evaluated in `double` precision. However, when the `--relaxed_fp` option is used, 3.0 will be converted to `float` and the whole expression can be evaluated in `float` precision.

To set related options, choose:

Project>Options>C/C++ Compiler>Language 2>Floating-point semantics

--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 269.

Project>Options>C/C++ Compiler>Diagnostics>Enable remarks

--require_prototypes

Syntax

```
--require_prototypes
```

Description

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

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

Project>Options>C/C++ Compiler>Language 1>Require prototypes

--ropi

Syntax

```
--ropi
```

Description

Use this option to make the compiler generate code that uses `PC`-relative references to address code and read-only data. **In 64-bit mode**, this option has no effect.

When this option is used, these limitations apply:

- C++ constructions cannot be used
- The object attribute `__ramfunc` cannot be used

- Pointer constants cannot be initialized with the address of another constant, a string literal, or a function. However, writable variables can be initialized to constant addresses at runtime.

Consider using `movable` blocks in the linker configuration file. See *define block directive*, page 511.

See also

--*no_rw_dynamic_init*, page 307, and *Description of predefined preprocessor symbols*, page 472.

Project>Options>C/C++ Compiler>Code>Code and read-only data (ropi)

--ropi_cb

Syntax

`--ropi_cb`

Description

Use this option to make all accesses to constant data, base-addressed relative to the register `R8`. **In 64-bit mode**, this option has no effect.

Use `--ropi_cb` together with `--ropi` to activate a variant of ROPI that uses the Arm core register `R8` as the base address for read-only data, instead of using the `PC`. This is useful, for example, when using ROPI in code that runs from execute-only memory, which is enabled if you compile and link with `--no_literal_pool`.

Note:

- The use of `--ropi_cb` is not AEABI-compliant.
- There is no provided setup of the register `R8`. This must be handled by your application.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--rwpi

Syntax

`--rwpi`

Description

Use this option to make the compiler generate code that uses the offset from the static base register (`R9`) to address-writable data. **In 64-bit mode**, this option has no effect.

When this option is used, these limitations apply:

- The object attribute `__ramfunc` cannot be used

- Pointer constants cannot be initialized with the address of a writable variable. However, static writable variables can be initialized to writable addresses at runtime.

Consider using `movable` blocks in the linker configuration file. See *define block directive*, page 511.

See also

--*no_rw_dynamic_init*, page 307, and *Description of predefined preprocessor symbols*, page 472.

Project>Options>C/C++ Compiler>Code>Read/write data (rwpi)

--rwpi_near

Syntax

`--rwpi_near`

Description

Use this option to make the compiler generate code that uses the offset from the static base register (`R9`) to address-writable data. **In 64-bit mode**, this option has no effect.

When this option is used, these limitations apply

- The object attribute `__ramfunc` cannot be used.
- Pointer constants cannot be initialized with the address of a writable variable. However, static writable variables can be initialized to writable addresses at runtime.
- A maximum of 64 Kbytes of read/write memory can be addressed.

See also

--*no_rw_dynamic_init*, page 307 and *Description of predefined preprocessor symbols*, page 472.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--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: 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

Controlling data and function placement in memory, page 242.

Project>Options>C/C++ Compiler>Output>Code section name

--section_prefix

Syntax

`--section_prefix=`*`prefix`*

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 sections that are not explicitly named using the `@` notation or the `#pragma location` directive.

This option creates section names by putting a prefix before the default name for the section type. This makes it possible to use different section-selectors for different purposes. You can use `tcm.*` in the example below to match the prefix or, for example, `*.bss` to match sections with zero-initialized data.

Note: Any changes to the section names require corresponding modifications in the linker configuration file.

Example

Specifying `--section_prefix=tcm` places:

- code in `tcm.text` instead of `.text`
- read-only data in `tcm.rodata` instead of `.rodata`
- zero-initialized data in `tcm.bss` instead of `.bss`
- other initialized data in `tcm.data` instead of `.data`

See also

Controlling data and function placement in memory, page 242.

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.

--source_encoding

Syntax `--source_encoding {locale|utf8}`

Parameters

`locale`	The default source encoding is the system locale encoding.
`utf8`	The default source encoding is the UTF-8 encoding.

Description When reading a source file with no Byte Order Mark (BOM), use this option to specify the encoding. If this option is not specified and the source file does not have a BOM, the Raw encoding will be used.

See also *Text encodings*, page 266.

Project>Options>C/C++ Compiler>Encodings>Default source file encoding

--stack_protection

Syntax `--stack_protection`

Description Use this option to enable stack protection for the functions that are considered to need it.

See also *Stack protection*, page 92.

Project>Options>C/C++ Compiler>Code>Stack protection

--strict

Syntax `--strict`

Description By default, the compiler accepts a relaxed superset of Standard C and C++. Use this option to ensure that the source code of your application instead conforms to strict Standard C and C++.

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

See also *Enabling language extensions*, page 201.

Project>Options>C/C++ Compiler>Language 1>Language conformance>Strict

--system_include_dir

Syntax `--system_include_dir path`

Parameters To specify the path to the system include files, see *Rules for specifying a filename or directory as parameters*, page 274.

Description By default, the compiler automatically locates the system include files. Use this option to explicitly specify a different path to the system include files. This might be useful if you have not installed IAR Embedded Workbench in the default location.

See also *--dlib_config*, page 291, and *--no_system_include*, page 308.

This option is not available in the IDE.

--text_out

Syntax `--text_out {utf8|utf16le|utf16be|locale}`

Parameters

`utf8`	Uses the UTF-8 encoding
`utf16le`	Uses the UTF-16 little-endian encoding
`utf16be`	Uses the UTF-16 big-endian encoding
`locale`	Uses the system locale encoding

Description

Use this option to specify the encoding to be used when generating a text output file.

The default for the compiler list files is to use the same encoding as the main source file. The default for all other text files is UTF-8 with a Byte Order Mark (BOM).

If you want text output in UTF-8 encoding without a BOM, use the option `--no_bom`.

See also

--*no_bom*, page 301 and *Text encodings*, page 266.

Project>Options>C/C++ Compiler>Encodings>Text output file encoding

--thumb

Syntax

```
--thumb
```

Description

Use this option to set default function mode to Thumb (T32 or T in 32-bit mode).

Note: This option has the same effect as the `--cpu_mode=thumb` option.

Project>Options>C/C++ Compiler>Code>Processor mode>Thumb

--uniform_attribute_syntax

Syntax

```
--uniform_attribute_syntax
```

Description

By default, an IAR type attribute specified before the type specifier applies to the object or typedef itself, and not to the type specifier, as `const` and `volatile` do. If you specify this option, IAR type attributes obey the same syntax rules as `const` and `volatile`.

The default for IAR type attributes is to *not* use uniform attribute syntax.

See also

--*no_uniform_attribute_syntax*, page 310 and *Syntax for type attributes used on data objects*, page 384.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--use_c++_inline

Syntax `--use_c++_inline`

Description Standard C uses slightly different semantics for the `inline` keyword than C++ does. Use this option if you want C++ semantics when you are using C.

See also *Inlining functions*, page 90.

Project>Options>C/C++ Compiler>Language 1>C dialect>C++ inline semantics

--use_paths_as_written

Syntax `--use_paths_as_written`

Description By default, the compiler ensures that all paths in the debug information are absolute, even if not originally specified that way.

If you use this option, paths that were originally specified as relative will be relative in the debug information.

The paths affected by this option are:

- the paths to source files
- the paths to header files that are found using an include path that was specified as relative

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--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 file 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**.

--utf8_text_in

Syntax

`--utf8_text_in`

Description

Use this option to specify that the compiler shall use UTF-8 encoding when reading a text input file with no Byte Order Mark (BOM).

Note: This option does not apply to source files.

See also

Text encodings, page 266.

Project>Options>C/C++ Compiler>Encodings>Default input file encoding

--vectorize

Syntax

`--vectorize`

Description

Use this option to enable generation of NEON vector instructions **in 32-bit mode**.

Loops will only be vectorized if the target processor has NEON capability and the optimization level is `-Ohs`. Auto-vectorization is not supported **in 64-bit mode**.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Vectorize

--version

Syntax

`--version`

Description

Use this option to make the compiler send version information to the console and then exit.

This option is not available in the IDE.

--vla

Syntax

`--vla`

Description

Use this option to enable support for variable length arrays in C code. Such arrays are located on the heap. This option requires Standard C and cannot be used together with the `--c89` compiler option.

Note: `--vla` should not be used together with the `longjmp` library function, as that can lead to memory leakages.

See also

C language overview, page 199.

Project>Options>C/C++ Compiler>Language 1>C dialect>Allow VLA

--warn_about_c_style_casts

Syntax

`--warn_about_c_style_casts`

Description

Use this option to make the compiler warn when C-style casts are used in C++ source code.

This option is not available in the IDE.

--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 290.

Project>Options>C/C++ Compiler>Diagnostics>Treat all warnings as errors

Linker options

- Summary of linker options
- Descriptions of linker options

For general syntax rules, see *Options syntax*, page 273.

Summary of linker options

This table summarizes the linker options:

Command line option	Description
`--abi`	Specifies which data model to link for
`--advanced_heap`	Uses an advanced heap
`--basic_heap`	Uses a basic heap
`--BE8`	Uses the big-endian format BE8
`--BE32`	Uses the big-endian format BE32
`--bounds_table_size`	Specifies the size of the global bounds table. See the C-RUN documentation in the *C-SPY® Debugging Guide for Arm*.
`--call_graph`	Produces a call graph file in XML format
`--config`	Specifies the linker configuration file to be used by the linker
`--config_def`	Defines symbols for the configuration file
`--config_search`	Specifies more directories to search for linker configuration files
`--cpp_init_routine`	Specifies a user-defined C++ dynamic initialization routine
`--cpu`	Specifies a processor variant
`--debug_heap`	Uses the checked heap. See the C-RUN documentation in the *C-SPY® Debugging Guide for Arm*.
`--default_to_complex_ranges`	Makes `complex ranges` the default decompressor in `initialize` directives

Table 31: Linker options summary

Command line option	Description
--define_symbol	Defines symbols that can be used by the application
--dependencies	Lists file dependencies
--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
--do_segment_pad	Pads each ELF segment to n-byte alignment
--enable_hardware_workaround	Enables specified hardware workaround
--enable_stack_usage	Enables stack usage analysis
--entry	Treats the symbol as a root symbol and as the start of the application
--entry_list_in_address_order	Generates an additional entry list in the map file sorted in address order
--error_limit	Specifies the allowed number of errors before linking stops
--exception_tables	Generates exception tables for C code
--export_builtin_config	Produces an `icf` file for the default configuration
--extra_init	Specifies an extra initialization routine that will be called if it is defined.
-f	Extends the command line
--f	Extends the command line, optionally with a dependency.
--force_exceptions	Always includes exception runtime code
--force_output	Produces an output file even if errors occurred
--fpu	Selects the FPU to link your application for
--ignore_uninstrumented_pointers	Disables checking of accessing via pointers in memory for which no bounds have been set. See the C-RUN documentation in the *C-SPY® Debugging Guide for Arm*.
--image_input	Puts an image file in a section
--import_cmse_lib_in	Reads previous version of import library for building a non-secure image

Table 31: Linker options summary (Continued)

Command line option	Description
--import_cmse_lib_out	Produces an import library, for building a non-secure image
--inline	Inlines small routines
--keep	Forces a symbol to be included in the application
-L	Specifies more directories to search for object and library files. Alias for --search.
--log	Enables log output for selected topics
--log_file	Directs the log to a file
--mangled_names_in_messages	Adds mangled names in messages
--manual_dynamic_initialization	Suppresses automatic initialization during system startup
--map	Produces a map file
--merge_duplicate_sections	Merges equivalent read-only sections
--no_bom	Omits the Byte Order Mark from UTF-8 output files
--no_dynamic_rtti_elimination	Includes dynamic runtime type information even when it is not needed.
--no_entry	Sets the entry point to zero
--no_exceptions	Generates an error if exceptions are used
--no_fragments	Disables section fragment handling
--no_free_heap	Uses the smallest possible heap implementation
--no_inline	Excludes functions from small function inlining
--no_library_search	Disables automatic runtime library search
--no_literal_pool	Generates code that should run from a memory region where it is not allowed to read data, only to execute code
--no_locals	Removes local symbols from the ELF executable image.
--no_range_reservations	Disables range reservations for absolute symbols
--no_remove	Disables removal of unused sections
--no_vfe	Disables Virtual Function Elimination
--no_warnings	Disables generation of warnings
--no_wrap_diagnostics	Does not wrap long lines in diagnostic messages
-o	Sets the object filename. Alias for --output.

Table 31: Linker options summary (Continued)

Command line option	Description
--only_stdout	Uses standard output only
--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.
--preconfig	Reads the specified file before reading the linker configuration file
--printf_multibytes	Makes the printf formatter support multibytes
--redirect	Redirects a reference to a symbol to another symbol
--remarks	Enables remarks
--scanf_multibytes	Makes the scanf formatter support multibytes
--search	Specifies more directories to search for object and library files
--semihosting	Links with debug interface
--silent	Sets silent operation
--stack_usage_control	Specifies a stack usage control file
--strip	Removes debug information from the executable image
--text_out	Specifies the encoding for text output files
--threaded_lib	Configures the runtime library for use with threads
--timezone_lib	Enables the time zone and daylight savings time functionality in the library
--treat_rvct_modules_as_softfp	Treats all modules generated by RVCT as using the standard (non-VFP) calling convention
--use_full_std_template_names	Enables full names for standard C++ templates
--use_optimized_variants	Controls the use of optimized variants of DLIB library functions
--utf8_text_in	Uses the UTF-8 encoding for text input files
--version	Sends version information to the console and then exits
--vfe	Controls Virtual Function Elimination

Table 31: Linker options summary (Continued)

Command line option	Description
`--warnings_affect_exit_code`	Warnings affects exit code
`--warnings_are_errors`	Warnings are treated as errors
`--whole_archive`	Treats every object file in the archive as if it was specified on the command line.

Table 31: Linker options summary (Continued)

Descriptions of linker options

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

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.

--abi

Syntax

`--abi {ilp32|lp64}`

Parameters

`ilp32`	Links A64 code for the ILP32 data model
`lp64`	Links A64 code for the LP64 data model

Description

When linking for the A64 instruction set, the linker detects the data model from attributes in the object files. If you use this option to specify the intended data model, the linker issues an error if an object file does not have the expected attribute.

See also

--*aarch64*, page 281 and --*cpu_mode*, page 286.

To set related options, choose:

Project>Options>General Options>Target>Execution mode

and

Project>Options>General Options>64-bit>Data model

--advanced_heap

Syntax

`--advanced_heap`

Description

Use this option to use an advanced heap.

See also *Heap memory handlers*, page 219.

Project>Options>General Options>Library options 2>Heap selection

--basic_heap

Syntax `--basic_heap`

Description Use this option to use the basic heap handler.

See also *Heap memory handlers*, page 219.

Project>Options>General Options>Library options 2>Heap selection

--BE8

Syntax `--BE8`

Description Use this option to specify the Byte Invariant Addressing mode. **In 64-bit mode**, this option has no effect.

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 70, *Byte order (32-bit mode only)*, page 368, *--BE32*, page 332, and *--endian*, page 294.

Project>Options>General Options>32-bit>Byte order

--BE32

Syntax `--BE32`

Description Use this option to specify the legacy big-endian mode. **In 64-bit mode**, this option has no effect.

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 70, *Byte order (32-bit mode only)*, page 368, *--BE8*, page 332, and *--endian*, page 294.

Project>Options>General Options>32-bit>Byte order

--call_graph

Syntax

`--call_graph {filename|directory}`

Parameters

See *Rules for specifying a filename or directory as parameters*, page 274.

Description

Use this option to produce a call graph file. If no filename extension is specified, the extension `cgx` is used. This option can only be used once on the command line.

Using this option enables stack usage analysis in the linker.

See also

Stack usage analysis, page 105

Project>Options>Linker>Advanced>Enable stack usage analysis>Call graph output (XML)

--config

Syntax

`--config filename`

Parameters

See *Rules for specifying a filename or directory as parameters*, page 274.

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.
`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 than once on the command line.

See also --*define_symbol*, page 336 and *Interaction between ILINK and the application*, page 122.

Project>Options>Linker>Config>Defined symbols for configuration file

--config_search

Syntax `--config_search` *`path`*

Parameters

`path`	A path to a directory where the linker should search for linker configuration include files.

Description Use this option to specify more directories to search for files when processing an `include` directive in a linker configuration file.

By default, the linker searches for configuration include files only in the system configuration directory. To specify more than one search directory, use this option for each path.

See also *include directive*, page 535.

To set this option, use **Project>Options>Linker>Extra Options**.

--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 require another routine to handle the initialization, for instance if you are not using the standard library.

To set this option, use **Project>Options>Linker>Extra Options**.

--cpu

Syntax `--cpu=`*`core`*`|list`

Parameters

`core`	Specifies a specific processor variant
`list`	Lists all supported values for the option `--cpu`

Description Use this option to select the processor variant to link your application for. The default is to use a processor or architecture compatible with the object file attributes.

See also --*cpu*, page 284

Project>Options>General Options>Target>Processor variant

--default_to_complex_ranges

Syntax `--default_to_complex_ranges`

Description Normally, if `initialize` directives in a linker configuration file do not specify `simple ranges` or `complex ranges`, the linker uses `simple ranges` if the associated `section placement` directives use single range regions.

Use this option to make the linker always use `complex ranges` by default. This was the behavior of the linker before the introduction of `simple ranges` and `complex ranges`.

See also *initialize directive*, page 517.

To set this option, use **Project>Options>Linker>Extra Options**

--define_symbol

Syntax `--define_symbol symbol=constant_value`

Parameters

symbol	The name of the constant symbol that can be used by the application.
constant_value	The constant value of the symbol.

Description Use this option to define a constant symbol, that is a label, that can be used by your application. This option can be used more than once on the command line.

Note: This option is different from the `define symbol` directive.

See also *--config_def*, page 334 and *Interaction between ILINK and the application*, page 122.

Project>Options>Linker>#define>Defined symbols

--dependencies

Syntax `--dependencies[=[i|m]] {filename|directory}`

Parameters

`i` (default)	Lists only the names of files

`m`	Lists in makefile style

See also *Rules for specifying a filename or directory as parameters*, page 274.

Description

Use this option to make the linker list the names of the linker configuration, object, and library files opened for input into a file with the default filename extension `i`.

Example

If `--dependencies` or `--dependencies=i` is used, the name of each opened input file, including the full path, if available, is output on a separate line. For example:

```
c:\myproject\foo.o
d:\myproject\bar.o
```

If `--dependencies=m` is used, the output is in makefile style. For each input file, one line containing a makefile dependency rule is produced. Each line consists of the name of the output file, a colon, a space, and the name of an input file. For example:

```
a.out: c:\myproject\foo.o
a.out: d:\myproject\bar.o
```

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 problem 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 `Go109`

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.

Note: Not all diagnostic messages can be reclassified. 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 `Pa180`

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.

Note: Not all diagnostic messages can be reclassified.

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 `Li004`

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.

Note: Not all diagnostic messages can be reclassified.

Project>Options>Linker>Diagnostics>Treat these as warnings

--diagnostics_tables

Syntax

`--diagnostics_tables {filename|directory}`

Parameters

See *Rules for specifying a filename or directory as parameters*, page 274.

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.

--do_segment_pad

Syntax

`--do_segment_pad`

Description

Use this option to extend each ELF segment in the executable file with content, to make it an even multiple of 4 bytes long (if possible). Some runtime library routines might access memory in units of 4 bytes, and might, if the right data object is placed last in an ELF segment, access memory outside the strict bounds of the segment. If you are executing in an environment where this is a problem, you can use this option to extend the ELF segments appropriately so that this is not a problem.

This option is not available in the IDE.

--enable_hardware_workaround

Syntax

```
--enable_hardware_workaround=waid[waid[...]]
```

Parameters

waid	The ID number of the workaround that you want to enable. For a list of available workarounds, see the release notes available in the Information Center.

Description

Use this option to make the linker generate a workaround for a specific hardware problem.

See also

The release notes for the linker for a list of available parameters.

To set this option, use **Project>Options>Linker>Extra Options.**

--enable_stack_usage

Syntax

```
--enable_stack_usage
```

Description

Use this option to enable stack usage analysis. If a linker map file is produced, a stack usage chapter is included in the map file.

Note: If you use at least one of the `--stack_usage_control` or `--call_graph` options, stack usage analysis is automatically enabled.

See also

Stack usage analysis, page 105.

Project>Options>Linker>Advanced>Enable stack usage analysis

--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 but a module part of a library is only included if needed.

Note: The label referred to must be available in your application. You must also make sure that the reset vector refers to the new start label, for example `--redirect __iar_program_start=_myStartLabel`.

See also --*no_entry*, page 352.

Project>Options>Linker>Library>Override default program entry

--entry_list_in_address_order

Syntax `--entry_list_in_address_order`

Description Use this option to generate an additional entry list in the map file. This entry list will be sorted in address order.

To set this option use **Project>Options>Linker>Extra Options**.

--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.

--exception_tables

Syntax

`--exception_tables={nocreate|unwind|cantunwind}`

Parameters

`nocreate` (default)	Does not generate entries. Uses the least amount of memory, but the result is undefined if an exception is propagated through a function without exception information.
`unwind`	Generates unwind entries that enable an exception to be correctly propagated through functions without exception information.
`cantunwind`	Generates no-unwind entries so that any attempt to propagate an exception through the function will result in a call to `terminate`.

Description

Use this option to determine what the linker should do with functions that do not have exception information but which do have correct call frame information.

The compiler ensures that C functions get correct call frame information. For functions written in assembler language you need to use assembler directives to generate call frame information.

See also

Using C++, page 207.

To set this option, use **Project>Options>Linker>Extra Options**.

--export_builtin_config

Syntax

`--export_builtin_config filename`

Parameters

See *Rules for specifying a filename or directory as parameters*, page 274.

Description

Exports the configuration used by default to a file.

This option is not available in the IDE.

--extra_init

Syntax `--extra_init routine`

Parameters

`routine`	A user-defined initialization routine.

Description

Use this option to make the linker add an entry for the specified routine at the end of the initialization table. The routine will be called during system startup, after other initialization routines have been called and before `main` is called. No entry is added if the routine is not defined.

Note: The routine must preserve the value passed to it in register `R0`. For this reason, it is safest to write it in assembler language.

To set this option, use **Project>Options>Linker>Extra Options**.

-f

Syntax `-f filename`

Parameters

See *Rules for specifying a filename or directory as parameters*, page 274.

Description

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.

See also

--f, page 344.

To set this option, use **Project>Options>Linker>Extra Options**.

--f

Syntax `--f filename`

Parameters See *Rules for specifying a filename or directory as parameters*, page 274.

Description 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.

If you use the linker option `--dependencies`, extended command line files specified using `--f` will generate a dependency, but those specified using `-f` will not generate a dependency.

See also *--dependencies*, page 287 and *-f*, page 295.

To set this option, use **Project>Options>Linker>Extra Options**.

--force_exceptions

Syntax `--force_exceptions`

Description Use this option to make the linker include exception tables and exception code even when the linker heuristics indicate that exceptions are not used.

The linker considers exceptions to be used if there is a `throw` expression that is not a `rethrow` in the included code. If there is no such `throw` expression in the rest of the code, the linker arranges for `operator new`, `dynamic_cast`, and `typeid` to call `abort` instead of throwing an exception on failure. If you need to catch exceptions from these constructs but your code contains no other throws, you might need to use this option.

See also *Using C++*, page 207.

Project>Options>Linker>Advanced>Allow C++ exceptions>Always include

--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**

--fpu

Syntax `--fpu={`*`name`*`|none}`

Parameters

`name`	The target FPU architecture.
`none`	No FPU.

Description By default, the linker links your application for the FPU compatible with the object file attribute. Use this option to explicitly specify an FPU to link your application for.

Note: In 64-bit mode, this option has no effect.

See also *--fpu*, page 296

Project>Options>General Options>32-bit>FPU

--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.

Note: Just as for sections from object files, sections created by using the `--image_input` option are not included unless actually needed. You can either specify a symbol in the option and reference this symbol in your application (or use a `--keep` option), or you can specify a section name and use the `keep` directive in a linker configuration file to ensure that the section is included.

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 347.

Project>Options>Linker>Input>Raw binary image

--import_cmse_lib_in

Syntax

```
--import_cmse_lib_in filename
```

Parameters

See *Rules for specifying a filename or directory as parameters*, page 274.

Description

Reads a previous version of the import library and creates gateway veneers with the same address as given in the import library. Use this option to create a secure image where each entry function that exists in the provided import library is placed at the same address in the output import library.

Note: In 64-bit mode, this option has no effect.

See also

--*cmse*, page 284 and --*import_cmse_lib_out*, page 347

To set this option, use **Project>Options>Linker>Extra Options**.

--import_cmse_lib_out

Syntax

`--import_cmse_lib_out filename|directory`

Parameters

See *Rules for specifying a filename or directory as parameters*, page 274.

Description

Use this option when building a secure image to automatically create an import library for use in a corresponding non-secure image. The import library consists of a relocatable ELF object module that contains only a symbol table. Each symbol specifies an absolute address of a secure gateway for an entry in the section `Veneer$$CMSE`.

Note: In 64-bit mode, this option has no effect.

See also

--cmse, page 284 and *--import_cmse_lib_in*, page 346

To set this option, use **Project>Options>Linker>Extra Options**.

--inline

Syntax

`--inline`

Description

Some routines are so small that they can fit in the space of the instruction that calls the routine. Use this option to make the linker replace the call of a routine with the body of the routine, where applicable.

See also

Small function inlining, page 127.

Project>Options>Linker>Optimizations>Inline small routines

--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 *topic* can be one of:

`call_graph`	Lists the call graph as seen by stack usage analysis.
`crt_routine_selection`	Lists details of the selection process for runtime routines—what definitions were available, what the requirements were, and which decision the process resulted in.
`demangle`	Uses demangled names instead of mangled names for C/C++ symbols in the log output, for example, `void h(int, char)` instead of `_Z1hic`.
`fragment_info`	Lists all fragments by number. The information contains the section they correspond to (name, section number and file) and the fragment size.
`initialization`	Lists copy batches and the compression selected for each batch.
`inlining`	Lists the functions that were inlined, and which sections (name, section number and file) they were inlined in. Note that inlining in the linker must be enabled by the `--inline` linker option. See *--inline*, page 347.
`libraries`	Lists all decisions made by the automatic library selector. This might include extra symbols needed (`--keep`), redirections (`--redirect`), as well as which runtime libraries that were selected.
`merging`	Lists the sections (name, section number and file) that were merged and which symbol redirections this resulted in. Note that section merging must be enabled by the `--merge_duplicate_sections` linker option. See *--merge_duplicate_sections*, page 351.
`modules`	Lists each module that is selected for inclusion in the application, and which symbol that caused it to be included.
`redirects`	Lists redirected symbols.
`sections`	Lists each symbol and section fragment that is selected for inclusion in the application, and the dependence that caused it to be included.

`veneers`	Lists some veneer creation and usage statistics.
`unused_fragments`	Lists those section fragments that were not included in the application.

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 349.

Project>Options>Linker>List>Generate log

--log_file

Syntax

`--log_file` *`filename`*

Parameters

See *Rules for specifying a filename or directory as parameters*, page 274.

Description

Use this option to direct the log output to the specified file.

See also

--log, page 348.

Project>Options>Linker>List>Generate log

--mangled_names_in_messages

Syntax

`--mangled_names_in_messages`

Description

Use this option to produce both mangled and demangled 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.

--manual_dynamic_initialization

Syntax

`--manual_dynamic_initialization`

Description

Normally, dynamic initialization (typically initialization of C++ objects with static storage duration) is performed automatically during application startup. If you use `--manual_dynamic_initialization`, you must call `__iar_dynamic_initialization` at some later point for this initialization to be done.

The function `__iar_dynamic_initialization` is declared in the header file `iar_dynamic_init.h`.

To set this option use **Project>Options>Linker>Extra Options**.

--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 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 also sometimes 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

--merge_duplicate_sections

Syntax `--merge_duplicate_sections`

Description Use this option to keep only one copy of equivalent read-only sections.

Note: This can cause different functions or constants to have the same address, so an application that depends on the addresses being different will not work correctly with this option enabled.

See also *Duplicate section merging*, page 127.

Project>Options>Linker>Optimizations>Merge duplicate sections

--no_bom

Syntax `--no_bom`

Description Use this option to omit the Byte Order Mark (BOM) when generating a UTF-8 output file.

See also *--text_out*, page 361 and *Text encodings*, page 266.

Project>Options>Linker>Encodings>Text output file encoding

--no_dynamic_rtti_elimination

Syntax `--no_dynamic_rtti_elimination`

Description Use this option to make the linker include dynamic (polymorphic) runtime type information (RTTI) data in the application image even when the linker heuristics indicate that it is not needed.

The linker considers dynamic runtime type information to be needed if there is a `typeid` or `dynamic_cast` expression for a polymorphic type in the included code. By default, if the linker detects no such expression, RTTI data will not be included just to make dynamic RTTI requests work.

Note: A `typeid` expression for a *non*-polymorphic type results in a direct reference to a particular RTTI object and will not cause the linker to include any potentially unneeded objects.

See also

Using C++, page 207.

To set this option, use **Project>Options>Linker>Extra Options**.

--no_entry

Syntax

`--no_entry`

Description

Use this option to set the entry point field to zero for produced ELF files.

See also

--*entry*, page 341.

Project>Options>Linker>Library>Override default program entry

--no_exceptions

Syntax

`--no_exceptions`

Description

Use this option to make the linker generate an error if there is a throw in the included code. This option is useful for making sure that your application does not use exceptions.

See also

Using C++, page 207.

To set related options, choose:

Project>Options>Linker>Advanced>Allow C++ exceptions

--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 further minimize the size of the executable image. Use this option to disable the removal of fragments of sections, instead including or not including each section in its entirety, usually resulting in a larger application.

See also

Keeping symbols and sections, page 117.

To set this option, use **Project>Options>Linker>Extra Options**

--no_free_heap

Syntax

```
--no_free_heap
```

Description

Use this option to use the smallest possible heap implementation. Because this heap does not support `free` or `realloc`, it is only suitable for applications that in the startup phase allocate heap memory for various buffers, etc, and for applications that never deallocate memory.

See also

Heap memory handlers, page 219

Project>Options>General Options>Library Options 2>Heap selection

--no_inline

Syntax

`--no_inline` *`func`*`[,`*`func`*`...]`

Parameters

`func`	The name of a function symbol

Description

Use this option to exclude some functions from small function inlining.

See also

--*inline*, page 347.

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.

Note: The option disables all steps of the automatic library selection, some of which might need to be reproduced if you are using the standard libraries. Use the `--log libraries` linker option together with automatic library selection enabled to determine which the steps are.

Project>Options>Linker>Library>Automatic runtime library selection

--no_literal_pool

Syntax
`--no_literal_pool`

Description
Use this option for code that should run from a memory region where it is not allowed to read data, only to execute code.

When this option is used, the linker will use the `MOV32` pseudo instruction in a mode-changing veneer, to avoid using the data bus to load the destination address. The option also means that libraries compiled with this option will be used.

The option `--no_literal_pool` is only allowed for Armv6-M and Armv7-M cores.

See also
--*no_literal_pool*, page 305.

To set this option, use **Project>Options>Linker>Extra Options**.

--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_range_reservations

Syntax `--no_range_reservations`

Description Normally, the linker reserves any ranges used by absolute symbols with a non-zero size, excluding them from consideration for `place in` commands.

When this option is used, these reservations are disabled, and the linker is free to place sections in such a way as to overlap the extent of absolute symbols.

To set this option, use **Project>Options>Linker>Extra Options**.

--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 117.

To set this option, use **Project>Options>Linker>Extra Options**.

--no_vfe

Syntax `--no_vfe`

Description Use this option to disable the Virtual Function Elimination optimization. All virtual functions in all classes with at least one instance will be kept, and Runtime Type Information data will be kept for all polymorphic classes. Also, no warning message will be issued for modules that lack VFE information.

See also --*vfe*, page 364 and *Virtual function elimination*, page 127.

To set related options, choose:

Project>Options>Linker>Optimizations>Perform C++ Virtual Function Elimination

--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.

--only_stdout

Syntax `--only_stdout`

Description Use this option to make the linker use the standard output stream (`stdout`), and 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 See *Rules for specifying a filename or directory as parameters*, page 274.

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 veneer is larger and slower than normal veneers.

See also

Veneers, page 123.

To set this option, use **Project>Options>Linker>Extra Options.**

--place_holder

Syntax

`--place_holder` *`symbol`*`[,`*`size`*`[,`*`section`*`[,`*`alignment`*`]]]`

Parameters

`symbol`	The name of the symbol to create
`size`	Size in ROM. Default is 4 bytes
`section`	Section name to use. Default is `.text`
`alignment`	Alignment of section. Default is `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 553.

To set this option, use **Project>Options>Linker>Extra Options**

--preconfig

Syntax

`--preconfig` *`filename`*

Parameters

See *Rules for specifying a filename or directory as parameters*, page 274.

Description

Use this option to make the linker read the specified file before reading the linker configuration file.

To set this option, use **Project>Options>Linker>Extra Options**.

--printf_multibytes

Syntax

```
--printf_multibytes
```

Description

Use this option to make the linker automatically select a `printf` formatter that supports multibytes.

Project>Options>General Options>Library options 1>Printf formatter

--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 references to an external symbol so that they refer to another symbol.

Note: Redirection will normally not affect references within a module.

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 269.

Project>Options>Linker>Diagnostics>Enable remarks

--scanf_multibytes

Syntax `--scanf_multibytes`

Description Use this option to make the linker automatically select a `scanf` formatter that supports multibytes.

Project>Options>General Options>Library options 1>Scanf formatter

--search, -L

Syntax `--search path`

`-L path`

Parameters

path	A path to a directory where the linker should search for object and library files.

Description Use this option to specify more directories for the linker to search for object and library files in.

By default, the linker searches for object and library files only in the working directory. Each use of this option on the command line adds another search directory.

See also *The linking process in detail*, page 97.

This option is not available in the IDE.

--semihosting

Syntax `--semihosting[=iar_breakpoint]`

Parameters

`iar_breakpoint`	The IAR-specific mechanism can be used when debugging applications that use SVC extensively.

Description

Use this option to include the debug interface—breakpoint mechanism—in the output image. If no parameter is specified, the behavior is as described in *The semihosting mechanism*, page 148.

See also

The semihosting mechanism, page 148.

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.

--stack_usage_control

Syntax

`--stack_usage_control=`*`filename`*

Parameters

See *Rules for specifying a filename or directory as parameters*, page 274.

Description

Use this option to specify a stack usage control file. This file controls stack usage analysis, or provides more stack usage information for modules or functions. You can use this option multiple times to specify multiple stack usage control files. If no filename extension is specified, the extension `suc` is used.

Using this option enables stack usage analysis in the linker.

See also

Stack usage analysis, page 105.

Project>Options>Linker>Advanced>Enable stack usage analysis>Control file

--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

--text_out

Syntax `--text_out{utf8|utf16le|utf16be|locale}`

Parameters

`utf8`	Uses the UTF-8 encoding
`utf16le`	Uses the UTF-16 little-endian encoding
`utf16be`	Uses the UTF-16 big-endian encoding
`locale`	Uses the system locale encoding

Description

Use this option to specify the encoding to be used when generating a text output file.

The default for the linker list files is to use the same encoding as the main source file. The default for all other text files is UTF-8 with a Byte Order Mark (BOM).

If you want text output in UTF-8 encoding without BOM, you can use the option `--no_bom` as well.

See also

--*no_bom*, page 351 and *Text encodings*, page 266.

Project>Options>Linker>Encodings>Text output file encoding

--threaded_lib

Syntax `--threaded_lib`

Description

Use this option to automatically configure the runtime library for use with threads.

When this option is used, the linker creates the sections `__iar_tls$$DATA` and `__iar_tls$$INIT_DATA`, and the sections `.tdata` and `.tbss` will continue to use the names `.tdata` and `.tbss`. If the option `--threaded_lib` is *not* used, the contents of the section .tdata will be handled as if they resided in .data and the contents of the section .tbss will be handled as if they resided in .bss.

Project>Options>General Options>Library Configuration>Enable thread support in library

--timezone_lib

Syntax --timezone_lib

Description Use this option to enable the time zone and daylight savings time functionality in the DLIB library.

Note: You need to implement the time zone functionality.

See also *__getzone*, page 158.

To set this option, use **Project>Options>Linker>Extra Options**.

--treat_rvct_modules_as_softfp

Syntax --treat_rvct_modules_as_softfp

Description Use this option to treat all modules generated by RVCT as using the standard (non-VFP) calling convention.

To set this option, use **Project>Options>Linker>Extra Options**.

--use_full_std_template_names

Syntax --use_full_std_template_names

Description In the demangled names of C++ entities, the linker by default uses shorter names for some classes. For example, "std::string" instead of "std::basic_string<char, std::char_traits<char>,std::allocator<char>>". Use this option to make the linker instead use the full, unabbreviated names.

This option is not available in the IDE.

--use_optimized_variants

Syntax

`--use_optimized_variants={no|auto|small|fast}`

Parameters

`no`
: Always uses the default variant with standard optimizations.

`auto`
: Uses variants based on AEABI attributes that indicate the requested optimization goal:

If a module is compiled with `-Ohs`, and the DLIB library contains a fast variant of a function that is referenced in the module, that variant is used.

If all modules referencing a function are compiled with `-Ohz`, and the DLIB library contains a small variant of that function, that variant is used.

This is the default behavior of the linker.

`small`
: Always uses a small variant (balances code size and execution speed, favoring size) if there is one in the DLIB library.

`fast`
: Always uses a fast variant (maximum execution speed) if there is one in the DLIB library.

Description

Use this option to control the use of optimized variants of some DLIB library functions. (Some DLIB libraries delivered with the product contain optimized variants, such as a small integer division routing for Cortex-M0, or a fast `strcpy` implementation for cores that support the Thumb-2 ISA architecture.)

To see which variants that this option selected, inspect the list of redirects in the linker map file.

To set this option, use **Project>Options>Linker>Extra Options**.

--utf8_text_in

Syntax

`--utf8_text_in`

Description

Use this option to specify that the linker shall use the UTF-8 encoding when reading a text input file with no Byte Order Mark (BOM).

Note: This option does not apply to source files.

See also

Text encodings, page 266.

Project>Options>Linker>Encodings>Default input file encoding

--version

Syntax

```
--version
```

Description

Use this option to make the linker send version information to the console and then exit.

This option is not available in th IDE.

--vfe

Syntax

```
--vfe[=forced]
```

Parameters

`forced`	Performs Virtual Function Elimination even if one or more modules lack the needed virtual function elimination information.

Description

By default, Virtual Function Elimination is always performed but requires that all object files contain the necessary virtual function elimination information. Use `--vfe=forced` to perform Virtual Function Elimination even if one or more modules do not have the necessary information.

Forcing the use of Virtual Function Elimination can be unsafe if some of the modules that lack the needed information perform virtual function calls or use dynamic Runtime Type Information.

See also

--*no_vfe*, page 355 and *Virtual function elimination*, page 127.

To set related options, choose:

Project>Options>Linker>Optimizations>Perform C++ Virtual Function Elimination

--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` will also be treated as errors when `--warnings_are_errors` is used.

See also
--*diag_warning*, page 290 and --*diag_warning*, page 339.

Project>Options>Linker>Diagnostics>Treat all warnings as errors

--whole_archive

Syntax
`--whole_archive filename`

Parameters
See *Rules for specifying a filename or directory as parameters*, page 274.

Description
Use this option to make the linker treat every object file in the archive as if it was specified on the command line. This is useful when an archive contains root content that is always included from an object file (filename extension `o`), but only included from an archive if some entry from the module is referred to.

Example
If `archive.a` contains the object files `file1.o`, `file2.o`, and `file3.o`, using `--whole_archive archive.a` is equivalent to specifying `file1.o file2.o file3.o`.

See also
Keeping modules, page 117.

To set this option, use **Project>Options>Linker>Extra Options**

Data representation

- Alignment
- Byte order (32-bit mode only)
- Basic data types—integer types
- Basic data types—floating-point types
- Pointer types
- Structure types
- Type qualifiers
- Data types in C++

See the chapter *Efficient coding for embedded applications* for information about which data types 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 strictest 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 379.

Note: With the `#pragma data_alignment` directive, you can increase the alignment demands on specific variables.

See also the Standard C file `stdalign.h`.

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 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.

For related information, see *--align_sp_on_irq*, page 282 and *--no_const_align*, page 303.

Byte order (32-bit mode only)

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 (**can only be selected in 32-bit mode**), 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 370.

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—integer types

The compiler supports both all Standard C basic data types and some additional types.

These topics are covered:

INTEGER TYPES—AN OVERVIEW

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
`signed int`	32 bits	-2^{31} to 2^{31}-1	4
`unsigned int`	32 bits	0 to 2^{32}-1	4
`signed long`			4
32-bit mode and ILP32 in 64-bit mode	32 bits	-2^{31} to 2^{31}-1	
LP64 in 64-bit mode	64 bits	-2^{63} to 2^{63}-1	
`unsigned long`			4
32-bit mode and ILP32 in 64-bit mode	32 bits	0 to 2^{32}-1	
LP64 in 64-bit mode	64 bits	0 to 2^{64}-1	
`signed long long`	64 bits	-2^{63} to 2^{63}-1	8
`unsigned long long`	64 bits	0 to 2^{64}-1	8

Table 32: Integer types

Signed variables are represented using the two's complement form.

BOOL

The bool data type is supported by default in the C++ language. If you have enabled language extensions, the bool type can also be used in C source code if you include the file stdbool.h. This will also enable the boolean values false and true.

THE 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};
```

See also the C++ enum struct syntax.

For related information, see *--enum_is_int*, page 294.

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: The library is compiled with the char type as unsigned.

THE WCHAR_T TYPE

The wchar_t data type is 4 bytes and the encoding used for it is UTF-32.

THE CHAR16_T TYPE

The char16_t data type is 2 bytes and the encoding used for it is UTF-16.

THE CHAR32_T TYPE

The char32_t data type is 4 bytes and the encoding used for it is UTF-32.

BITFIELDS

In Standard C, int, signed int, and unsigned int can be used as the base type for integer bitfields. In standard C++, and in C when language extensions are enabled in the compiler, any integer or enumeration type can be used as the base type. It is implementation defined whether a plain integer type (`char`, `short`, `int`, etc) results in a signed or unsigned bitfield.

In the IAR C/C++ Compiler for Arm, plain integer types are treated as unsigned.

Bitfields in expressions are treated as `int` if `int` can represent all values of the bitfield. Otherwise, they are treated as the bitfield base type.

Each bitfield is placed in the next suitably aligned container of its base type that has enough available bits to accommodate the bitfield. Within each container, the bitfield is placed in the first available byte or bytes, taking the byte order into account. Note that containers can overlap if needed, as long as they are suitably aligned for their type.

In addition, the compiler supports an alternative bitfield allocation strategy (disjoint types), where bitfield containers of different types are not allowed to overlap. Using this allocation strategy, each bitfield is placed in a new container if its type is different from that of the previous bitfield, or if the bitfield does not fit in the same container as the previous bitfield. Within each container, the bitfield is placed from the least significant bit to the most significant bit (disjoint types) or from the most significant bit to the least significant bit (reverse disjoint types). This allocation strategy will never use less space than the default allocation strategy (joined types), and can use significantly more space when mixing bitfield types.

Use the `#pragma bitfields` directive to choose which bitfield allocation strategy to use, see *bitfields*, page 404.

Assume this example:

```
struct BitfieldExample
{
  uint32_t a : 12;
  uint16_t b : 3;
  uint16_t c : 7;
  uint8_t  d;
};
```

The example in the joined types bitfield allocation strategy

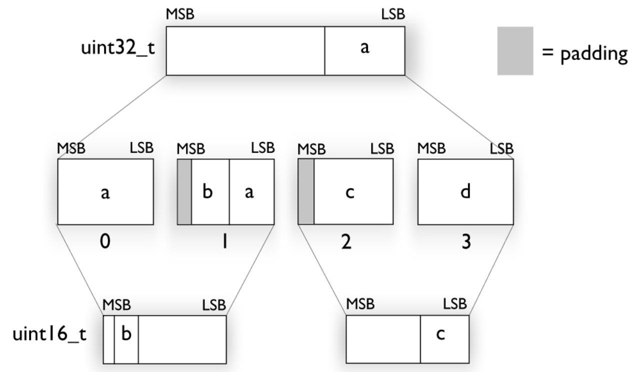
To place the first bitfield, `a`, the compiler allocates a 32-bit container at offset `0` and puts `a` into the first and second bytes of the container.

For the second bitfield, `b`, a 16-bit container is needed and because there are still four bits free at offset `0`, the bitfield is placed there.

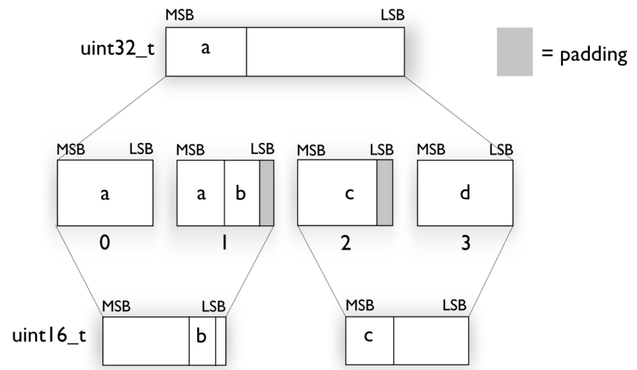
For the third bitfield, `c`, as there is now only one bit left in the first 16-bit container, a new container is allocated at offset `2`, and `c` is placed in the first byte of this container.

The fourth member, d, can be placed in the next available full byte, which is the byte at offset 3.

In little-endian mode, each bitfield is allocated starting from the least significant free bit of its container to ensure that it is placed into bytes from left to right.



In big-endian mode, each bitfield is allocated starting from the most significant free bit of its container to ensure that it is placed into bytes from left to right.



The example in the disjoint types bitfield allocation strategy

To place the first bitfield, a, the compiler allocates a 32-bit container at offset 0 and puts a into the least significant 12 bits of the container.

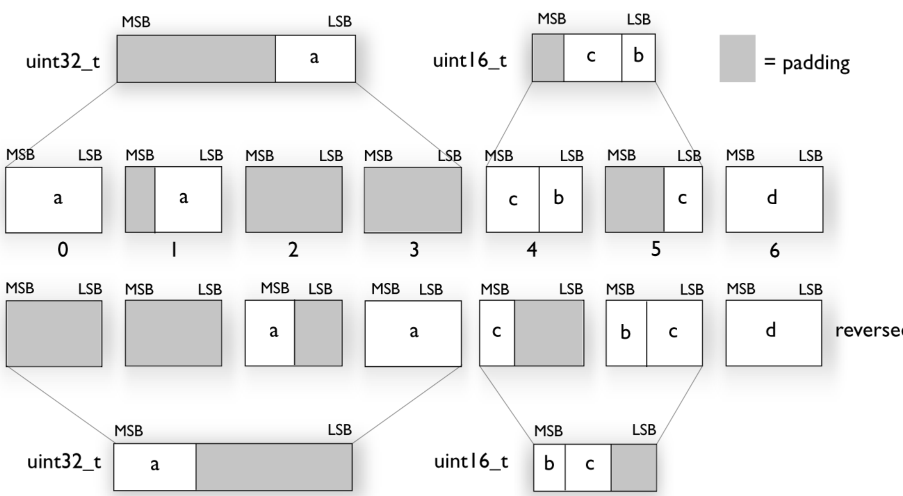
To place the second bitfield, b, a new container is allocated at offset 4, because the type of the bitfield is not the same as that of the previous one. b is placed into the least significant three bits of this container.

The third bitfield, `c`, has the same type as `b` and fits into the same container.

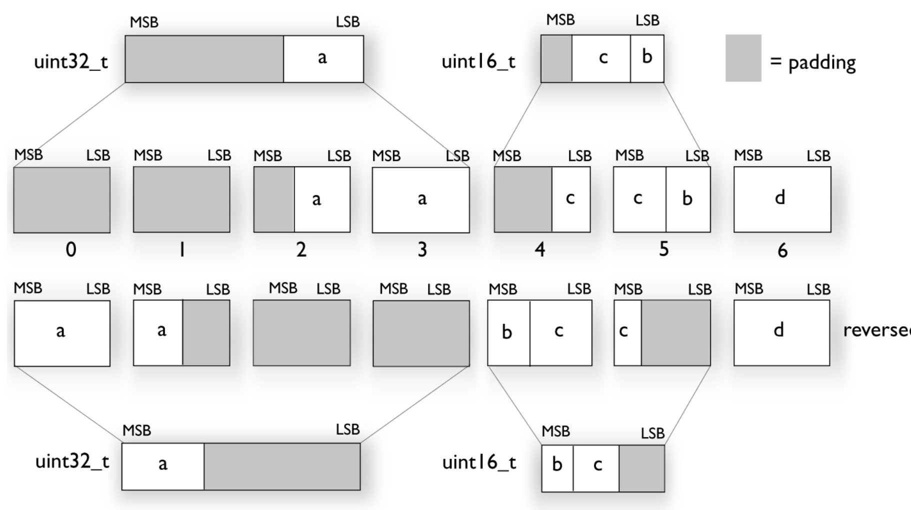
The fourth member, `d`, is allocated into the byte at offset `6`. `d` cannot be placed into the same container as `b` and `c` because it is not a bitfield, it is not of the same type, and it would not fit.

When using reverse order (reverse disjoint types), each bitfield is instead placed starting from the most significant bit of its container.

This is the layout of `bitfield_example` in little-endian mode:



This is the layout of `bitfield_example` in big-endian mode:



Basic data types—floating-point types

In the IAR C/C++ Compiler for Arm, floating-point values are represented in standard IEC 60559 format. The sizes for the different floating-point types are:

Type	Size	Range (+/-)	Decimals	Exponent	Mantissa	Alignment
`__fp16`	16 bits	±2E-14 to 65504	3	5 bits	11 bits	2
`float`	32 bits	±1.18E-38 to ±3.40E+38	7	8 bits	23 bits	4
`double`	64 bits	±2.23E-308 to ±1.79E+308	15	11 bits	52 bits	8
`long double`	64 bits	±2.23E-308 to ±1.79E+308	15	11 bits	52 bits	8

Table 33: 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 376.

The `__fp16` floating-point type is only a storage type. All numerical operations will operate on values promoted to `float`. There is also a standard type `_Float16`, which is layout-compatible with `__fp16`. Some cores support numerical operations directly on `_Float16` values. For other cores, it is a storage-only type.

Note: The C/C++ standard library does *not* support the `_Float16` type. If you want to use any of the standard library functions on the `_Float16` type, you must cast the `_Float16` value to single-precision or double-precision, and then use the appropriate library function. Because there is no string format specifier for the `_Float16` type, an explicit cast is required.

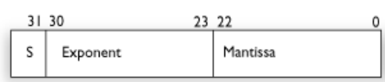
FLOATING-POINT ENVIRONMENT

Exception flags for floating-point values are supported for devices with a VFP unit, and they are defined in the `fenv.h` file. For devices without a VFP unit, the functions defined in the `fenv.h` file exist but have no functionality.

The `feraiseexcept` function does not raise an `inexact` floating-point exception when called with `FE_OVERFLOW` or `FE_UNDERFLOW`.

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`

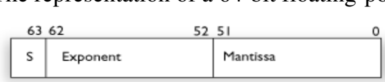
The range of the number is at least:

`±1.18E-38 to ±3.39E+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 at least:

`±2.23E-308 to ±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 the most significant bit in the mantissa to 1. The value of the sign bit is ignored.
- 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 subnormal, even though the number is treated as if the exponent was 1. Unlike normal numbers, subnormal numbers do not have an implicit 1 as the most significant bit (the MSB) of the mantissa. The value of a subnormal 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 function pointers have these properties:

Execution mode	Data model	Pointer size	Address range
32-bit	n/a	32 bits	0–0xFFFF'FFFF
64-bit	ILP32	32 bits	0–0xFFFF'FFFF
64-bit	LP64	64 bits	0–0xFFFF'FFFF'FFFF'FFFF

Table 34: Function pointers

Note: In the ILP32 data model, the representation of a pointer in a register is always 64-bit. A 32-bit pointer is zero-extended when it is loaded into a register, and a store operation only stores the lowest 32 bits.

When function pointer types are declared, attributes are inserted before the * sign, for example:

```
typedef void (__thumb * IntHandler) (void);
```

This can be rewritten using `#pragma` directives:

```
#pragma type_attribute=__thumb
typedef void IntHandler_function(void);
typedef IntHandler_function *IntHandler;
```

DATA POINTERS

There is one data pointer available. It has these properties:

Execution mode	Data model	Pointer size	Address range
32-bit	n/a	32 bits	`0-0xFFFF'FFFF`
64-bit	ILP32	32 bits	`0-0xFFFF'FFFF`
64-bit	LP64	64 bits	`0-0xFFFF'FFFF'FFFF'FFFF`

Table 35: Data pointers

Note: In the ILP32 data model, the representation of a pointer in a register is always 64-bit. A 32-bit pointer is zero-extended when it is loaded into a register, and a store operation only stores the lowest 32 bits.

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 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
- Casting a value of an unsigned integer type to a pointer of a larger type is performed by zero extension

size_t

`size_t` is the unsigned integer type of the result of the `sizeof` operator. In 32-bit mode and when using the ILP32 data model in 64-bit mode, the type used for `size_t` is `unsigned int`. In the LP64 data model, the type used for `size_t` is `unsigned long`.

ptrdiff_t

`ptrdiff_t` is the signed integer type of the result of subtracting two pointers. In 32-bit mode and when using the ILP32 data model in 64-bit mode, the type used for `ptrdiff_t` is the signed integer variant of the `size_t` type. In the LP64 data model, the type used for `ptrdiff_t` is `signed long`.

intptr_t

`intptr_t` is a signed integer type large enough to contain a `void *`. In the IAR C/C++ Compiler for Arm, the type used for `intptr_t` is `signed long int`.

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 OF STRUCTURE TYPES

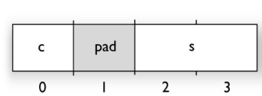
The `struct` and `union` types have the same alignment as the member with the highest alignment requirement—this alignment requirement also applies to a member that is a structure. To allow arrays of aligned structure objects, the size of a `struct` is adjusted to an even multiple of the alignment.

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).

```
struct First
{
  char c;
  short s;
} s;
```

This diagram shows the layout in memory:



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: 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.

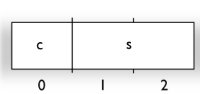
Special care is also needed when creating and using pointers to misaligned members. For direct access to misaligned members in a packed `struct`, the compiler will emit the correct (but slower and larger) code when needed. However, when a misaligned member is accessed through a pointer to the member, the normal (smaller and faster) code is used. In the general case, this will not work, because the normal code might depend on the alignment being correct.

This example declares a packed structure:

```
#pragma pack(1)
struct S
{
  char c;
  short s;
};

#pragma pack()
```

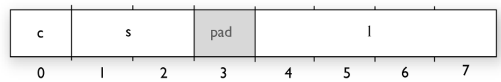
The structure `S` has this memory layout:



The next 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;
};
```

The structure `S2` has this memory 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 4.

For more information, see *Alignment of elements in a structure*, page 240.

Type qualifiers

According to the C standard, `volatile` and `const` are type qualifiers.

DECLARING OBJECTS VOLATILE

By declaring an object `volatile`, the compiler is informed that the value of the object can change beyond the compiler's control. The compiler must also assume that any accesses can have side effects—therefore all accesses to the `volatile` object must be preserved.

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 C 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
- Adding a `const` qualifier to a `volatile` object will make write accesses to the object impossible. However, the object will be placed in RAM as specified by the C standard.

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 combinations of object types not listed, only the rule that states that all accesses are preserved applies.

DECLARING OBJECTS VOLATILE AND CONST

If you declare a `volatile` object `const`, it will be write-protected but it will still be stored in RAM memory as the C standard specifies.

To store the object in read-only memory instead, but still make it possible to access it as a `const volatile` object, declare it with the `__ro_placement` attribute. See *__ro_placement*, page 396.

To store the object in read-only memory instead, but still make it possible to access it as a `const volatile` object, define the variable like this:

```
const volatile int x @ "FLASH";
```

The compiler will generate the read/write section FLASH. That section should be placed in ROM and is used for manually initializing the variables when the application starts up.

Thereafter, the initializers can be reflashed with other values at any time.

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 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.

Extended keywords

- General syntax rules for extended keywords
- Summary of extended keywords
- Descriptions of extended keywords
- Supported GCC attributes

General syntax rules for extended keywords

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 more information about each attribute, see *Descriptions of extended keywords*, page 387.

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 293.

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 explicitly in your declarations, or use the pragma directive `#pragma type_attribute`.

General type attributes

Available *function type attributes* (affect how the function should be called):

`__arm`, `__cmse_nonsecure_call`, `__exception`, `__fiq`, `__interwork`, `__irq`, `__svc`, `__swi__no_scratch`, `__task`, `__thumb`

Available *data type attributes*:

`__big_endian`, `__little_endian`, `__packed`

You can specify as many type attributes as required for each level of pointer indirection.

Note: Data type attributes (except `__packed`) are not allowed on structure type fields.

Syntax for type attributes used on data objects

If you select the *uniform attribute syntax*, data type attributes use the same syntax rules as the type qualifiers `const` and `volatile`.

If not, data type attributes use almost the same syntax rules as the type qualifiers `const` and `volatile`. For example:

```
__little_endian int i;
int __little_endian j;
```

Both `i` and `j` will be accessed with little-endian byte order.

Unlike `const` and `volatile`, when a type attribute is used before the type specifier in a derived type, the type attribute applies to the object, or typedef itself, except in structure member declarations.

Using a type definition can sometimes make the code clearer:

```
typedef __packed int packed_int;
packed_int *q1;
```

`packed_int` is a typedef for packed integers. The variable `q1` can point to such integers.

You can also use the `#pragma type_attributes` directive to specify type attributes for a declaration. The type attributes specified in the pragma directive are applied to the data object or typedef being declared.

```
#pragma type_attribute=__packed
int * q2;
```

The variable `q2` is packed.

For more information about the uniform attribute syntax, see --*uniform_attribute_syntax*, page 322 and --*no_uniform_attribute_syntax*, page 310.

Syntax for type attributes used 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 inside the parentheses for function pointers, for example:

```
__irq __arm void my_handler(void);
```

or

```
void (__irq __arm * my_fp)(void);
```

You can also use `#pragma type_attribute` to specify the function type attributes:

```
#pragma type_attribute=__irq __arm
void my_handler(void);

#pragma type attribute=__irq __arm
typedef void my_fun_t(void);
my_fun_t * my_fp;
```

OBJECT ATTRIBUTES

Normally, object attributes affect the internal functionality of functions and data objects, but not directly how the function is called or how the data is accessed. This means that an object attribute does not normally need to be present in the declaration of an object.

These object attributes are available:

- Object attributes that can be used for variables:

  `__absolute`, `__no_alloc`, `__no_alloc16`, `__no_alloc_str`, `__no_alloc_str16`, `__no_init`, `__ro_placement`
- Object attributes that can be used for functions and variables:

  `location`, `@`, `__root`, `__weak`
- Object attributes that can be used for functions:

  `__cmse_nonsecure_entry`, `__intrinsic`, `__naked`, `__nested`, `__noreturn`, `__ramfunc`, `__stackless`

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 242.

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
`__absolute`	Makes references to the object use absolute addressing
`__arm`	Makes a function execute in Arm mode
`__big_endian`	Declares a variable to use the big-endian byte order
`__cmse_nonsecure_call`	Declares a function pointer to call non-secure code
`__cmse_nonsecure_entry`	Makes a function callable from a non-secure image
`__exception`	Declares a 64-bit mode exception function
`__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
`__naked`	Declares a function without generating code to set up or tear down the function's frame
`__nested`	Allows an `__irq` declared interrupt function to be nested, that is, interruptible by the same type of interrupt
`__no_alloc`, `__no_alloc16`	Makes a constant available in the execution file
`__no_alloc_str`, `__no_alloc_str16`	Makes a string literal available in the execution file

Table 36: Extended keywords summary

Extended keyword	Description
`__no_init`	Places a data object in non-volatile memory
`__noreturn`	Informs the compiler that the function will not return
`__packed`	Decreases data type alignment to 1
`__pcrel`	Used internally by the compiler for constant data when the `--ropi` compiler option is used
`__ramfunc`	Makes a function execute in RAM
`__ro_placement`	Places `const volatile` data in read-only memory.
`__root`	Ensures that a function or variable is included in the object code even if unused
`__sbrel`	Used internally by the compiler for constant data when the `--rwpi` compiler option is used
`__stackless`	Makes a function callable without a working stack
`__svc`	Declares a 32-bit mode software interrupt function or a 64-bit mode software exception function
`__swi`	Alias for `__svc`
`__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 36: Extended keywords summary (Continued)

Descriptions of extended keywords

This section gives detailed information about each extended keyword.

__absolute

Syntax

See *Syntax for object attributes*, page 386.

Description

The `__absolute` keyword makes references to the object use absolute addressing.

The following limitations apply:

- Only available when the `--ropi` or `--rwpi` compiler option is used
- Can only be used on external declarations.

Example

```
extern __absolute char otherBuffer[100];
```

__arm

Syntax

See *Syntax for type attributes used on functions*, page 385.

Description

The `__arm` keyword makes a function execute in Arm mode.

A function declared `__arm` cannot be declared `__thumb`.

In 64-bit mode, this keyword cannot be used.

Example

```
__arm int func1(void);
```

__big_endian

Syntax

See *Syntax for type attributes used on data objects*, page 384.

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. It cannot be used **in 64-bit mode**.

Note: If this keyword is used with a pointer, that pointer will not be compatible with pointers that do not have a `__big_endian` or `__little_endian` attribute.

Example

```
__big_endian long my_variable;
```

See also

__little_endian, page 391.

__cmse_nonsecure_call

Syntax

See *Syntax for type attributes used on functions*, page 385.

Description

The keyword `__cmse_nonsecure_call` can be used on a function pointer, and indicates that a call via the pointer will enter non-secure state. The execution state will be cleared up before such a call, to avoid leaking sensitive data to the non-secure state.

The `__cmse_nonsecure_call` keyword can only be used with a function pointer, and it is only allowed when compiling with `--cmse`.

The keyword `__cmse_nonsecure_call` is not supported for variadic functions, for functions with parameters or return values that do not fit in registers, or for functions with parameters or return values in floating-point registers.

In 64-bit mode, this keyword cannot be used.

Example

```
#include <arm_cmse.h>
typedef __cmse_nonsecure_call void (*fp_ns_t)(void);
static fp_ns_t callback_ns = 0;
__cmse_nonsecure_entry void set_callback_ns(fp_ns_t func_ns) {
  callback_ns = cmse_nsfptr_create(func_ns);
}
```

See also

--*cmse*, page 284.

__cmse_nonsecure_entry

Syntax

See *Syntax for object attributes*, page 386.

Description

The `__cmse_nonsecure_entry` keyword declares an entry function that can be called from the non-secure state. The execution state will be cleared before returning to the caller, to avoid leaking sensitive data to the non-secure state.

The keyword `__cmse_nonsecure_entry` is not supported for variadic functions or functions with parameters or return values that do not fit in registers.

The keyword `__cmse_nonsecure_entry` is only allowed when compiling with `--cmse`.

In 64-bit mode, this keyword cannot be used.

Example

```
#include <arm_cmse.h>
__cmse_nonsecure_entry int secure_add(int a, int b) {
  return cmse_nonsecure_caller() ? a + b : 0;
}
```

See also

--*cmse*, page 284.

__exception

Syntax

See *Syntax for type attributes used on functions*, page 385.

Description

The `__exception` keyword makes a function usable in one of the exception vectors in the 64-bit mode exception table.

Example

```
__exception void my_function(void);
```

See also

Exception functions for 64-bit mode, page 86.

__fiq

Syntax

See *Syntax for type attributes used on functions*, page 385.

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. This keyword is not available when you compile for Cortex-M devices.

In 64-bit mode, this keyword cannot be used.

Example

```
__fiq __arm void interrupt_function(void);
```

__interwork

Syntax

See *Syntax for type attributes used on functions*, page 385.

Description

A function declared `__interwork` can be called from functions executing in either Arm or Thumb mode. **In 64-bit mode**, this keyword cannot be used.

Note: All functions are interwork. The keyword exists for compatibility reasons.

Example

```
typedef void (__thumb __interwork *IntHandler)(void);
```

__intrinsic

Description

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

__irq

Syntax

See *Syntax for type attributes used on functions*, page 385.

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. This keyword is not available when you compile for Cortex-M devices. **In 64-bit mode**, this keyword cannot be used.

Example

```
__irq __arm void interrupt_function(void);
```

See also

--align_sp_on_irq, page 282.

__little_endian

Syntax

See *Syntax for type attributes used on data objects*, page 384.

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. It cannot be used **in 64-bit mode**.

Note: If this keyword is used with a pointer, that pointer will not be compatible with pointers that do not have a `__big_endian` or `__little_endian` attribute.

Example

```
__little_endian long my_variable;
```

See also

__big_endian, page 388.

__naked

Syntax

See *Syntax for object attributes*, page 386.

Description

This keyword declares a function for which the compiler does not generate code to set up or tear down the function's frame.

The compiler is severely limited by not having a frame layout, so the body of the declared function body should consist of `__asm` statements. Using extended assembly, parameter references, or mixing C code with `__asm` statements might not work reliably.

Note: It is not possible to call a function declared with the `__naked` keyword.

Example

```
__naked void save_process_state(void);
__naked void restore_process_state(void);
```

__nested

Syntax

See *Syntax for object attributes*, page 386.

Description

In 32-bit mode, 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.

In 64-bit mode, the `__nested` keyword modifies the enter and exit of an exception function to allow for nested exceptions. See *Exception functions for 64-bit mode*, page 86.

Example

```
__irq __nested __arm void interrupt_handler(void);
```

See also

Nested interrupts, page 84 and --*align_sp_on_irq*, page 282.

__no_alloc, __no_alloc16

Syntax

See *Syntax for object attributes*, page 386.

Description

Use the `__no_alloc` or `__no_alloc16` object attribute on a constant to make the constant available in the executable file without occupying any space in the linked application.

You cannot access the contents of such a constant from your application. You can take its address, which is an integer offset to the section of the constant. The type of the offset is `unsigned long` when `__no_alloc` is used, and `unsigned short` when `__no_alloc16` is used.

Example

```
__no_alloc const struct MyData my_data @ "XXX" = {...};
```

See also

__no_alloc_str, __no_alloc_str16, page 392.

__no_alloc_str, __no_alloc_str16

Syntax

`__no_alloc_str(`*`string_literal`* `@` *`section`*`)`

and

`__no_alloc_str16(`*`string_literal`* `@` *`section`*`)`

where

`string_literal`	The string literal that you want to make available in the executable file.
`section`	The name of the section to place the string literal in.

Description

Use the `__no_alloc_str` or `__no_alloc_str16` operators to make string literals available in the executable file without occupying any space in the linked application.

The value of the expression is the offset of the string literal in the section. For `__no_alloc_str`, the type of the offset is `unsigned long`. For `__no_alloc_str16`, the type of the offset is `unsigned short`.

Example

```
#define MYSEG "YYY"
#define X(str) __no_alloc_str(str @ MYSEG)

extern void dbg_printf(unsigned long fmt, ...)

#define DBGPRINTF(fmt, ...) dbg_printf(X(fmt), __VA_ARGS__)

void
foo(int i, double d)
{
  DBGPRINTF("The value of i is: %d, the value of d is: %f",i,d);
}
```

Depending on your debugger and the runtime support, this could produce trace output on the host computer.

Note: There is no such runtime support in C-SPY, unless you use an external plugin module.

See also

__no_alloc, __no_alloc16, page 392.

__no_init

Syntax

See *Syntax for object attributes*, page 386.

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

Non-initialized variables, page 257 and *do not initialize directive*, page 520.

__noreturn

Syntax

See *Syntax for object attributes*, page 386.

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`.

Note: At optimization levels Medium or High, the `__noreturn` keyword might cause incorrect call stack debug information at any point where it can be determined that the current function cannot return.

Note: The extended keyword `__noreturn` has the same meaning as the Standard C keyword `_Noreturn` or the macro `noreturn` (if `stdnoreturn.h` has been included) and as the Standard C++ attribute `[[noreturn]]`.

Example

```
__noreturn void terminate(void);
```

__packed

Syntax

See *Syntax for type attributes used on data objects*, page 384. An exception is when the keyword is used for modifying the structure type in a `struct` or `union` declarations, see below.

Description

Use the `__packed` keyword to specify a data alignment of 1 for a data type. `__packed` can be used in two ways:

- When used before the `struct` or `union` keyword in a structure definition, the maximum alignment of each member in the structure is set to 1, eliminating the need for gaps between the members.

  You can also use the `__packed` keyword with structure declarations, but it is illegal to refer to a structure type defined without the `__packed` keyword using a structure declaration with the `__packed` keyword.
- When used in any other position, it follows the syntax rules for type attributes, and affects a type in its entirety. A type with the `__packed` type attribute is the same as the type attribute without the `__packed` type attribute, except that it has a data 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.

Use either `__packed` or `#pragma pack` to relax the alignment restrictions for a type and the objects defined using that type. Mixing `__packed` and `#pragma pack` might lead to unexpected behavior.

Example

```
/* No pad bytes in X: */
__packed struct X { char ch; int i; };
/* __packed is optional here: */
struct X * xp;

/* NOTE: no __packed: */
struct Y { char ch; int i; };
/* ERROR: Y not defined with __packed: */
__packed struct Y * yp ;

/* Member 'i' has alignment 1: */
struct Z { char ch; __packed int i; };

void Foo(struct X * xp)
{
  /* Error:"int __packed *" -> "int *" not allowed: */
  int * p1 = &xp->1;
  /* OK: */
  int __packed * p2 = &xp->i;
  /* OK, char not affected */
  char * p3 = &xp->ch;
}
```

See also

pack, page 418.

__ramfunc

Syntax

See *Syntax for object attributes*, page 386.

Description

The `__ramfunc` keyword makes a function execute in RAM. Two code sections will be created: one for the RAM execution (`.textrw`), and one for the ROM initialization (`.textrw_init`).

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 can safely ignore or disable these warnings.

Functions declared `__ramfunc` are by default stored in the section named `.textrw`.

Example

```
__ramfunc int FlashPage(char * data, char * page);
```

See also

The *C-SPY® Debugging Guide for Arm* to read more about `__ramfunc` declared functions in relation to breakpoints.

__ro_placement

Syntax

See *Syntax for object attributes*, page 386.

Description

The `__ro_placement` attribute specifies that a data object should be placed in read-only memory. There are two cases where you might want to use this object attribute:

- Data objects declared `const volatile` are by default placed in read-write memory. Use the `__ro_placement` object attribute to place the data object in read-only memory instead.
- In C++, a data object declared `const` and that needs dynamic initialization is placed in read-write memory and initialized at system startup. If you use the `__ro_placement` object attribute, the compiler will give an error message if the data object needs dynamic initialization.

You can only use the `__ro_placement` object attribute on `const` objects.

You can use the `__ro_placement` attribute with C++ objects if the compiler can optimize the C++ dynamic initialization of the data objects into static initialization. This is possible only for relatively simple constructors that have been defined in the header files of the relevant class definitions, so that they are visible to the compiler. If the compiler cannot find the constructor, or if the constructor is too complex, an error message will be issued (`Error[Go023]`) and the compilation will fail.

Example

```
__ro_placement const volatile int x = 10;
```

__root

Syntax

See *Syntax for object attributes*, page 386.

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

For more information about root symbols and how they are kept, see *Keeping symbols and sections*, page 117.

__stackless

Syntax

See *Syntax for object attributes*, page 386.

Description

The `__stackless` keyword declares a function that can be called without a working stack.

A function declared `__stackless` violates the calling convention in such a way that it is not possible to return from it. However, the compiler cannot reliably detect if the function returns and will not issue an error if it does.

Example

```
__stackless void start_application(void);
```

__svc

Syntax

See *Syntax for type attributes used on functions*, page 385.

Description

32-bit mode:

The `__svc` 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 `__svc` accepts arguments and returns values. The `__svc` 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 `__svc` keyword also expects a software interrupt number which is specified with the `#pragma svc_number=`*`number`* directive. The `svc_number` is used as an argument to the generated assembler `SVC` instruction, and can be used by the SVC interrupt handler, for example `SVC_Handler`, to select one software interrupt function in a system that contains several such functions.

Note: 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.

64-bit mode:

The `__svc` keyword makes a function usable in one of the exception vectors in the 64-bit Arm exception table. See *Exception functions for 64-bit mode*, page 86.

Example

To declare your software interrupt function, typically in a header file, write for example like this:

```
#pragma svc_number=0x23
__svc int svc0x23_function(int a, int b);
...
```

To call the function:

```
...
int x = svc0x23_function(1, 2);  /* Will be replaced by SVC 0x23,
                                    hence the linker will never
                                    try to locate the
                                    svc0x23_function */
...
```

Somewhere in your application source code, you define your software interrupt function:

```
...
__svc __arm int the_actual_svc0x23_function(int a, int b)
{
  ...
  return 42;
}
```

See also

Software interrupts, page 85, *Calling convention*, page 185, and *Exception functions for 64-bit mode*, page 86.

__task

Syntax

See *Syntax for type attributes used on functions*, page 385.

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 See *Syntax for type attributes used on functions*, page 385.

Description The `__thumb` keyword makes a function execute in Thumb mode.

A function declared `__thumb` cannot be declared `__arm`.

In 64-bit mode, this keyword cannot be used.

Example

```
__thumb int func2(void);
```

__weak

Syntax See *Syntax for object attributes*, page 386.

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;
}
```

Supported GCC attributes

In extended language mode, the IAR C/C++ Compiler also supports a limited selection of GCC-style attributes. Use the `__attribute__ ((`*`attribute-list`*`))` syntax for these attributes.

The following attributes are supported in part or in whole. For more information, see the GCC documentation.

- `alias`
- `aligned`
- `always_inline`
- `cmse_nonsecure_call`
- `cmse_nonsecure_entry`
- `const`
- `constructor`
- `deprecated`
- `naked`
- `noinline`
- `noreturn`
- `packed`
- `pcs` (for IAR type attributes used on functions)
- `pure`
- `section`
- `target` (for IAR object attributes used on functions)
- `transparent_union`
- `unused`
- `used`
- `volatile`
- `weak`

Pragma directives

- Summary of pragma directives
- Descriptions of pragma directives

Summary of pragma directives

The #pragma directive is defined by Standard C 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.

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.
`calls`	Lists possible called functions for indirect calls.
`call_graph_root`	Specifies that the function is a call graph root.
`cstat_disable`	See the *C-STAT® Static Analysis Guide*.
`cstat_enable`	See the *C-STAT® Static Analysis Guide*.
`cstat_restore`	See the *C-STAT® Static Analysis Guide*.
`cstat_suppress`	See the *C-STAT® Static Analysis Guide*.
`data_alignment`	Gives a variable a higher (more strict) alignment.
`default_function_attributes`	Sets default type and object attributes for declarations and definitions of functions.
`default_no_bounds`	Applies `#pragma no_bounds` to a whole set of functions. See the C-RUN documentation in the *C-SPY® Debugging Guide for Arm*.
`default_variable_attributes`	Sets default type and object attributes for declarations and definitions of variables.

Table 37: Pragma directives summary

Pragma directive	Description
define_with_bounds	Instruments a function to track pointer bounds. See the C-RUN documentation in the *C-SPY® Debugging Guide for Arm*.
define_without_bounds	Defines the version of a function that does not have extra bounds information. See the C-RUN documentation in the *C-SPY® Debugging Guide for Arm*.
deprecated	Marks an entity as deprecated.
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.
disable_check	Specifies that the immediately following function does not check accesses against bounds. See the C-RUN documentation in the *C-SPY® Debugging Guide for Arm*.
error	Signals an error while parsing.
function_category	Declares function categories for stack usage analysis.
generate_entry_without_bounds	Enables generation of an extra entry without bounds for the immediately following function. See the C-RUN documentation in the *C-SPY® Debugging Guide for Arm*.
include_alias	Specifies an alias for an include file.
inline	Controls inlining of a function.
language	Controls the IAR Systems language extensions.
location	Specifies the absolute address of a variable, places a variable in a register, or places groups of functions or variables in named sections.
message	Prints a message.
no_arith_checks	Specifies that no C-RUN arithmetic checks will be performed in the following function. See the C-RUN documentation in the *C-SPY® Debugging Guide for Arm*.

Table 37: Pragma directives summary (Continued)

Pragma directive	Description
`no_bounds`	Specifies that the immediately following function is not instrumented for bounds checking. See the C-RUN documentation in the *C-SPY® Debugging Guide for Arm*.
`no_stack_protect`	Disables stack protection for the following function.
`object_attribute`	Adds object attributes to the declaration or definition of a variable or function.
`optimize`	Specifies the type and level of an optimization.
`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.
`public_equ`	Defines a public assembler label and gives it a value.
`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.
`segment`	This directive is an alias for #pragma section.
`section_prefix`	Adds a prefix to the names of sections.
`stack_protect`	Forces stack protection for the function that follows.
`STDC CX_LIMITED_RANGE`	Specifies whether the compiler can use normal complex mathematical formulas or not.
`STDC FENV_ACCESS`	Specifies whether your source code accesses the floating-point environment or not.
`STDC FP_CONTRACT`	Specifies whether the compiler is allowed to contract floating-point expressions or not.
`svc_number`	Sets the interrupt number of a software interrupt (**32-bit mode**) or exception (**64-bit mode**).
`type_attribute`	Adds type attributes to a declaration or to definitions.
`unroll`	Unrolls loops.

Table 37: Pragma directives summary (Continued)

Pragma directive	Description
vectorize	Enables or disables generation of NEON vector instructions for a loop.
weak	Makes a definition a weak definition, or creates a weak alias for a function or a variable.

Table 37: Pragma directives summary (Continued)

Note: For portability reasons, see also *Recognized pragma directives (6.10.6)*, page 631.

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 information, see *Bitfields*, page 370.
`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 the 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 bitfield types. */
struct S
{
  unsigned char  error : 1;
  unsigned char  size  : 4;
  unsigned short code  : 10;
};
#pragma bitfields=default /* Restores to default setting. */
```

See also

Bitfields, page 370.

calls

Syntax

`#pragma calls=arg[, arg...]`

Parameters

arg can be one of these:

function	A declared function
category	A string that represents the name of a function category

Description

Use this pragma directive to specify all functions that can be indirectly called in the following statement. This information can be used for stack usage analysis in the linker. You can specify individual functions or function categories. Specifying a category is equivalent to specifying all included functions in that category.

Example

```
void Fun1(), Fun2();

void Caller(void (*fp)(void))
{
#pragma calls = Fun1, Fun2, "Cat1"
  (*fp)();        // Can call Fun1, Fun2, and all
                  // functions in category "Cat1"
}
```

See also

function_category, page 412 and *Stack usage analysis*, page 105.

call_graph_root

Syntax

```
#pragma call_graph_root[=category]
```

Parameters

category — A string that identifies an optional call graph root category

Description

Use this pragma directive to specify that, for stack usage analysis purposes, the immediately following function is a call graph root. You can also specify an optional category. The compiler will usually automatically assign a call graph root category to interrupt and task functions. If you use the `#pragma call_graph_root` directive on such a function you will override the default category. You can specify any string as a category.

Example

```
#pragma call_graph_root="interrupt"
```

See also

Stack usage analysis, page 105.

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 the immediately following 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 therefore be used for creating situations where the size is not a multiple of the alignment.

Note: To comply with the ISO C11 standard and later, it is recommended to use the alignment specifier `_Alignas` for C code. To comply with the C++11 standard and later, it is recommended to use the alignment specifier `alignas` for C++ code.

default_function_attributes

Syntax

```
#pragma default_function_attributes=[ attribute...]
```

where *attribute* can be:

```
type_attribute
object_attribute
@ section_name
```

Parameters

type_attribute	See *Type attributes*, page 383.
object_attribute	See *Object attributes*, page 385.
@ *section_name*	See *Data and function placement in sections*, page 244.

Description

Use this pragma directive to set default section placement, type attributes, and object attributes for function declarations and definitions. The default settings are only used for declarations and definitions that do not specify type or object attributes or location in some other way.

Specifying a `default_function_attributes` pragma directive with no attributes, restores the initial state where no such defaults have been applied to function declarations and definitions.

Example

```
/* Place following functions in section MYSEC" */
#pragma default_function_attributes = @ "MYSEC"
int fun1(int x) { return x + 1; }
int fun2(int x) { return x - 1;
/* Stop placing functions into MYSEC */
#pragma default_function_attributes =
```

has the same effect as:

```
int fun1(int x) @ "MYSEC" { return x + 1; }
int fun2(int x) @ "MYSEC" { return x - 1; }
```

See also

location, page 414.

object_attribute, page 416.

type_attribute, page 424.

default_variable_attributes

Syntax

```
#pragma default_variable_attributes=[ attribute...]
```

where *attribute* can be:

```
type_attribute
object_attribute
@ section_name
```

Parameters

type_attribute	See *Type attributes*, page 383.
object_attributes	See *Object attributes*, page 385.
@ *section_name*	See *Data and function placement in sections*, page 244.

Description

Use this pragma directive to set default section placement, type attributes, and object attributes for declarations and definitions of variables with static storage duration. The default settings are only used for declarations and definitions that do not specify type or object attributes or location in some other way.

Specifying a `default_variable_attributes` pragma directive with no attributes restores the initial state of no such defaults being applied to variables with static storage duration.

Note: The extended keyword `__packed` can be used in two ways: as a normal type attribute and in a structure type definition. The pragma directive `default_variable_attributes` only affects the use of `__packed` as a type attribute. Structure definitions are not affected by this pragma directive. See *__packed*, page 394.

Example

```
/* Place following variables in section MYSEC" */
#pragma default_variable_attributes = @ "MYSEC"
int var1 = 42;
int var2 = 17;
/* Stop placing variables into MYSEC */
#pragma default_variable_attributes =
```

has the same effect as:

```
int var1 @ "MYSEC" = 42;
int var2 @ "MYSEC" = 17;
```

See also

location, page 414.

object_attribute, page 416.

type_attribute, page 424.

deprecated

Syntax

```
#pragma deprecated=entity
```

Description

If you place this pragma directive immediately before the declaration of a type, variable, function, field, or constant, any use of that type, variable, function, field, or constant will result in a warning.

The `deprecated` pragma directive has the same effect as the C++ attribute `[[deprecated]]`, but is available in C as well.

Example

```
#pragma deprecated
typedef int * intp_t;  // typedef intp_t is deprecated

#pragma deprecated
extern int fun(void);  // function fun is deprecated

#pragma deprecated
struct xx {            // struct xx is deprecated
  int x;
};

struct yy {
#pragma deprecated
  int y;               // field y is deprecated
};

intp_t fun(void)       // Warning here
{
  struct xx ax;        // Warning here
  struct yy ay;
  fun();               // Warning here
  return ay.y;         // Warning here
}
```

See also

Annex K (*Bounds-checking interfaces*) of the C standard.

diag_default

Syntax

```
#pragma diag_default=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 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. This level remains in effect until changed by another diagnostic-level pragma directive.

See also

Diagnostics, page 268.

diag_error

Syntax

```
#pragma diag_error=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 `error` for the specified diagnostics. This level remains in effect until changed by another diagnostic-level pragma directive.

See also

Diagnostics, page 268.

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. This level remains in effect until changed by another diagnostic-level pragma directive.

See also

Diagnostics, page 268.

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. This level remains in effect until changed by another diagnostic-level pragma directive.

See also *Diagnostics*, page 268.

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. This level remains in effect until changed by another diagnostic-level pragma directive.

See also *Diagnostics*, page 268.

error

Syntax `#pragma error message`

Parameters

message A string that represents the error message.

Description Use this pragma directive to cause an error message when it is parsed. This mechanism is different from the preprocessor directive `#error`, because the `#pragma error` directive can be included in a preprocessor macro using the `_Pragma` form of the directive and only causes an error if the macro is used.

Example

```
#if FOO_AVAILABLE
#define FOO ...
#else
#define FOO _Pragma("error\"Foo is not available\"")
#endif
```

If `FOO_AVAILABLE` is zero, an error will be signaled if the `FOO` macro is used in actual source code.

function_category

Syntax

```
#pragma function_category=category[, category...]
```

Parameters

category	A string that represents the name of a function category.

Description

Use this pragma directive to specify one or more function categories that the immediately following function belongs to. When used together with `#pragma calls`, the `function_category` directive specifies the destination for indirect calls for stack usage analysis purposes.

Example

```
#pragma function_category="Cat1"
```

See also

calls, page 405 and *Stack usage analysis*, page 105.

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 263.

inline

Syntax

```
#pragma inline[=forced|=never|=no_body|=forced_no_body]
```

Parameters

No parameter	Has the same effect as the `inline` keyword.
`forced`	Disables the compiler's heuristics and forces inlining.
`never`	Disables the compiler's heuristics and makes sure that the function will not be inlined.
`no_body`	Has the same effect as the `inline` keyword, but the generated function will not have a body.
`forced_no_body`	Disables the compiler's heuristics and forces inlining. The generated function will not have a body.

Description

Use `#pragma inline` to advise the compiler that the function defined immediately after the directive should be inlined according to C++ inline semantics.

Specifying `#pragma inline=forced` or `#pragma inline=forced_no_body` will always inline the defined function. If the compiler fails to inline the function for some reason, for example due to recursion, a warning message is emitted.

Inlining is normally performed only on the High optimization level. Specifying `#pragma inline=forced` or `#pragma inline=forced_no_body` will inline the function on all optimization levels or result in an error due to recursion, etc.

See also

Inlining functions, page 90.

language

Syntax

```
#pragma language={extended|default|save|restore}
```

Parameters

`extended`	Enables the IAR Systems language extensions from the first use of the pragma directive and onward.
`default`	From the first use of the pragma directive and onward, restores the settings for the IAR Systems language extensions to whatever that was specified by compiler options.
`save|restore`	Saves and restores, respectively, the IAR Systems language extensions setting around a piece of source code. Each use of `save` must be followed by a matching `restore` in the same file without any intervening `#include` directive.

Description

Use this pragma directive to control the use of language extensions.

Example

At the top of a file that needs to be compiled with IAR Systems extensions enabled:

```
#pragma language=extended
/* The rest of the file. */
```

Around a particular part of the source code that needs to be compiled with IAR Systems extensions enabled, but where the state before the sequence cannot be assumed to be the same as that specified by the compiler options in use:

```
#pragma language=save
#pragma language=extended
/* Part of source code. */
#pragma language=restore
```

See also

-*e*, page 293 and --*strict*, page 321.

location

Syntax

```
#pragma location={address|register|NAME}
```

Parameters

`address`	The absolute address of the global or static variable or function for which you want an absolute location.

register	An identifier that corresponds to one of the Arm core registers `R4–R11`. This parameter cannot be used **in 64-bit mode**.
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 variables must be declared `__no_init`.
- An identifier specifying a register. The variable defined after the pragma directive is placed in the register. The variable must be declared as `__no_init` and have file scope.

A string specifying a section for placing either a variable or function whose declaration follows the pragma directive. Do not place variables that would normally be in different sections—for example, variables declared as `__no_init` and variables declared as `const`—in the same named section.

Example

```
#pragma location=0xFFFF0400
__no_init volatile char PORT1; /* PORT1 is located at address
                                  0xFFFF0400 */

#pragma location=R8
__no_init int TASK; /* TASK is placed in R8 */

#pragma location="FLASH"
char PORT2; /* PORT2 is located in section FLASH */

/* 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 242 and *Declare and place your own sections*, page 116.

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
```

no_stack_protect

Syntax

```
#pragma no_stack_protect
```

Description

Use this pragma directive to disable stack protection for the defined function that follows.

This pragma directive only has effect if the compiler option `--stack_protection` has been used.

See also

Stack protection, page 92.

object_attribute

Syntax

```
#pragma object_attribute=object_attribute[ object_attribute...]
```

Parameters

For information about object attributes that can be used with this pragma directive, see *Object attributes*, page 385.

Description

Use this pragma directive to add one or more IAR-specific object attributes to the declaration or definition of a variable or function. Object attributes affect the actual variable or function and not its type. When you define a variable or function, the union of the object attributes from all declarations including the definition, is used.

Example

```
#pragma object_attribute=__no_init
char bar;
```

is equivalent to:

```
__no_init char bar;
```

See also *General syntax rules for extended keywords*, page 383.

optimize

Syntax

```
#pragma optimize=[goal][level][vectorize][disable]
```

Parameters

goal	Choose between: `size`, optimizes for size `balanced`, optimizes balanced between speed and size `speed`, optimizes for speed. `no_size_constraints`, optimizes for speed, but relaxes the normal restrictions for code size expansion.
level	Specifies the level of optimization—choose between `none`, `low`, `medium`, or `high`.
`vectorize`	Enables generation of NEON vector instructions.
disable	Disables one or several optimizations (separated by spaces). Choose from: `no_code_motion`, disables code motion `no_cse`, disables common subexpression elimination `no_inline`, disables function inlining `no_relaxed_fp`, disables the language relaxation that optimizes floating-point expressions more aggressively `no_tbaa`, disables type-based alias analysis `no_scheduling`, disables instruction scheduling `no_vectorize`, disables generation of NEON vector instructions `no_unroll`, disables loop unrolling

Description

Use this pragma directive to decrease the optimization level, or to turn off some specific optimizations. This pragma directive only affects the function that follows immediately after the directive.

The parameters `size`, `balanced`, `speed`, and `no_size_constraints` 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 SmallAndUsedOften()
{
  /* Do something here. */
}

#pragma optimize=size
int BigAndSeldomUsed()
{
  /* Do something here. */
}
```

See also

Fine-tuning enabled transformations, page 249.

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 the end of the compilation unit.

Note: This can result in significantly larger and slower code when accessing members of the structure.

Use either `__packed` or `#pragma pack` to relax the alignment restrictions for a type and the objects defined using that type. Mixing `__packed` and `#pragma pack` might lead to unexpected behavior.

See also

Structure types, page 378 and *__packed*, page 394.

__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.

You cannot use this pragma directive on functions that are members of an overload set with more than one member.

Example

```
#pragma __printf_args
int printf(char const *,...);

void PrintNumbers(unsigned short x)
{
  printf("%d", x);  /* Compiler checks that x is an integer */
}
```

public_equ

Syntax

```
#pragma public_equ="symbol",value
```

Parameters

symbol	The name of the assembler symbol to be defined (string).
value	The value of the defined assembler symbol (integer constant expression).

Description — Use this pragma directive to define a public assembler label and give it a value.

Example

```
#pragma public_equ="MY_SYMBOL",0x123456
```

See also — *--public_equ*, page 315.

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.

__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.

You cannot use this pragma directive on functions that are members of an overload set with more than one member.

Example

```
#pragma __scanf_args
int scanf(char const *,...);

int GetNumber()
{
  int nr;
  scanf("%d", &nr);  /* Compiler checks that
                        the argument is a
                        pointer to an integer */

  return nr;
}
```

section

Syntax

```
#pragma section="NAME"
```

alias

```
#pragma segment="NAME"
```

Parameters

NAME	The name of the section.

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 alignment.

Note: To place variables or functions in a specific section, use the `#pragma location` directive or the `@` operator.

Example

```
#pragma section="MYSECTION"
```

See also

Dedicated section operators, page 202, and the chapter *Linking your application*.

section_prefix

Syntax

```
#pragma section_prefix="prefix"
```

Parameters

prefix	A prefix to add to all section names.

Description

This pragma directive has the same effect as the compiler option `--section_prefix`. The names of all sections in the translation unit that are not explicitly named using the `@` notation or the `#pragma location` directive are changed.

See also

--*section_prefix*, page 319.

stack_protect

Syntax

```
#pragma stack_protect
```

Description

Use this pragma directive to force stack protection for the defined function that follows.

See also

Stack protection, page 92.

STDC CX_LIMITED_RANGE

Syntax `#pragma STDC CX_LIMITED_RANGE {ON|OFF|DEFAULT}`

Parameters

`ON`	Normal complex mathematic formulas can be used.
`OFF`	Normal complex mathematic formulas cannot be used.
`DEFAULT`	Sets the default behavior, that is `OFF`.

Description

Use this pragma directive to specify that the compiler can use the normal complex mathematic formulas for `*` (multiplication), `/` (division), and `abs`.

Note: This directive is required by Standard C. The directive is recognized but has no effect in the compiler.

STDC FENV_ACCESS

Syntax `#pragma STDC FENV_ACCESS {ON|OFF|DEFAULT}`

Parameters

`ON`	Source code accesses the floating-point environment. **Note:** This argument is not supported by the compiler.
`OFF`	Source code does not access the floating-point environment.
`DEFAULT`	Sets the default behavior, that is `OFF`.

Description

Use this pragma directive to specify whether your source code accesses the floating-point environment or not.

Note: This directive is required by Standard C.

STDC FP_CONTRACT

Syntax `#pragma STDC FP_CONTRACT {ON|OFF|DEFAULT}`

Parameters

`ON`	The compiler is allowed to contract floating-point expressions.
`OFF`	The compiler is not allowed to contract floating-point expressions.

`DEFAULT`	Sets the default behavior, that is `ON`. To change the default behavior, use the option `--no_default_fp_contract`.

Description
Use this pragma directive to specify whether the compiler is allowed to contract floating-point expressions or not. This directive is required by Standard C.

Example
```
#pragma STDC FP_CONTRACT ON
```

See also
--no_default_fp_contract, page 303

svc_number

Syntax
`#pragma svc_number=`*`number`*

Parameters

`number`	The software call number

Description
Use this pragma directive together with the `__svc` extended keyword. It is used as an argument to the generated `SVC` assembler instruction, and is used for selecting one software interrupt function in a system containing several such functions.

Example
```
#pragma svc_number=17
```

See also
Software interrupts, page 85 and *Exception functions for 64-bit mode*, page 86.

type_attribute

Syntax
`#pragma type_attribute=`*`type_attr`*`[` *`type_attr...`*`]`

Parameters
For information about type attributes that can be used with this pragma directive, see *Type attributes*, page 383.

Description
Use this pragma directive to specify IAR-specific *type attribute*s, which are not part of Standard C. 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 `myFunc`:

```
#pragma type_attribute=__thumb
void myFunc(void)
{
}
```

This declaration, which uses extended keywords, is equivalent:

```
__thumb void myFunc(void)
{
}
```

See also

The chapter *Extended keywords*.

unroll

Syntax

`#pragma unroll=n`

Parameters

n	The number of loop bodies in the unrolled loop, a constant integer. `#pragma unroll = 1` will prevent the unrolling of a loop.

Description

Use this pragma directive to specify that the loop following immediately after the directive should be unrolled and that the unrolled loop should have *n* copies of the loop body. The pragma directive can only be placed immediately before a `for`, `do`, or `while` loop, whose number of iterations can be determined at compile time.

Normally, unrolling is most effective for relatively small loops. However, in some cases, unrolling larger loops can be beneficial if it exposes opportunities for further optimizations between the unrolled loop iterations, for example, common subexpression elimination or dead code elimination.

The `#pragma unroll` directive can be used to force a loop to be unrolled if the unrolling heuristics are not aggressive enough. The pragma directive can also be used to reduce the aggressiveness of the unrolling heuristics.

Example

```
#pragma unroll=4
for (i = 0; i < 64; ++i)
{
  foo(i * k, (i + 1) * k);
}
```

See also

Loop unrolling, page 250

vectorize

Syntax

```
#pragma vectorize [= never]
```

Parameters

No parameter	Enables generation of NEON vector instructions **in 32-bit mode**.
`never`	Disables generation of NEON vector instructions.

Description

Use this pragma directive to enable or disable generation of NEON vector instructions for the loop that follows immediately after the pragma directive. This pragma directive can only be placed immediately before a `for`, `do`, or `while` loop. If the optimization level is lower than High, the pragma directive has no effect.

Note: Auto-vectorization is not supported **in 64-bit mode**.

Example

```
#pragma vectorize
for (i = 0; i < 1024; ++i)
{
  a[i] = b[i] * c[i];]
}
```

See also

Vectorization, page 252.

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 399.

Intrinsic functions

- Summary of intrinsic functions
- Descriptions of IAR Systems intrinsic functions

Summary of intrinsic functions

The IAR C/C++ Compiler for Arm can be used with several different sets of intrinsic functions.

To use the IAR Systems intrinsic functions in an application, include the header file `intrinsics.h`.

To use the ACLE (*Arm C Language Extensions*) intrinsic functions in an application, include the header file `arm_acle.h`. For more information, see *Intrinsic functions for ACLE*, page 429.

To use the Neon intrinsic functions in an application, include the header file `arm_neon.h`. For more information, see *Intrinsic functions for Neon instructions*, page 430.

To use the MVE intrinsic functions in an application, include the header file `arm_mve.h`. For more information, see *Intrinsic functions for MVE instructions*, page 430.

To use the CMSIS intrinsic functions in an application, include the main CMSIS header file for your device or core. Note that the CMSIS header files should not be included in the same module as `intrinsics.h`. For information about CMSIS, see *CMSIS integration (32-bit mode)*, page 233.

Note: The intrinsic function names start with double underscores, for example:

```
__disable_interrupt
```

INTRINSIC FUNCTIONS FOR ACLE

ACLE (*Arm C Language Extensions*) specifies a number of intrinsic functions. These are not documented here. Instead, see the *Arm C Language Extensions (IHI 0053D)*.

To use the intrinsic functions for ACLE in an application, include the header file `arm_acle.h`.

INTRINSIC FUNCTIONS FOR NEON INSTRUCTIONS

The Neon co-processor implements the Advanced SIMD instruction set extension, as defined by the Arm architecture. To use Neon intrinsic functions in an application, include the header file `arm_neon.h`. The functions use vector types that are named according to this pattern:

`<type><size>x<number_of_lanes>_t`

where:

- *type* is `int`, `unsigned int`, `float`, or `poly`
- *size* is `8`, `16`, `32`, or `64`
- *number_of_lanes* is `1`, `2`, `4`, `8`, or `16`.

The total bit width of a vector type is *size* times *number_of_lanes*, and should fit in a D register (64 bits) or a Q register (128 bits).

For example:

```
__intrinsic float32x2_t vsub_f32(float32x2_t, float32x2_t);
```

The intrinsic function `vsub_f32` inserts a `VSUB.F32` instruction that operates on two 64-bit vectors (D registers), each with two elements (lanes) of 32-bit floating-point type.

Some functions use an array of vector types. As an example, the definition of an array type with four elements of type `float32x2_t` is:

```
typedef struct
{
  float32x2_t val[4];
}
float32x2x4_t;
```

INTRINSIC FUNCTIONS FOR MVE INSTRUCTIONS

The M-profile vector extension (MVE) is defined in the Armv8.1-M architecture. To use Arm MVE intrinsic functions in an application, include the header file `arm_mve.h`. For more information on the MVE intrinsics, see *Arm C Language Extensions* (document number: 101028). The functions use vector types that are named according to this pattern:

`<type><size>x<number_of_lanes>_t`

where:

- *type* is `int`, `unsigned int`, or `float`
- *size* is `8`, `16`, `32`, or `64`
- *number_of_lanes* is `1`, `2`, `4`, `8`, or `16`.

The total bit width of a vector type is *size* times *number_of_lanes*, and should fit in a Q register (128 bits).

For example:

```
__intrinsic float32x4_t vsubq_f32(float32x4_t, float32x4_t);
```

The intrinsic function `vsub_f32` inserts a `VSUB.F32` instruction that operates on two 128-bit vectors (Q registers), each with four elements (lanes) of 32-bit floating-point type.

Some functions use an array of vector types. As an example, the definition of an array type with four elements of type `float32x2_t` is:

```
typedef struct
{
  float32x2_t val[4];
}
float32x2x4_t;
```

Descriptions of IAR Systems intrinsic functions

This section gives reference information about each IAR Systems intrinsic function.

__arm_cdp, __arm_cdp2

Syntax

```
void __arm_cdp(__cpid coproc, __cpopcw opc1, __cpreg CRd,
__cpreg CRn, __cpreg CRm, __cpopc opc2);
void __arm_cdp2(__cpid , __cpopw coprocopc1, __cpreg CRd,
__cpreg CRn, __cpreg CRm, __cpopc opc2);
```

Parameters

coproc	The coprocessor number `0..15`.
opc1, *opc2*	Coprocessor-specific operation codes.
CRd, *CRn*, *CRm*	Coprocessor registers.

Description

Inserts the coprocessor-specific data operation instruction `CDP` or `CDP2`. The parameters will be encoded in the instruction and must therefore be constants.

These intrinsic functions are defined according to the *Arm C Language Extensions* (ACLE).

Note: These intrinsic functions cannot be used **in 64-bit mode**.

See also *__CDP, __CDP2*, page 436.

__arm_ldc, __arm_ldcl, __arm_ldc2, __arm_ldc12

Syntax

```
void __arm_ldc(__cpid coproc, __cpreg CRd, const void* p);
void __arm_ldcl(__cpid coproc, __cpreg CRd, const void* p);
void __arm_ldc2(__cpid coproc, __cpreg CRd, const void* p);
void __arm_ldcl2(__cpid coproc, __cpreg CRd, const void* p);
```

Parameters

coproc	The coprocessor number `0..15`.
CRd	A coprocessor register.
p	Pointer to memory that the coprocessor will read from.

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 *CRd* will be encoded in the instruction and must therefore be constants.

These intrinsic functions are defined according to the *Arm C Language Extensions* (ACLE).

Note: These intrinsic functions cannot be used **in 64-bit mode**.

See also *__LDC, __LDCL, __LDC2, __LDC2L*, page 445.

__arm_mcr, __arm_mcr2, __arm_mcrr, __arm_mcrr2

Syntax

```
void __arm_mcr(__cpid coproc, __cpopc opc1, __ul src, __cpreg
CRn, __cpreg CRm, __cpopc opc2);
void __arm_mcr2(__cpid coproc, __cpopc opc1, __ul src, __cpreg
CRn, __cpreg CRm, __cpopc opc2);
void __arm_mcrr(__cpid coproc, __cpopc opc1, unsigned long long
src, __cpreg CRm);
void __arm_mcrr2(__cpid coproc, __cpopc opc1, unsigned long long
src, __cpreg CRm);
```

Parameters

coproc	The coprocessor number `0..15`.
opc1, *opc2*	Coprocessor-specific operation code.
src	The value to be written to the coprocessor.

CRn, *CRm* — The coprocessor register to read from.

Description

Inserts a coprocessor write instruction, `MCR`, `MCR2`, `MCRR`, or `MCRR2`. The parameters *coproc*, *opc1*, *opc2*, *CRn*, and *CRm* will be encoded in the instruction and must therefore be constants.

These intrinsic functions are defined according to the *Arm C Language Extensions* (ACLE).

Note: These intrinsic functions cannot be used **in 64-bit mode**.

See also

__MCR, __MCR2, page 446 and *__MCRR, __MCRR2*, page 447.

__arm_mrc, __arm_mrc2, __arm_mrrc, __arm_mrrc2

Syntax

```
unsigned int __arm_mrc(__cpid coproc, __cpopc opc1, __cpreg CRn,
__cpreg CRm, __cpopc opc2);
unsigned int __arm_mrc2(__cpid coproc, __cpopc opc1, __cpreg
CRn, __cpreg CRm, __cpopc opc2);
unsigned long long __arm_mrrc(__cpid coproc, __cpopc opc1,
__cpreg CRm);
unsigned long long __arm_mrrc2(__cpid coproc, __cpopc opc1,
__cpreg CRm);
```

Parameters

coproc	The coprocessor number `0..15`.
opc1, *opc2*	Coprocessor-specific operation code.
CRn, *CRm*	The coprocessor register to read from.

Description

Inserts a coprocessor read instruction, `MRC`, `MRC2`, `MRRC`, or `MRRC2`. Returns the value of the specified coprocessor register. The parameters *coproc*, *opc1*, *opc2*, *CRn*, and *CRm* will be encoded in the instruction and must therefore be constants.

These intrinsic functions are defined according to the *Arm C Language Extensions* (ACLE).

Note: These intrinsic functions cannot be used **in 64-bit mode**.

See also

__MCR, __MCR2, page 446, and *__MRRC, __MRRC2*, page 448.

__arm_rsr, __arm_rsr64, __arm_rsrp

Syntax

```
unsigned int __arm_rsr(sys_reg special_register);
unsigned long long __arm_rsr64(__sys_reg special_register);
void * __arm_rsrp(sys_reg special_register);
```

Parameters

special_register	A string literal specifying a register.

Description

Reads a system register. Use a string literal to specify which register to read. For `__arm_rsr` and `__arm_rsrp`, the string literal can specify the name of a system register accepted in an `MRS` or `VMRS` instruction for the architecture specified by the compiler option `--cpu`.

For `__arm_rsr` and `__arm_rsrp`, the string literal can also specify a 32-bit coprocessor register, using this format:

`coprocessor : opc1 :c CRn :c CRm : opc2`

For `__arm_rsr64`, the string literal can specify a 64-bit coprocessor register using this format:

`coprocessor : opc1 :c CRm`

where, for both formats

- *coprocessor* is a number, `c0..c15` or `cp0..cp15`
- *opc1* and *opc2* are coprocessor-specific operation codes, `0..7`
- *CRn* and *CRm* are coprocessor registers `0..15`

These intrinsic functions are defined according to the *Arm C Language Extensions* (ACLE).

__arm_stc, __arm_stcl, __arm_stc2, __arm_stc2l

Syntax

```
void __arm_stc(__cpid coproc, __cpreg CRd, const void* p);
void __arm_stcl(__cpid coproc, __cpreg CRd, const void* p);
void __arm_stc2(__cpid coproc, __cpreg CRd, const void* p);
void __arm_stc2l(__cpid coproc, __cpreg CRd, const void* p);
```

Parameters

coproc	The coprocessor number `0..15`.
CRd	A coprocessor register.
p	Pointer to memory that the coprocessor will write to.

Description

Inserts the coprocessor store instruction `STC` (or one of its variants). The parameters *`coproc`*, *`CRd`*, and *`p`* will be encoded in the instruction and must therefore be constants.

These intrinsic functions are defined according to the *Arm C Language Extensions* (ACLE).

Note: These intrinsic functions cannot be used **in 64-bit mode**.

See also

__STC, __STCL, __STC2, __STC2L, page 462.

__arm_wsr, __arm_wsr64, __arm_wsrp

Syntax

```
void __arm_wsr(const char * special_reg, _uint32_t value);
void __arm_wsr64(const char * special_reg, uint64_t value);
void __arm_wsrp(const char * special_reg, const void * value);
```

Parameters

`special_reg`	A string literal specifying a system register.
`value`	The value to write to the system register.

Description

Writes to a system register. Use a string literal to specify which register to write to. For `__arm_wsr` and `__arm_wsrp`, the string literal can specify the name of a system register accepted in an `MSR` or `VMSR` instruction for the architecture specified by the compiler option `--cpu`.

For `__arm_wsr` and `__arm_wsrp`, the string literal can also specify a 32-bit coprocessor register, using this format:

`coprocessor` `:` *`opc1`* `:c` *`CRn`* `:c` *`CRm`* `:` *`opc2`*

For `__arm_wsr64`, the string literal can specify a 64-bit coprocessor register using this format:

`coprocessor` `:` *`opc1`* `:c` *`CRm`*

where, for both formats

- *`coprocessor`* is the coprocessor number, `cp0..cp15` or `p0..p15`
- *`opc1`* and *`opc2`* are coprocessor-specific operation codes, `0..7`
- *`CRn`* and *`CRm`* are coprocessor registers, `0..15`

These intrinsic functions are defined according to ACLE (*Arm C Language Extensions)*.

__CDP, __CDP2

Syntax

```
void __CDP(__cpid coproc, __cpopcw opc1, __cpreg CRd, __cpreg
CRn, __cpreg CRm, __cpopc opc2);
void __CDP2(__cpid coproc, __cpopcw opc1, __cpreg CRd, __cpreg
CRn, __cpreg CRm, __cpopc opc2);
```

Parameters

coproc	The coprocessor number `0..15`.
opc1, *opc2*	Coprocessor-specific operation codes.
CRd, *CRn*, *CRm*	Coprocessor registers.

Description

Inserts the coprocessor-specific data operation instruction `CDP` or `CDP2`.

The parameters will be encoded in the instruction and must therefore be constants.

The intrinsic functions `__CDP` and `__CDP2` require an Armv5 architecture or higher for Arm mode, or Armv6 or higher for Thumb mode.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

See also

__arm_cdp, __arm_cdp2, page 431.

__CLREX

Syntax

```
void __CLREX(void);
```

Description

Inserts a `CLREX` instruction.

This intrinsic function requires architecture Armv6K or Armv7 for Arm mode, and AVRv7 for Thumb mode.

Note: This intrinsic function cannot be used **in 64-bit mode**.

__CLZ

Syntax

```
unsigned int __CLZ(unsigned int);
```

Description

Inserts a `CLZ` instruction. If the `CLZ` instruction is not available, a separate sequence of instructions is inserted to achieve the same result.

Note: This intrinsic function cannot be used **in 64-bit mode**.

See also

The *Arm C Language Extensions* (ACLE) intrinsic functions `__clz`, `__clzl`, and `__clzll`.

__crc32b, __crc32h, __crc32w, __crc32d

Syntax

```
unsigned int __crc32b(unsigned int crc, unsigned char data);
unsigned int __crc32h(unsigned int crc, unsigned short data);
unsigned int __crc32w(unsigned int crc, unsigned int data);
unsigned int __crc32d(unsigned int crc, unsigned long long
data);
```

Description

Calculates a CRC32 checksum from a checksum (or initial value) `crc` and one item of `data`.

Note: The 32-bit Arm/Thumb instructions do not include `CRC32X`, so `__crc32d` is implemented as two calls to `__crc32w`.

These intrinsic functions are defined according to the *Arm C Language Extensions* (ACLE).

__crc32cb, __crc32ch, __crc32cw, __crc32cd

Syntax

```
unsigned int __crc32cb(unsigned int crc, unsigned char data);
unsigned int __crc32ch(unsigned int crc, unsigned short data);
unsigned int __crc32cw(unsigned int crc, unsigned int data);
unsigned int __crc32cd(unsigned int crc, unsigned long long
data);
```

Description

Calculates a CRC32C checksum from a checksum (or initial value) `crc` and one item of `data`.

Note: The 32-bit Arm/Thumb instructions do not include `CRC32CX`, so `__crc32cd` is implemented as two calls to `__crc32cw`.

These intrinsic functions are defined according to the *Arm C Language Extensions* (ACLE).

__disable_debug

Syntax

```
void __disable_debug(void);
```

Description

Disables debug requests in the `DAIF` system register (bit 4).

Note: This intrinsic function cannot be used **in 32-bit mode**.

__disable_fiq

Syntax

```
void __disable_fiq(void);
```

Description

In 32-bit mode: Disables fast interrupt requests (fiq). This intrinsic function can only be used in privileged mode and is not available for Cortex-M devices.

In 64-bit mode: Disables fast interrupt requests in the `DAIF` system register (bit 1).

__disable_interrupt

Syntax

```
void __disable_interrupt(void);
```

Description

Disables interrupts.

In 32-bit mode: 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.

In 64-bit mode: Disables all four types of interrupts in the `DAIF` system register (low 4 bits).

__disable_irq

Syntax

```
void __disable_irq(void);
```

Description

In 32-bit mode: Disables interrupt requests (irq). This intrinsic function can only be used in privileged mode and is not available for Cortex-M devices.

In 64-bit mode: Disables interrupt requests in the `DAIF` system register (bit 2).

__disable_SError

Syntax

```
void __disable_SError(void);
```

Description

Disables synchronous error requests in the `DAIF` system register (bit 3).

Note: This intrinsic function cannot be used **in 32-bit mode**.

__DMB

Syntax `void __DMB(void);`

Description Inserts a `DMB` instruction. This intrinsic function requires an Armv6M architecture, or an Armv7 architecture or higher.

Note: This intrinsic function cannot be used **in 64-bit mode**.

See also The *Arm C Language Extensions* (ACLE) intrinsic function `__dmb`.

__DSB

Syntax `void __DSB(void);`

Description Inserts a `DSB` instruction. This intrinsic function requires an Armv6M architecture, or an Armv7 architecture or higher.

Note: This intrinsic function cannot be used **in 64-bit mode**.

See also The *Arm C Language Extensions* (ACLE) intrinsic function `__dsb`.

__enable_debug

Syntax `void __enable_debug(void);`

Description Enables debug requests in the `DAIF` system register (bit 4).

Note: This intrinsic function cannot be used **in 32-bit mode**.

__enable_fiq

Syntax `void __enable_fiq(void);`

Description **In 32-bit mode:** Enables fast interrupt requests (fiq). This intrinsic function can only be used in privileged mode, and it is not available for Cortex-M devices.

In 64-bit mode: Enables fast interrupt requests in the `DAIF` system register (bit 1).

__enable_interrupt

Syntax

```
void __enable_interrupt(void);
```

Description

Enables interrupts.

In 32-bit mode: 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.

In 64-bit mode: Enables all four types of interrupts in the `DAIF` system register (low 4 bits).

__enable_irq

Syntax

```
void __enable_irq(void);
```

Description

In 32-bit mode: Enables interrupt requests (irq). This intrinsic function can only be used in privileged mode, and it is not available for Cortex-M devices.

In 64-bit mode: Enables interrupt requests in the `DAIF` system register (bit 2).

__enable_SError

Syntax

```
void __enable_SError(void);
```

Description

Enables synchronous error requests in the `DAIF` system register (bit 3).

Note: This intrinsic function cannot be used **in 32-bit mode**.

__fma, __fmaf

Syntax

```
double __fma(double x, double y, double z);
float __fmaf(float x, float y, float z);
```

Description

Fused floating-point multiply-accumulate computes `x*y+z` without intermediate rounding, which corresponds either to the intrinsic call `__VFMA_F64(z, x, y)` for double precision, or `__VFMA_F32(z, x, y)` for single precision.

These intrinsic functions are defined according to the *Arm C Language Extensions* (ACLE).

__get_BASEPRI

Syntax `unsigned int __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, Cortex-M4, or Cortex-M7 device.

Note: This intrinsic function cannot be used **in 64-bit mode**.

See also *__arm_rsr, __arm_rsr64, __arm_rsrp*, page 434.

__get_CONTROL

Syntax `unsigned int __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.

Note: This intrinsic function cannot be used **in 64-bit mode**.

See also *__arm_rsr, __arm_rsr64, __arm_rsrp*, page 434.

__get_CPSR

Syntax `unsigned int __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.

Note: This intrinsic function cannot be used **in 64-bit mode**.

See also *__arm_rsr, __arm_rsr64, __arm_rsrp*, page 434.

__get_FAULTMASK

Syntax `unsigned int __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, Cortex-M4, or Cortex-M7 device.

Note: This intrinsic function cannot be used **in 64-bit mode**.

See also *__arm_rsr, __arm_rsr64, __arm_rsrp*, page 434.

__get_FPSCR

Syntax `unsigned int __get_FPSCR(void);`

Description Returns the value of `FPSCR` (floating-point status and control register).

This intrinsic function is only available for devices with a VFP coprocessor.

Note: This intrinsic function cannot be used **in 64-bit mode**.

See also *__arm_rsr, __arm_rsr64, __arm_rsrp*, page 434.

__get_interrupt_state

Syntax `__istate_t __get_interrupt_state(void);`

Description **In 32-bit mode:**

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.

In 64-bit mode:

Returns the 4 low bits of the `DAIF` system register (`__istate_t` is unsigned `long long`).

Example

```
#include "intrinsics.h"

void CriticalFn()
{
  __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_IPSR

Syntax `unsigned int __get_IPSR(void);`

Description Returns the value of the `IPSR` register (Interrupt Program Status Register). This intrinsic function can only be used in privileged mode, and is only available for Cortex-M devices.

Note: This intrinsic function cannot be used **in 64-bit mode**.

See also *__arm_rsr, __arm_rsr64, __arm_rsrp*, page 434.

__get_LR

Syntax `unsigned int __get_LR(void);`

Description Returns the value of the link register (`R14`).

Note: This intrinsic function cannot be used **in 64-bit mode**.

__get_MSP

Syntax `unsigned int __get_MSP(void);`

Description Returns the value of the `MSP` register (Main Stack Pointer). This intrinsic function can only be used in privileged mode, and is only available for Cortex-M devices.

Note: This intrinsic function cannot be used **in 64-bit mode**.

See also *__arm_rsr, __arm_rsr64, __arm_rsrp*, page 434.

__get_PRIMASK

Syntax `unsigned int __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.

Note: This intrinsic function cannot be used **in 64-bit mode**.

See also *__arm_rsr, __arm_rsr64, __arm_rsrp*, page 434.

__get_PSP

Syntax
```
unsigned int __get_PSP(void);
```

Description

Returns the value of the `PSP` register (Process Stack Pointer). This intrinsic function can only be used in privileged mode, and is only available for Cortex-M devices.

Note: This intrinsic function cannot be used **in 64-bit mode**.

See also

__arm_rsr, __arm_rsr64, __arm_rsrp, page 434.

__get_PSR

Syntax
```
unsigned int __get_PSR(void);
```

Description

Returns the value of the `PSR` register (combined Program Status Register). This intrinsic function can only be used in privileged mode, and is only available for Cortex-M devices.

Note: This intrinsic function cannot be used **in 64-bit mode**.

See also

__arm_rsr, __arm_rsr64, __arm_rsrp, page 434.

__get_SB

Syntax
```
unsigned int __get_SB(void);
```

Description

Returns the value of the static base register (`R9`).

Note: This intrinsic function cannot be used **in 64-bit mode**.

__get_SP

Syntax
```
unsigned int __get_SP(void);
```

Description

Returns the value of the stack pointer register (`R13`).

Note: This intrinsic function cannot be used **in 64-bit mode**.

__ISB

Syntax

```
void __ISB(void);
```

Description

Inserts an `ISB` instruction. This intrinsic function requires an Armv6M architecture, or an Armv7 architecture or higher.

Note: This intrinsic function cannot be used **in 64-bit mode**.

See also

The *Arm C Language Extensions* (ACLE) intrinsic function `__isb`.

__LDC, __LDCL, __LDC2, __LDC2L

Syntax

`void __LDCxxx(__ul coproc, __ul CRn, __ul const *src);`

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.

The intrinsic functions `__LDC` and `__LDCL` require architecture Armv4 or higher for Arm mode, and Armv6T2 or higher for Thumb mode.

The intrinsic functions `__LDC2` and `__LDC2L` require architecture Armv5 or higher for Arm mode, and Armv6T2 or higher for Thumb mode.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

See also

__arm_ldc, __arm_ldcl, __arm_ldc2, __arm_ldc12, page 432.

__LDC_noidx, __LDCL_noidx, __LDC2_noidx, __LDC2L_noidx

Syntax

`void __LDCxxx_noidx(__ul coproc, __ul CRn, __ul const *src, __ul option);`

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.

The intrinsic functions `__LDC_noidx` and `__LDCL_noidx` require architecture Armv4 or higher for Arm mode, and Armv6T2 or higher for Thumb mode.

The intrinsic functions `__LDC2_noidx` and `__LDC2L_noidx` require architecture Armv5 or higher for Arm mode, and Armv6T2 or higher for Thumb mode.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

__LDREX, __LDREXB, __LDREXD, __LDREXH

Syntax

```
unsigned int __LDREX(unsigned int *);
unsigned char __LDREXB(unsigned char *);
unsigned long long __LDREXD(unsigned long long *);
unsigned short __LDREXH(unsigned short *);
```

Description

Inserts the specified instruction.

The `__LDREX` intrinsic function requires architecture Armv6 or higher for Arm mode, and Armv6T2 or baseline Armv8-M for Thumb mode.

The `__LDREXB` and the `__LDREXH` intrinsic functions require architecture Armv6K or Armv7 for Arm mode, and Armv7 or baseline Armv8-M for Thumb mode.

The `__LDREXD` intrinsic function requires architecture Armv6K or Armv7 for Arm mode, and Armv7 but not Armv7-M for Thumb mode.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

__MCR, __MCR2

Syntax

```
void __MCR(__ul coproc, __ul opcode_1, __ul src, __ul CRn, __ul
CRm, __ul opcode_2);
void __MCR2(__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` or `MCR2`). The parameters *coproc*, *opcode_1*, *CRn*, *CRm*, and *opcode_2* will be encoded in the instruction and must therefore be constants.

The intrinsic function `__MCR` requires either Arm mode, or an Armv6T2 or higher for Thumb mode.

The intrinsic function `__MCR2` requires an Armv5T architecture or higher for Arm mode, or Armv6T2 or higher for Thumb mode.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

See also

__arm_mcr, __arm_mcr2, __arm_mcrr, __arm_mcrr2, page 432.

__MCRR, __MCRR2

Syntax

```
void __MCRR(__cpid coproc, __cpopc opc1, unsigned long long src,
__cpreg CRm);
void __MCRR2(__cpid coproc, __cpopc opc1, unsigned long long
src, __cpreg CRm);
```

Parameters

coproc	The coprocessor number `0..15`.
opc1	Coprocessor-specific operation code.
src	The value to be written to the coprocessor.
CRm	The coprocessor register to read from.

Description

Inserts a coprocessor write instruction, `MCRR` or `MCRR2`. The parameters *coproc*, *opc1*, and *CRm* will be encoded in the instruction and must therefore be constants.

The intrinsic functions `__MCRR` and `__MCRR2` require an Armv6 architecture or higher for Arm mode, or Armv6T2 or higher for Thumb mode.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

See also

__arm_mcr, __arm_mcr2, __arm_mcrr, __arm_mcrr2, page 432.

__MRC, __MRC2

Syntax

```
unsigned int __MRC(__ul coproc, __ul opcode_1, __ul CRn, __ul
CRm, __ul opcode_2);
unsigned int __MRC2(__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` or `MRC2`). Returns the value of the specified coprocessor register. The parameters *coproc*, *opcode_1*, *CRn*, *CRm*, and *opcode_2* will be encoded in the instruction and must therefore be constants.

The intrinsic function `__MRC` requires either Arm mode, or an Armv6T2 or higher for Thumb mode.

The intrinsic function `__MRC2` requires an Armv5T architecture or higher for Arm mode, or Armv6T2 or higher for Thumb mode.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

See also

__arm_mrc, __arm_mrc2, __arm_mrrc, __arm_mrrc2, page 433.

__MRRC, __MRRC2

Syntax

```
unsigned long long __MRRC(__cpid coproc, __cpopc opc1, __cpreg
CRm);
unsigned long long __MRRC2(__cpid coproc, __cpopc opc1, __cpreg
CRm);
```

Parameters

coproc	The coprocessor number `0..15`.
opc1	Coprocessor-specific operation code.
CRm	The coprocessor register to read from.

Description

Inserts a coprocessor read instruction, `MRRC` or `MRRC2`. Returns the value of the specified coprocessor register. The parameters *coproc*, *opc1*, and *CRm* will be encoded in the instruction and must therefore be constants.

The intrinsic functions `__MRRC` and `__MRRC2` require an Armv6 architecture or higher for Arm mode, or ArmV6T2 or higher for Thumb mode.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

See also

__arm_mrc, __arm_mrc2, __arm_mrrc, __arm_mrrc2, page 433.

__no_operation

Syntax

```
void __no_operation(void);
```

Description

Inserts a `NOP` instruction.

This intrinsic function is equivalent to the *Arm C Language Extensions* (ACLE) intrinsic function `__nop`.

__PKHBT

Syntax

```
unsigned int __PKHBT(unsigned int x, unsigned int y, unsigned
int count);
```

Parameters

x	First operand.
y	Second operand, optionally shifted left.
count	Shift count `0–31`, where `0` means no shift.

Description

Inserts a `PKHBT` instruction, with an optionally shifted operand (LSL) for count in the range 1–31.

This intrinsic function requires an Arm v6 architecture or higher for Arm mode, and Armv7-A, Armv7-R, or Armv7E-M for Thumb mode.

Note: This intrinsic function cannot be used **in 64-bit mode**.

__PKHTB

Syntax

```
unsigned int __PKHTB(unsigned int x, unsigned int y, unsigned
int count);
```

Parameters

x	First operand.
y	Second operand, optionally shifted right (arithmetic shift).
count	Shift count `0–32`, where `0` means no shift.

Description

Inserts a `PKHTB` instruction, with an optionally shifted operand (ASR) for count in the range 1–32.

This intrinsic function requires an Arm v6 architecture or higher for Arm mode, and Armv7-A, Armv7-R, or Arm v7E-M for Thumb mode.

Note: This intrinsic function cannot be used **in 64-bit mode**.

__PLD, __PLDW

Syntax

```
void __PLD(void const *);
void __PLDW(void const *);
```

Description

Inserts a preload data instruction (`PLD` or `PLDW`).

The intrinsic function `__PLD` requires an Armv7 architecture. `__PLDW` requires an Armv7 architecture with MP extensions, for example, Cortex-A5.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

See also

The *Arm C Language Extensions* (ACLE) intrinsic functions `__pld`.

__PLI

Syntax

```
void __PLI(void const *);
```

Description

Inserts a `PLI` instruction. This intrinsic function requires an Arm v7 architecture.

Note: This intrinsic function cannot be used **in 64-bit mode**.

See also

The *Arm C Language Extensions* (ACLE) intrinsic function `__pli`.

__QADD, __QDADD, __QDSUB, __QSUB

Syntax

```
signed int __Qxxx(signed int, signed int);
```

Description

Inserts the specified instruction.

These intrinsic functions require architecture Armv5E or higher for Arm mode, and Armv7-A, Armv7-R, or Armv7E-M for Thumb mode.

These intrinsic functions are equivalent to the *Arm C Language Extensions* (ACLE) intrinsic functions `__qadd` and `__qsub`.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

__QADD8, __QADD16, __QASX, __QSAX, __QSUB8, __QSUB16

Syntax

```
unsigned int __Qxxx(unsigned int, unsigned int);
```

Description

Inserts the specified instruction.

These intrinsic functions require architecture Armv6 or higher for Arm mode, and Armv7-A, Armv7-R, or Armv7E-M for Thumb mode.

These intrinsic functions are equivalent to the *Arm C Language Extensions* (ACLE) intrinsic functions `__qadd8`, `__qadd16`, `__qasx`, `__qsax`, `__qsub8`, and `__qsub16`.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

__QCFlag

Syntax

```
unsigned int __QCFlag(void);
```

Description

Returns the value of the cumulative saturation flag `QC` of the `FPSCR` register (Floating-point Status and Control Register). This intrinsic function is only available for devices with Neon (Advanced SIMD).

Note: This intrinsic function cannot be used **in 64-bit mode**.

__QDOUBLE

Syntax `signed int __QDOUBLE(signed int);`

Description Inserts an instruction `QADD Rd,Rs,Rs` for a source register `Rs`, and a destination register `Rd`.

This intrinsic function requires architecture Armv5E or higher for Arm mode, and Armv7-A, Armv7-R, or Armv7E-M for Thumb mode.

Note: This intrinsic function cannot be used **in 64-bit mode**.

__QFlag

Syntax `int __QFlag(void);`

Description Returns the `Q` flag that indicates if overflow/saturation has occurred.

This intrinsic function requires architecture Armv5E or higher for Arm mode, and Armv7-A, Armv7-R, or Armv7E-M for Thumb mode.

Note: This intrinsic function cannot be used **in 64-bit mode**.

__RBIT

Syntax `unsigned int __RBIT(unsigned int);`

Description Inserts an `RBIT` instruction, which reverses the bit order in a 32-bit register. If the `RBIT` instruction is not available, a separate sequence of instructions is inserted to achieve the same result.

This intrinsic function is equivalent to the *Arm C Language Extensions* (ACLE) intrinsic function `__rbit`.

Note: This intrinsic function cannot be used **in 64-bit mode**.

__reset_Q_flag

Syntax `void __reset_Q_flag(void);`

Description Clears 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 v7A, Arm v7R, or Arm v7E-M for Thumb mode.

Note: This intrinsic function cannot be used **in 64-bit mode**.

__reset_QC_flag

Syntax

```
void __reset_QC_flag(void);
```

Description

Clears the value of the cumulative saturation flag `QC` of the `FPSCR` register (Floating-point Status and Control Register). This intrinsic function is only available for devices with Neon (Advanced SIMD).

Note: This intrinsic function cannot be used **in 64-bit mode**.

__REV, __REV16, __REVSH

Syntax

```
unsigned int __REV(unsigned int);
unsigned int __REV16(unsigned int);
signed int __REVSH(short);
```

Description

Inserts the specified instruction. If the instruction is not available, a separate sequence of instruction is inserted to achieve the same result.

These intrinsic functions are equivalent to the *Arm C Language Extensions* (ACLE) intrinsic functions `__rev`, `__rev16`, and `__revsh`.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

__rintn, __rintnf

Syntax

```
double __rintn(double x);
float __rintnf(float x);
```

Description

Rounds a number `x` to the nearest integer number (with ties to even), which corresponds either to the intrinsic call `__VRINTN_F64(x)` for double precision, or `__VRINTN_F32(x)` for single precision.

These intrinsic functions are defined according to the *Arm C Language Extensions* (ACLE).

__ROR

Syntax `unsigned int __ROR(unsigned int);`

Description Inserts an `ROR` instruction.

This intrinsic function is equivalent to the *Arm C Language Extensions* (ACLE) intrinsic function `__ror`.

Note: This intrinsic function cannot be used **in 64-bit mode**.

__RRX

Syntax `unsigned int __RRX(unsigned int);`

Description Inserts an `RRX` instruction.

Note: This intrinsic function cannot be used **in 64-bit mode**.

__SADD8, __SADD16, __SASX, __SSAX, __SSUB8, __SSUB16

Syntax `unsigned int __Sxxx(unsigned int, unsigned int);`

Description Inserts the specified instruction.

These intrinsic functions require architecture Armv6 or higher for Arm mode, and Armv7-A, Armv7-R, or Armv7E-M for Thumb mode.

These intrinsic functions are equivalent to the *Arm C Language Extensions* (ACLE) intrinsic functions `__sadd8`, `__sadd16`, `__sasx`, `__ssax`, `__ssub8`, and `__ssub16`.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

__SEL

Syntax `unsigned int __SEL(unsigned int, unsigned int);`

Description Inserts an `SEL` instruction.

This intrinsic function requires architecture Armv6 or higher for Arm mode, and Armv7-A, Armv7-R, or Armv7E-M for Thumb mode.

Note: This intrinsic function cannot be used **in 64-bit mode**.

__set_BASEPRI

Syntax

```
void __set_BASEPRI(unsigned int);
```

Description

Sets the value of the BASEPRI register. This intrinsic function can only be used in privileged mode and it requires a Cortex-M3, Cortex-M4, or Cortex-M7 device.

Note: This intrinsic function cannot be used **in 64-bit mode**.

See also

__arm_wsr, __arm_wsr64, __arm_wsrp, page 435.

__set_CONTROL

Syntax

```
void __set_CONTROL(unsigned int);
```

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.

Note: This intrinsic function cannot be used **in 64-bit mode**.

See also

__arm_wsr, __arm_wsr64, __arm_wsrp, page 435.

__set_CPSR

Syntax

```
void __set_CPSR(unsigned int);
```

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.

Note: This intrinsic function cannot be used **in 64-bit mode**.

See also

__arm_wsr, __arm_wsr64, __arm_wsrp, page 435.

__set_FAULTMASK

Syntax

```
void __set_FAULTMASK(unsigned int);
```

Description

Sets the value of the FAULTMASK register. This intrinsic function can only be used in privileged mode and it requires a Cortex-M3, Cortex-M4, or Cortex-M7 device.

Note: This intrinsic function cannot be used **in 64-bit mode**.

See also *__arm_wsr, __arm_wsr64, __arm_wsrp*, page 435.

__set_FPSCR

Syntax `void __set_FPSCR(unsigned int);`

Description Sets the value of `FPSCR` (floating-point status and control register)

This intrinsic function is only available for devices with a VFP coprocessor.

Note: This intrinsic function cannot be used **in 64-bit mode**.

See also *__arm_wsr, __arm_wsr64, __arm_wsrp*, page 435.

__set_interrupt_state

Syntax `void __set_interrupt_state(__istate_t);`

Description **In 32-bit mode:** Restores the interrupt state to a value previously returned by the `__get_interrupt_state` function.

In 64-bit mode: Sets the 4 low bits of the `DAIF` system register.

For information about the `__istate_t` type, see *__get_interrupt_state*, page 442.

__set_LR

Syntax `void __set_LR(unsigned int);`

Description Assigns a new address to the link register (`R14`).

Note: This intrinsic function cannot be used **in 64-bit mode**.

__set_MSP

Syntax `void __set_MSP(unsigned int);`

Description Sets the value of the `MSP` register (Main Stack Pointer). This intrinsic function can only be used in privileged mode, and is only available for Cortex-M devices.

Note: This intrinsic function cannot be used **in 64-bit mode**.

See also *__arm_wsr, __arm_wsr64, __arm_wsrp*, page 435.

__set_PRIMASK

Syntax
```
void __set_PRIMASK(unsigned int);
```

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.

Note: This intrinsic function cannot be used **in 64-bit mode**.

See also

__arm_wsr, __arm_wsr64, __arm_wsrp, page 435.

__set_PSP

Syntax
```
void __set_PSP(unsigned int);
```

Description

Sets the value of the PSP register (Process Stack Pointer). This intrinsic function can only be used in privileged mode, and is only available for Cortex-M devices.

Note: This intrinsic function cannot be used **in 64-bit mode**.

See also

__arm_wsr, __arm_wsr64, __arm_wsrp, page 435.

__set_SB

Syntax
```
void __set_SB(unsigned int);
```

Description

Assigns a new address to the static base register (`R9`).

Note: This intrinsic function cannot be used **in 64-bit mode**.

__set_SP

Syntax
```
void __set_SP(unsigned int);
```

Description

Assigns a new address to the stack pointer register (`R13`).

Note: This intrinsic function cannot be used **in 64-bit mode**.

__SEV

Syntax `void __SEV(void);`

Description Inserts an `SEV` instruction.

This intrinsic function requires architecture Armv7 for Arm mode, and Armv6-M or Armv7 for Thumb mode.

This intrinsic function is equivalent to the *Arm C Language Extensions* (ACLE) intrinsic function `__sev`.

Note: This intrinsic function cannot be used **in 64-bit mode**.

__SHADD8, __SHADD16, __SHASX, __SHSAX, __SHSUB8, __SHSUB16

Syntax `unsigned int __SHxxx(unsigned int, unsigned int);`

Description Inserts the specified instruction.

These intrinsic functions require architecture Armv6 or higher for Arm mode, and Armv7-A, Armv7-R, or Armv7E-M for Thumb mode.

These intrinsic functions are equivalent to the *Arm C Language Extensions* (ACLE) intrinsic functions `__shadd8`, `__shadd16`, `__shasx`, `__shsax`, `__shsub8`, and `__shsub16`.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

__SMLABB, __SMLABT, __SMLATB, __SMLATT, __SMLAWB, __SMLAWT

Syntax `unsigned int __SMLAxxx(unsigned int, unsigned int, unsigned int);`

Description Inserts the specified instruction.

These intrinsic functions require architecture Armv6 or higher for Arm mode, and Armv7-A, Armv7-R, or Armv7E-M for Thumb mode.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

__SMLAD, __SMLADX, __SMLSD, __SMLSDX

Syntax

```
unsigned int __SMLxxx(unsigned int, unsigned int, unsigned int);
```

Description

Inserts the specified instruction.

These intrinsic functions require architecture Armv6 or higher for Arm mode, and Armv7-A, Armv7-R, or Armv7E-M for Thumb mode.

These intrinsic functions are equivalent to the *Arm C Language Extensions* (ACLE) intrinsic functions `__smlad`, `__smladx`, `__smlsd`, and `__smlsdx`.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

__SMLALBB, __SMLALBT, __SMLALTB, __SMLALTT

Syntax

```
unsigned long long __SMLALxxx(unsigned int, unsigned int,
unsigned long long);
```

Description

Inserts the specified instruction.

These intrinsic functions require architecture Armv6 or higher for Arm mode, and Armv7-A, Armv7-R, or Armv7E-M for Thumb mode.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

__SMLALD, __SMLALDX, __SMLSLD, __SMLSLDX

Syntax

```
unsigned long long __SMLxxx(unsigned int, unsigned int, unsigned
long long);
```

Description

Inserts the specified instruction.

These intrinsic functions require architecture Armv6 or higher for Arm mode, and Armv7-A, Armv7-R, or Armv7E-M for Thumb mode.

These intrinsic functions are equivalent to the *Arm C Language Extensions* (ACLE) intrinsic functions `__smlald`, `__smlaldx`, `__smlsld`, and `__smlsldx`.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

__SMMLA, __SMMLAR, __SMMLS, __SMMLSR

Syntax

```
unsigned int __SMMLxxx(unsigned int, unsigned int, unsigned
int);
```

Description

Inserts the specified instruction.

These intrinsic functions require architecture Armv6 or higher for Arm mode, and Armv7-A, Armv7-R, or Armv7E-M for Thumb mode.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

__SMMUL, __SMMULR

Syntax

```
signed int __SMMULxxx(signed int, signed int);
```

Description

Inserts the specified instruction.

These intrinsic functions require architecture Armv6 or higher for Arm mode, and Armv7-A, Armv7-R, or Armv7E-M for Thumb mode.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

__SMUAD, __SMUADX, __SMUSD, __SMUSDX

Syntax

```
unsigned int __SMUxxx(unsigned int, unsigned int);
```

Description

Inserts the specified instruction.

These intrinsic functions require architecture Armv6 or higher for Arm mode, and Armv7-A, Armv7-R, or Armv7E-M for Thumb mode.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

__SMUL

Syntax

```
signed int __SMUL(signed short, signed short);
```

Description

Inserts a signed 16-bit multiplication.

This intrinsic function requires architecture Armv5-E or higher for Arm mode, and Armv7-A, Armv7-R, or Armv7E-M for Thumb mode.

Note: This intrinsic function cannot be used **in 64-bit mode**.

__SMULBB, __SMULBT, __SMULTB, __SMULTT, __SMULWB, __SMULWT

Syntax

```
unsigned int __SMULxxx(unsigned int, unsigned int);
```

Description

Inserts the specified instruction.

These intrinsic functions require architecture Armv6 or higher for Arm mode, and Armv7-A, Armv7-R, or Armv7E-M for Thumb mode.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

__sqrt, __sqrtf

Syntax

```
double __sqrt(double x);
float __sqrtf(float x);
```

Description

Computes the square root of the operand `x`, which corresponds either to the intrinsic call `__VSQRT_F64(x)` for double precision, or `__VSQRT_F32(x)` for single precision.

These intrinsic functions are defined according to the *Arm C Language Extensions* (ACLE).

__SSAT

Syntax

```
signed int __SSAT(signed int, unsigned int);
```

Description

Inserts an `SSAT` instruction.

The compiler will incorporate a shift instruction into the operand when possible. For example, `__SSAT(x << 3,11)` compiles to `SSAT Rd,#11,Rn,LSL #3`, where the value of `x` has been placed in register `Rn` and the return value of `__SSAT` will be placed in register `Rd`.

This intrinsic function requires architecture Armv6 or higher for Arm mode, and Armv7-A, Armv7-R, or Armv7-M for Thumb mode.

This intrinsic function is equivalent to the *Arm C Language Extensions* (ACLE) intrinsic function `__ssat`.

Note: This intrinsic function cannot be used **in 64-bit mode**.

__SSAT16

Syntax

```
unsigned int __SSAT16(unsigned int, unsigned int);
```

Description

Inserts an `SSAT16` instruction.

This intrinsic function requires architecture Armv6 or higher for Arm mode, and Armv7-A, Armv7-R, or Arm v7E-M for Thumb mode.

This intrinsic function is equivalent to the *Arm C Language Extensions* (ACLE) intrinsic function `__ssat16`.

Note: This intrinsic function cannot be used **in 64-bit mode**.

__STC, __STCL, __STC2, __STC2L

Syntax

```
void __STCxxx(__ul coproc, __ul CRn, __ul const *dst);
```

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.

The intrinsic functions `__STC` and `__STCL` require architecture Armv4 or higher for Arm mode, and Arm v6T2 or higher for Thumb mode.

The intrinsic functions `__STC2` and `__STC2L` require architecture Armv5 or higher for Arm mode, and Armv6-T2 or higher for Thumb mode.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

See also

__arm_stc, __arm_stcl, __arm_stc2, __arm_stc2l, page 434.

__STC_noidx, __STCL_noidx, __STC2_noidx, __STC2L_noidx

Syntax

```
void __STCxxx_noidx(__ul coproc, __ul CRn, __ul const *dst, __ul
option);
```

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.

The intrinsic functions `__STC_noidx` and `__STCL_noidx` require architecture Armv4 or higher for Arm mode, and Armv6-T2 or higher for Thumb mode.

The intrinsic functions `__STC2_noidx` and `__STC2L_noidx` require architecture Armv5 or higher for Arm mode, and Armv6-T2 or higher for Thumb mode.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

__STREX, __STREXB, __STREXD, __STREXH

Syntax

```
unsigned int __STREX(unsigned int, unsigned int *);
unsigned int __STREXB(unsigned char, unsigned char *);
unsigned int __STREXD(unsigned long long, unsigned long long*);
unsigned int __STREXH(unsigned short, unsigned short *);
```

Description

Inserts the specified instruction.

The `__STREX` intrinsic function requires architecture Armv6 or higher for Arm mode, and Armv6-T2 or baseline Armv8-M for Thumb mode.

The `__STREXB` and the `__STREXH` intrinsic functions require architecture Armv6K or Armv7 for Arm mode, and Armv7 or baseline Armv8-M for Thumb mode.

The `__STREXD` intrinsic function requires architecture Armv6K or Armv7 for Arm mode, and Armv7 except for Armv7-M for Thumb mode.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

__SWP, __SWPB

Syntax

```
unsigned int __SWP(unsigned int, unsigned int *);
char __SWPB(unsigned char, unsigned char *);
```

Description

Inserts the specified instruction. These intrinsic functions require Arm mode.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

__SXTAB, __SXTAB16, __SXTAH, __SXTB16

Syntax

```
unsigned int __SXTxxx(unsigned int, unsigned int);
```

Description

Inserts the specified instruction.

These intrinsic functions require architecture Armv6 or higher for Arm mode, and Armv7-A, Armv7-R, or Armv7E-M for Thumb mode.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

__TT, __TTT, __TTA, __TTAT

Syntax

```
unsigned int __TT(unsigned int);
unsigned int __TTT(unsugned int);
unsigned int __TTA(unsigned int);
unsigned int __TTAT(unsigned int);
```

Description

Inserts the specified instruction. Avoid using these intrinsic functions directly. Instead use the functions `cmse_TT`, `cmse_TTT`, `cmse_TT_fptr`, and `cmse_TTT_fptr`, which are defined in the header file `arm_cmse.h`.

These intrinsic functions require architecture Armv8-M with security extensions.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

See also

--*cmse*, page 284.

__UADD8, __UADD16, __UASX, __USAX, __USUB8, __USUB16

Syntax

```
unsigned int __Uxxx(unsigned int, unsigned int);
```

Description

Inserts the specified instruction.

These intrinsic functions require architecture Armv6 or higher for Arm mode, and Armv7-A, Armv7-R, or Armv7E-M for Thumb mode.

These intrinsic functions are equivalent to the *Arm C Language Extensions* (ACLE) intrinsic functions `__uadd8`, `__uadd16`, `__uasx`, `__usax`, `__usub8`, and `__usub16`.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

__UHADD8, __UHADD16, __UHASX, __UHSAX, __UHSUB8, __UHSUB16

Syntax

```
unsigned int __UHxxx(unsigned int, unsigned int);
```

Description

Inserts the specified instruction.

These intrinsic functions require architecture Armv6 or higher for Arm mode, and Armv7-A, Armv7-R, or Armv7E-M for Thumb mode.

These intrinsic functions are equivalent to the *Arm C Language Extensions* (ACLE) intrinsic functions `__uhadd8`, `__uhadd16`, `__uhasx`, `__uhsax`, `__uhsub8`, and `__uhsub16`.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

__UMAAL

Syntax

```
unsigned long long __UMAAL(unsigned int, unsigned int, unsigned
int, unsigned int);
```

Description

Inserts an `UMAAL` instruction.

This intrinsic function requires architecture Armv6 or higher for Arm mode, and Armv7-A, Armv7-R, or Armv7E-M for Thumb mode.

Note: This intrinsic function cannot be used **in 64-bit mode**.

__UQADD8, __UQADD16, __UQASX, __UQSAX, __UQSUB8, __UQSUB16

Syntax

```
unsigned int __UQxxx(unsigned int, unsigned int);
```

Description

Inserts the specified instruction.

These intrinsic functions require architecture Armv6 or higher for Arm mode, and Armv7-A, Armv7-R, or Armv7E-M for Thumb mode.

These intrinsic functions are equivalent to the *Arm C Language Extensions* (ACLE) intrinsic functions `__uqadd8`, `__uqadd16`, `__uqasx`, `__uqsax`, `__uqsub8`, and `__uqsub16`.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

__USAD8, __USADA8

Syntax

```
unsigned int __USADxxx(unsigned int, unsigned int);
```

Description

Inserts the specified instruction.

These intrinsic functions require architecture Armv6 or higher for Arm mode, and Armv7-A, Armv7-R, or Armv7E-M for Thumb mode.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

__USAT

Syntax

```
unsigned int __USAT(signed int, unsigned int);
```

Description

Inserts a `USAT` instruction.

The compiler will incorporate a shift instruction into the operand when possible. For example, `__USAT(x << 3,11)` compiles to `USAT Rd,#11,Rn,LSL #3`, where the value of `x` has been placed in register `Rn` and the return value of `__USAT` will be placed in register `Rd`.

This intrinsic function requires architecture Armv6 or higher for Arm mode, and Armv7-A, Armv7-R, or Armv7-M for Thumb mode.

This intrinsic function is equivalent to the *Arm C Language Extensions* (ACLE) intrinsic function `__usat`.

Note: This intrinsic function cannot be used **in 64-bit mode**.

__USAT16

Syntax

```
unsigned int __USAT16(unsigned int, unsigned int);
```

Description

Inserts a `USAT16` instruction.

This intrinsic function requires architecture Armv6 or higher for Arm mode, and Armv7-A, Armv7-R, or Armv7E-M for Thumb mode.

This intrinsic function is equivalent to the *Arm C Language Extensions* (ACLE) intrinsic function `__usat16`.

Note: This intrinsic function cannot be used **in 64-bit mode**.

__UXTAB, __UXTAB16, __UXTAH, __UXTB16

Syntax

```
unsigned int __UXTxxx(unsigned int, unsigned int);
```

Description

Inserts the specified instruction.

These intrinsic functions require architecture Armv6 or higher for Arm mode, and Armv7-A, Armv7-R, or Armv7E-M for Thumb mode.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

__VFMA_F64, __VFMS_F64, __VFNMA_F64, __VFNMS_F64, __VFMA_F32, __VFMS_F32, __VFNMA_F32, __VFNMS_F32

Syntax

```
double __VFMA_F64(double a, double x, double y);
double __VFMS_F64(double a, double x, double y);
double __VFNMA_F64(double a, double x, double y);
double __VFNMS_F64(double a, double x, double y);
float __VFMA_f32(float a, float x, float y);
float __VFMS_f32(float a, float x, float y);
float __VFNMA_F32(float a, float x, float y);
float __VFNMS_F32(float a, float x, float y);
```

Description

Inserts a fused floating-point multiply-accumulate instruction `VFMA`, `VFMS`, `VFNMA`, or `VFNMS`.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

__VMINNM_F64, __VMAXNM_F64, __VMINNM_F32, __VMAXNM_F32

Syntax

```
double __VMINNM_F64(double x, double y);
double __VMAXNM_F64(double x, double y);
float __VMINNM_F32(float x, float y);
float __VMAXNM_F32(float x, float y);
```

Description

Inserts a `VMINNM` or `VMAXNM` instruction.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

__VRINTA_F64, __VRINTM_F64, __VRINTN_F64, __VRINTP_F64, __VRINTX_F64, __VRINTR_F64, __VRINTZ_F64, __VRINTA_F32, __VRINTM_F32, __VRINTN_F32, __VRINTP_F32, __VRINTX_F32, __VRINTR_F32, __VRINTZ_F32

Syntax

```
double __VRINTA_F64(double x);
double __VRINTM_F64(double x);
double __VRINTN_F64(double x);
double __VRINTP_F64(double x);
double __VRINTX_F64(double x);
double __VRINTR_F64(double x);
double __VRINTZ_F64(double x);
float __VRINTA_F32(float x);
float __VRINTM_F32(float x);
float __VRINTN_F32(float x);
float __VRINTP_F32(float x);
float __VRINTX_F32(float x);
float __VRINTR_F32(float x);
float __VRINTZ_F32(float x);
```

Description

Performs a directed rounding and inserts the corresponding instruction:

- `VRINTA`: Rounds floating-point to integer to Nearest with Ties to Away
- `VRINTM`: Rounds floating-point to integer towards -Infinity
- `VRINTN`: Rounds floating-point to integer to Nearest
- `VRINTP`: Rounds floating-point to integer towards +Infinity
- `VRINTR`: Rounds floating-point to integer (using rounding mode in FPSCR)
- `VRINTX`: rounds floating-point to integer inexact (using rounding mode in FPSCR)
- `VRINTZ`: Rounds floating-point to integer towards Zero

If the result of, for example `__VRINTA_F64`, is converted to `int`, the instruction `VCVTA.S32.F64` is used instead. For conversion to `unsigned int`, the instruction `VCVTA.U32.F64` is used instead. Similarly, `VRINTM`, `VRINTN`, `VRINTP`, and `VRINTR` use corresponding instructions `VCVTM`, `VCVTN`, `VCVTP`, and `VCVTR` for integer conversion.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

__VSQRT_F64, __VSQRT_F32

Syntax

```
double __VSQRT_F64(double x);
float __VSQRT_F32(float x);
```

Description

Inserts the square root instruction `VSQRT`.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

__WFE, __WFI, __YIELD

Syntax

```
void int __xxx(void);
```

Description

Inserts the specified instruction.

These intrinsic functions require architecture Armv7 for Arm mode, and Armv6-M, or Armv7 for Thumb mode.

These intrinsic functions are equivalent to the *Arm C Language Extensions* (ACLE) intrinsic functions `__wfe`, `__wfi`, and `__yield`.

Note: These intrinsic functions cannot be used **in 64-bit mode**.

The preprocessor

- Overview of the preprocessor
- Description of predefined preprocessor symbols
- Descriptions of miscellaneous preprocessor extensions

Overview of the preprocessor

The preprocessor of the IAR C/C++ Compiler for Arm adheres to Standard C. 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 more information, see *Description of predefined preprocessor symbols*, page 472.
- 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 286.
- Preprocessor extensions

  There are several preprocessor extensions, for example, many pragma directives. For more information, see the chapter *Pragma directives*. For information about other extensions related to the preprocessor, see *Descriptions of miscellaneous preprocessor extensions*, page 487.
- Preprocessor output

  Use the option `--preprocess` to direct preprocessor output to a named file, see *--preprocess*, page 314.

To specify a path for an include file, use forward slashes:

```
#include "mydirectory/myfile"
```

In source code, use forward slashes:

```
file = fopen("mydirectory/myfile","rt");
```

Note: Backslashes can also be used—use one in include file paths and two in source code strings.

Description of predefined preprocessor symbols

This section lists and describes the preprocessor symbols.

Note: To list the predefined preprocessor symbols, use the compiler option `--predef_macros`. See *--predef_macros*, page 313.

__AAPCS__

Description

An integer that is set based on the compiler option `--aapcs`. The symbol is set to `1` if the AAPCS base standard is the selected calling convention (`--aapcs=std`). The symbol is undefined for other calling conventions.

This preprocessor symbol is equivalent to the ACLE (*Arm C Language Extensions*) macro `__ARM_PCS`.

See also

--*aapcs*, page 281.

__AAPCS_VFP__

Description

An integer that is set based on the compiler option `--aapcs`. The symbol is set to `1` if the VFP variant of AAPCS is the selected calling convention (`--aapcs=vfp`). The symbol is undefined for other calling conventions.

This preprocessor symbol is equivalent to the ACLE (*Arm C Language Extensions*) macro `__ARM_PCS_VFP`.

See also

--*aapcs*, page 281.

__aarch64__

Description

The symbol is set to `1` if A64 is the selected instruction set in AArch64.

See also

--*aarch64*, page 281, --*cpu_mode*, page 286, and --*abi*, page 281.

__arm__

Description

The symbol is defined when generating code for the A32 instruction set.

See also

--*cpu_mode*, page 286.

__ARM_32BIT_STATE

Description The symbol is defined when compiling in 32-bit mode.

This preprocessor symbol is defined according to ACLE (*Arm C Language Extensions*).

See also *Arm C Language Extensions* (IHI 0053D)

__ARM_64BIT_STATE

Description The symbol is defined when compiling in 64-bit mode.

This preprocessor symbol is defined according to ACLE (*Arm C Language Extensions*).

See also *Arm C Language Extensions* (IHI 0053D)

__ARM_ADVANCED_SIMD__

Description An integer that is set based on the compiler option `--cpu`. The symbol is set to `1` if the selected processor architecture has the Advanced SIMD architecture extension. The symbol is undefined for other cores.

This preprocessor symbol is equivalent to the ACLE (*Arm C Language Extensions*) macro `__ARM_NEON`.

See also *--cpu*, page 284.

__ARM_ALIGN_MAX_PWR

Description An integer that identifies the maximum alignment of static objects.

This preprocessor symbol is defined according to ACLE (*Arm C Language Extensions*).

See also *Arm C Language Extensions* (IHI 0053D)

__ARM_ALIGN_MAX_STACK_PWR

Description An integer that identifies the maximum alignment of stack objects.

This preprocessor symbol is defined according to ACLE (*Arm C Language Extensions*).

See also *Arm C Language Extensions* (IHI 0053D)

__ARM_ARCH

Description This symbol is defined according to ACLE (*Arm C Language Extensions*).

See also *Arm C Language Extensions* (IHI 0053D)

__ARM_ARCH_ISA_A64

Description This symbol is defined according to ACLE (*Arm C Language Extensions*).

See also *Arm C Language Extensions* (IHI 0053D)

__ARM_ARCH_ISA_ARM

Description This symbol is defined according to ACLE (*Arm C Language Extensions*).

See also *Arm C Language Extensions* (IHI 0053D)

__ARM_ARCH_ISA_THUMB

Description This symbol is defined according to ACLE (*Arm C Language Extensions*).

See also *Arm C Language Extensions* (IHI 0053D)

__ARM_ARCH_PROFILE

Description This symbol is defined according to ACLE (*Arm C Language Extensions*).

See also *Arm C Language Extensions* (IHI 0053D)

__ARM_BIG_ENDIAN

Description This symbol is defined according to ACLE (*Arm C Language Extensions*).

See also *Arm C Language Extensions* (IHI 0053D)

__ARM_FEATURE_AES

Description

This symbol is defined to 1 if the cryptographic AES instructions are supported.

This symbol is defined according to ACLE (*Arm C Language Extensions*).

__ARM_FEATURE_CLZ

Description

This symbol is defined to 1 if the CLZ instructions are supported.

This preprocessor symbol is defined according to ACLE (*Arm C Language Extensions*).

See also

Arm C Language Extensions (IHI 0053D)

__ARM_FEATURE_CMSE

Description

An integer that is set based on the compiler options `--cpu` and `--cmse`. The symbol is set to 3 if the selected processor architecture has CMSE (Cortex-M security extensions) and the compiler option `--cmse` is specified.

The symbol is set to 1 if the selected processor architecture has CMSE and the compiler option `--cmse` is not specified.

The symbol is undefined for cores without CMSE.

See also

--*cmse*, page 284 and --*cpu*, page 284

__ARM_FEATURE_CRC32

Description

This symbol is defined to 1 if the CRC32 instructions are supported (optional in Armv8-A/R).

This symbol is defined according to ACLE (*Arm C Language Extensions*).

__ARM_FEATURE_CRYPTO

Description

This symbol is defined to 1 if the cryptographic instructions are supported (implies Armv8-A/R with Neon).

This symbol is defined according to ACLE (*Arm C Language Extensions*).

__ARM_FEATURE_DIRECTED_ROUNDING

Description
This symbol is defined to 1 if the directed rounding and conversion instructions are supported.

This symbol is defined according to ACLE (*Arm C Language Extensions*).

__ARM_FEATURE_DSP

Description
This symbol is defined according to ACLE (*Arm C Language Extensions*).

See also
Arm C Language Extensions (IHI 0053D)

__ARM_FEATURE_FMA

Description
This symbol is defined to 1 if the FPU supports fused floating-point multiply-accumulate.

This symbol is defined according to ACLE (*Arm C Language Extensions*).

__ARM_FEATURE_FP16_FML

Description
This symbol is defined to 1 if the Armv8.2-A FP16 multiplication instructions are supported.

This symbol is defined according to ACLE (*Arm C Language Extensions*).

__ARM_FEATURE_IDIV

Description
This symbol is defined according to ACLE (*Arm C Language Extensions*).

See also
Arm C Language Extensions (IHI 0053D)

__ARM_FEATURE_NUMERIC_MAXMIN

Description
This symbol is defined to 1 if the floating-point maximum and minimum instructions are supported.

This symbol is defined according to ACLE (*Arm C Language Extensions*).

__ARM_FEATURE_QBIT

Description

This symbol is defined to `1` if the `Q` (saturation) global flag is supported.

This symbol is defined according to ACLE (*Arm C Language Extensions*).

__ARM_FEATURE_QRDMX

Description

This symbol is defined to `1` if `SQRDMLAH` and `SQRDMLSH` instructions are supported.

This symbol is defined according to ACLE (*Arm C Language Extensions*).

__ARM_FEATURE_SAT

Description

This symbol is defined to `1` if `SSAT` and `USAT` instructions are supported.

This symbol is defined according to ACLE (*Arm C Language Extensions*).

__ARM_FEATURE_SHA2

Description

This symbol is defined to `1` if the cryptographic SHA1 and SHA2 instructions are supported.

This symbol is defined according to ACLE (*Arm C Language Extensions*).

__ARM_FEATURE_SHA3

Description

This symbol is defined to `1` if the cryptographic SHA3 instructions are supported.

This symbol is defined according to ACLE (*Arm C Language Extensions*).

__ARM_FEATURE_SHA512

Description

This symbol is defined to `1` if the Armv8.2-A cryptographic SHA2 instructions are supported.

This symbol is defined according to ACLE (*Arm C Language Extensions*).

__ARM_FEATURE_SIMD32

Description

This symbol is defined to `1` if 32-bit SIMD instructions are supported.

This symbol is defined according to ACLE (*Arm C Language Extensions*).

__ARM_FEATURE_SM3

Description

This symbol is defined to `1` if the cryptographic SM3 instructions are supported.

This symbol is defined according to ACLE (*Arm C Language Extensions*).

__ARM_FEATURE_SM4

Description

This symbol is defined to `1` if the cryptographic SM4 instructions are supported.

This symbol is defined according to ACLE (*Arm C Language Extensions*).

__ARM_FEATURE_UNALIGNED

Description

This symbol is defined only if the target supports unaligned access, and unaligned access is allowed. The compiler option `--no_unaligned_access` can be used to disallow unaligned access.

This symbol is defined according to ACLE (*Arm C Language Extensions*).

__ARM_FP

Description

This symbol is defined according to ACLE (*Arm C Language Extensions*).

See also

Arm C Language Extensions (IHI 0053D)

__ARM_FP16_ARGS

Description

This symbol is defined according to ACLE (*Arm C Language Extensions*).

See also

Arm C Language Extensions (IHI 0053D)

__ARM_FP16_FML

Description — This symbol is defined to `1` if FP16 multiplication instructions from Armv8.2-A are supported. This symbol is defined according to ACLE (*Arm C Language Extensions*).

See also — *Arm C Language Extensions* (IHI 0053D)

__ARM_FP16_FORMAT_IEEE

Description — This symbol is defined according to ACLE (*Arm C Language Extensions*).

See also — *Arm C Language Extensions* (IHI 0053D)

__ARM_MEDIA__

Description — An integer that is set based on the compiler option `--cpu`. The symbol is set to `1` if the selected processor architecture has the Armv6 SIMD extensions for multimedia. The symbol is undefined for other cores.

This preprocessor symbol is equivalent to the ACLE (*Arm C Language Extensions*) macro `__ARM_FEATURE_SIMD32`.

See also — --*cpu*, page 284.

__ARM_NEON

Description — This symbol is defined according to ACLE (*Arm C Language Extensions*).

__ARM_NEON_FP

Description — This symbol is defined according to ACLE (*Arm C Language Extensions*).

__ARM_PCS_AAPCS64

Description — This symbol is defined to `1` if the default procedure calling standard for the translation unit conforms to the AAPCS64 standard.

This symbol is defined according to ACLE (*Arm C Language Extensions*).

__ARM_PROFILE_M__

Description

An integer that is set based on the compiler option `--cpu`. The symbol is set to `1` if the selected processor architecture is a profile M core. The symbol is undefined for other cores.

This preprocessor symbol is related to the ACLE (*Arm C Language Extensions*) macro `__ARM_ARCH_PROFILE`.

See also

--cpu, page 284.

__ARM_ROPI

Description

This symbol is defined to `1` if the translation unit is being compiled in read-only position-independent mode.

This symbol is defined according to ACLE (*Arm C Language Extensions*).

__ARM_RWPI

Description

This symbol is defined to `1` if the translation unit is being compiled in read/write position-independent mode.

This symbol is defined according to ACLE (*Arm C Language Extensions*).

__ARM_SIZEOF_MINIMAL_ENUM

Description

An integer that identifies the size of a minimal enumeration type (`1` or `4`).

This preprocessor symbol is defined according to ACLE (*Arm C Language Extensions*).

See also

Arm C Language Extensions (IHI 0053D)

__ARM_SIZEOF_WCHAR_T

Description

An integer that identifies the size of the `wchar_t` type (`2` or `4`).

This preprocessor symbol is defined according to ACLE (*Arm C Language Extensions*).

See also

Arm C Language Extensions (IHI 0053D)

__ARMVFP__

Description

An integer that reflects the `--fpu` option and is defined to `__ARMVFPV2__`, `__ARMVFPV3__`, `__ARMVFPV4__`, or `__ARMVFPV5__`. These symbolic names can be used when testing the `__ARMVFP__` symbol. If VFP code generation is disabled (default), the symbol will be undefined.

See also

--*fpu*, page 296.

__ARMVFP_D16__

Description

An integer that is set based on the compiler option `--fpu`. The symbol is set to `1` if the selected FPU is a VFPv3 or VFPv4 unit with only 16 D registers. Otherwise, the symbol is undefined.

See also

--*fpu*, page 296.

__ARMVFP_SP__

Description

An integer that is set based on the compiler option `--fpu`. The symbol is set to `1` if the selected FPU only supports 32-bit single-precision. Otherwise, the symbol is undefined.

This preprocessor symbol is related to the ACLE (*Arm C Language Extensions*) macro `__ARM_FP`.

See also

--*fpu*, page 296.

__BASE_FILE__

Description

A string that identifies the name of the base source file (that is, not the header file), being compiled.

See also

__FILE__, page 483, and --*no_path_in_file_macros*, page 306.

__BUILD_NUMBER__

Description

A unique integer that identifies the build number of the compiler currently in use.

__CORE__

Description

An integer that identifies the chip core in use. The value reflects the setting of the --cpu option and is defined to __ARM4TM__, __ARM5__, __ARM5E__, __ARM6__, __ARM6M__, __ARM6SM__, __ARM7M__, __ARM7EM__, __ARM7A__, __ARM7R__, __ARM8A__, __ARM8M_BASELINE__, __ARM8M_MAINLINE__, __ARM8R__, or __ARM8EM_MAINLINE__. These symbolic names can be used when testing the __CORE__ symbol.

This preprocessor symbol is related to the ACLE (*Arm C Language Extensions*) macro __ARM_ARCH.

__COUNTER__

Description

A macro that expands to a new integer each time it is expanded, starting at zero (0) and counting up.

__cplusplus

Description

An integer which is defined when the compiler runs in any of the C++ modes, otherwise it is undefined. When defined, its value is 201703L. 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.

This symbol is required by Standard C.

__CPU_MODE__

Description

An integer that reflects the selected CPU mode and is defined to 1 for Thumb (T or T32), 2 for Arm (A32), or 3 for A64.

__DATE__

Description

A string that identifies the date of compilation, which is returned in the form "Mmm dd yyyy", for example, "Oct 30 2018".

This symbol is required by Standard C.

__EXCEPTIONS

Description A symbol that is defined when exceptions are supported in C++.

See also --*no_exceptions*, page 304.

__FILE__

Description A string that identifies the name of the file being compiled, which can be both the base source file and any included header file.

This symbol is required by Standard C.

See also *__BASE_FILE__*, page 481, and --*no_path_in_file_macros*, page 306.

__func__

Description A predefined string identifier that is initialized with 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 also -*e*, page 293 and *__PRETTY_FUNCTION__*, page 484.

__FUNCTION__

Description A predefined string identifier that is initialized with the name of the function in which the symbol is used, similar to `char _FUNCTION_[]="main";` if used in `main()`. This is useful for assertions and other trace utilities. The symbol requires that language extensions are enabled.

See also -*e*, page 293 and *__PRETTY_FUNCTION__*, page 484.

__IAR_SYSTEMS_ICC__

Description An integer that identifies the IAR compiler platform. The current value is 9—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__

Description

An integer that is set to `1` when the code is compiled with the IAR C/C++ Compiler for Arm.

__ilp32__

Description

This symbol is defined if the ILP32 data model is used when compiling for the A64 instruction set in the AArch64 state.

See also

--*abi*, page 281.

__LINE__

Description

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.

This symbol is required by Standard C.

__LITTLE_ENDIAN__

Description

An integer that reflects the setting of the compiler option `--endian` 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.

This preprocessor symbol is related to the ACLE (*Arm C Language Extensions*) macro `__ARM_BIG_ENDIAN`.

__lp64__

Description

This symbol is defined if the LP64 data model is used when compiling for the A64 instruction set in the AArch64 state.

See also

--*abi*, page 281.

__PRETTY_FUNCTION__

Description

A predefined string identifier that is initialized with 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 also

-e, page 293 and *__func__*, page 483.

__ROPI__

Description

An integer that is defined when the compiler option `--ropi` is used.

This preprocessor symbol is equivalent to the ACLE (*Arm C Language Extensions*) macro `__ARM_ROPI`.

See also

--ropi, page 316.

__RTTI__

Description

A symbol that is defined when runtime type information (RTTI) is supported in C++.

See also

--no_rtti, page 306.

__RWPI__

Description

An integer that is defined when the `compiler option --rwpi` is used.

This preprocessor symbol is equivalent to the ACLE (*Arm C Language Extensions*) macro `__ARM_RWPI`.

See also

--rwpi, page 317.

__STDC__

Description

An integer that is set to `1`, which means the compiler adheres to Standard C. This symbol can be tested with `#ifdef` to detect whether the compiler in use adheres to Standard C.*

This symbol is required by Standard C.

__STDC_LIB_EXT1__

Description

An integer that is set to `201112L` and that signals that Annex K, *Bounds-checking interfaces*, of the C standard is supported.

See also *__STDC_WANT_LIB_EXT1__*, page 488.

__STDC_NO_ATOMICS__

Description Set to `1` if the compiler does not support atomic types nor `stdatomic.h`.

__STDC_NO_THREADS__

Description Set to `1` to indicate that the implementation does not support threads.

__STDC_NO_VLA__

Description Set to `1` to indicate that C variable length arrays, VLAs, are not enabled.

See also *--vla*, page 324.

__STDC_UTF16__

Description Set to `1` to indicate that the values of type `char16_t` are UTF-16 encoded.

__STDC_UTF32__

Description Set to `1` to indicate that the values of type `char32_t` are UTF-32 encoded.

__STDC_VERSION__

Description An integer that identifies the version of the C standard in use. The symbol expands to `201710L`, unless the `--c89` compiler option is used, in which case the symbol expands to `199409L`.

This symbol is required by Standard C.

__thumb__

Description The symbol is defined when generating code for the T or T32 instruction set.

See also *--cpu_mode*, page 286.

__TIME__

Description

A string that identifies the time of compilation in the form `"hh:mm:ss"`.

This symbol is required by Standard C.

__TIMESTAMP__

Description

A string constant that identifies the date and time of the last modification of the current source file. The format of the string is the same as that used by the `asctime` standard function (in other words, `"Tue Sep 16 13:03:52 2014"`).

__VER__

Description

An integer that identifies the version number of the IAR compiler in use. For example, version 5.11.3 is returned as 5011003.

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 Standard C 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: The assert macro is defined in the `assert.h` standard include file.

In the IDE, the `NDEBUG` symbol is automatically defined if you build your application in the Release build configuration.

See also

__aeabi_assert, page 155.

__STDC_WANT_LIB_EXT1__

Description

If this symbol is defined to 1 prior to any inclusions of system header files, it will enable the use of functions from Annex K, *Bounds-checking interfaces*, of the C standard.

See also

Bounds checking functionality, page 139 and *C bounds-checking interface*, page 498.

#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 Standard C `#error` directive is used. This directive is not recognized when the `--strict` compiler option is used.

C/C++ standard library functions

- C/C++ standard library overview
- DLIB runtime environment—implementation details

For detailed reference information about the library functions, see the online help system.

C/C++ standard library overview

The IAR DLIB Runtime Environment is a complete implementation of the C/C++ standard library, compliant with Standard C and C++. This library also supports floating-point numbers in IEC 60559 format, and it can be configured to include different levels of support for locale, file descriptors, multibyte characters, etc.

For more information about customization, 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 more information about library functions, see the chapter *Implementation-defined behavior for Standard C*.

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 set up a runtime library, see *Setting up the runtime environment*, page 133. The linker will include only those routines that are required—directly or indirectly—by your application.

For information about how you can override library modules with your own versions, see *Overriding library modules*, page 137.

ALTERNATIVE MORE ACCURATE LIBRARY FUNCTIONS

The default implementation of `cos`, `sin`, `tan`, and `pow` is designed to be fast and small. As an alternative, there are versions designed to provide better accuracy. They are named `__iar_`*`xxx`*`_accuratef` for `float` variants of the functions and `__iar_`*`xxx`*`_accuratel` for `long double` variants of the functions, and where *`xxx`* is `cos`, `sin`, etc.

To use these more accurate versions, use the linker option.

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 runtime environment are reentrant, but the following functions and parts are not reentrant because they need static data:

- Heap functions—`malloc`, `free`, `realloc`, `calloc`, etc. and the C++ operators `new` and `delete`
- Locale functions—`localeconv`, `setlocale`
- Multibyte functions—`mblen`, `mbrlen`, `mbrtowc`, `mbsrtowc`, `mbtowc`, `wcrtomb`, `wcsrtomb`, `wctomb`
- Rand functions—`rand`, `srand`
- Time functions—`asctime`, `localtime`, `gmtime`, `mktime`
- The miscellaneous functions `atexit`, `perror`, `strerror`, `strtok`
- Functions that use files or the heap in some way. This includes `scanf`, `sscanf`, `getchar`, `getwchar`, `putchar`, and `putwchar`. In addition, if you are using the options `--printf_multibytes` and `--dlib_config=Full`, the `printf` and `sprintf` functions (or any variants) can also use the heap.

Functions that can set `errno` are not reentrant, because an `errno` value resulting from one of these functions can be destroyed by a subsequent use of the function before it is read. This applies to math and string conversion functions, among others.

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.

THE LONGJMP FUNCTION

A `longjmp` is in effect a jump to a previously defined `setjmp`. Any variable length arrays or C++ objects residing on the stack during stack unwinding will not be destroyed. This can lead to resource leaks or incorrect application behavior.

DLIB runtime environment—implementation details

These topics are covered:

BRIEFLY ABOUT THE DLIB RUNTIME ENVIRONMENT

The DLIB runtime environment provides most of the important C and C++ standard library definitions that apply to embedded systems. These are of the following types:

- Adherence to a free-standing implementation of Standard C. The library supports most of the hosted functionality, but you must implement some of its base functionality. For more information, see the chapter *Implementation-defined behavior for Standard C*.
- Standard C library definitions, for user programs.
- C++ library definitions, for user programs.
- `CSTARTUP`, the module containing the start-up code, see the chapter *The DLIB runtime environment*.

- Runtime support libraries, for example, low-level floating-point routines.
- Intrinsic functions, allowing low-level use of Arm features. For more information, see the chapter *Intrinsic functions*.

In addition, the DLIB runtime environment includes some added C functionality, see *Added C functionality*, page 497.

C HEADER FILES

This section lists the C header files specific to the DLIB runtime environment. Header files may additionally contain target-specific definitions, which are documented in the chapter *Using C*.

This table lists the C header files:

Header file	Usage
`assert.h`	Enforcing assertions when functions execute
`complex.h`	Computing common complex mathematical functions
`ctype.h`	Classifying characters
`errno.h`	Testing error codes reported by library functions
`fenv.h`	Floating-point exception flags
`float.h`	Testing floating-point type properties
`inttypes.h`	Defining formatters for all types defined in `stdint.h`
`iso646.h`	Alternative spellings
`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
`stdalign.h`	Handling alignment on data objects
`stdarg.h`	Accessing a varying number of arguments
`stdatomic.h`	Adding support for atomic operations. Atomic operations are available in cores where the instruction set supports them.
`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

Table 38: Traditional Standard C header files—DLIB

Header file	Usage
stdlib.h	Performing a variety of operations
stdnoreturn.h	Adding support for non-returning functions
string.h	Manipulating several kinds of strings
tgmath.h	Type-generic mathematical functions
threads.h	Adding support for multiple threads of execution This functionality is not supported.
time.h	Converting between various time and date formats
uchar.h	Unicode functionality
wchar.h	Support for wide characters
wctype.h	Classifying wide characters

Table 38: Traditional Standard C header files—DLIB (Continued)

C++ HEADER FILES

This section lists the C++ header files:

- The C++ library header files

  The header files that constitute the Standard C++ library.
- The C++ C header files

  The C++ header files that provide the resources from the C library.

The C++ library header files

This table lists the header files that can be used in C++:

Header file	Usage
algorithm	Defines several common operations on containers and other sequences
array	Adding support for the array sequencer container
atomic	Adding support for atomic operations Atomic operations are available in cores where the instruction set supports them.
bitset	Defining a container with fixed-sized sequences of bits
chrono	Adding support for time utilities
codecvt	Adding support for conversions between encodings
complex	Defining a class that supports complex arithmetic

Table 39: C++ header files

Header file	Usage
condition_variable	Adding support for thread condition variables. This functionality is not supported.
deque	A deque sequence container
exception	Defining several functions that control exception handling
forward_list	Adding support for the forward list sequence container
fstream	Defining several I/O stream classes that manipulate external files
functional	Defines several function objects
future	Adding support for passing function information between threads This functionality is not supported.
hash_map	A map associative container, based on a hash algorithm
hash_set	A set associative container, based on a hash algorithm
initializer_list	Adding support for the initializer_list class
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
iterator	Defines common iterators, and operations on iterators
limits	Defining numerical values
list	A doubly-linked list sequence container
locale	Adapting to different cultural conventions
map	A map associative container
memory	Defines facilities for managing memory
mutex	Adding support for the data race protection object mutex. This functionality is not supported.
new	Declaring several functions that allocate and free storage
numeric	Performs generalized numeric operations on sequences
ostream	Defining the class that performs insertions
queue	A queue sequence container
random	Adding support for random numbers

Table 39: C++ header files (Continued)

Header file	Usage
ratio	Adding support for compile-time rational arithmetic
regex	Adding support for regular expressions
scoped_allocator	Adding support for the memory resource scoped_allocator_adaptor
set	A set associative container
shared_mutex	Adding support for the data race protection object shared_mutex. This functionality is not supported.
slist	A singly-linked list sequence container
sstream	Defining several I/O stream classes that manipulate string containers
stack	A stack sequence container
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
system_error	Adding support for global error reporting
thread	Adding support for multiple threads of execution. This functionality is not supported.
tuple	Adding support for the tuple class
typeinfo	Defining type information support
typeindex	Adding support for type indexes
typetraits	Adding support for traits on types
unordered_map	Adding support for the unordered map associative container
unordered_set	Adding support for the unordered set associative container
utility	Defines several utility components
valarray	Defining varying length array container
vector	A vector sequence container

Table 39: C++ header files (Continued)

Using Standard C libraries in C++

The C++ library works in conjunction with some 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 former puts all declared symbols in the global and `std` namespace, whereas the latter puts them in the global namespace only.

This table shows the new header files:

Header file	Usage
`cassert`	Enforcing assertions when functions execute
`ccomplex`	Computing common complex mathematical functions
`cctype`	Classifying characters
`cerrno`	Testing error codes reported by library functions
`cfenv`	Floating-point exception flags
`cfloat`	Testing floating-point type properties
`cinttypes`	Defining formatters for all types defined in `stdint.h`
`ciso646`	Alternative spellings
`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
`cstdalign`	Handling alignment on data objects
`cstdarg`	Accessing a varying number of arguments
`cstdatomic`	Adding support for atomic operations
`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
`cstdnoreturn`	Adding support for non-returning functions
`cstring`	Manipulating several kinds of strings
`ctgmath`	Type-generic mathematical functions
`cthreads`	Adding support for multiple threads of execution. This functionality is not supported.
`ctime`	Converting between various time and date formats
`cuchar`	Unicode functionality
`cwchar`	Support for wide characters

Table 40: New Standard C header files—DLIB

Header file	Usage
cwctype	Classifying wide characters

Table 40: 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`.

NOT SUPPORTED C/C++ FUNCTIONALITY

The following files have contents that are not supported by the IAR C/C++ Compiler:

- `threads.h`, `condition_variable`, `future`, `mutex`, `shared_mutex`, `thread`, `cthreads`

Some library functions will have the same address. This occurs, most notably, when the library function parameters differ in type but not in size, as for example, `cos(double)` and `cosl(long double)`.

The IAR C/C++ compiler does not support threads as described in the C11 and C++14 standards. However, using `DLib_Threads.h` and an RTOS, you can build an application with thread support. For more information, see *Managing a multithreaded environment*, page 165.

ATOMIC OPERATIONS

When you compile for cores with instruction set support for atomic accesses, the standard C and C++ atomic operations are available in the files `stdatomic.h` and `atomic`. If atomic operations are not available, the predefined preprocessor symbol `__STDC_NO_ATOMICS__` is defined to `1`. This is true both in C and C++.

Atomic operations that cannot be handled natively by the hardware are passed on to library functions. The IAR C/C++ Compiler for Arm does not include implementations for these functions. A template implementation can be found in the file `src\lib\atomic\libatomic.c`.

ADDED C FUNCTIONALITY

The DLIB runtime environment includes some added C functionality:

- C bounds-checking interface
- `DLib_Threads.h`
- `iar_dlmalloc.h`
- `LowLevelIOInterface.h`

- `stdio.h`
- `stdlib.h`
- `string.h`
- `time.h (time32.h, time64.h)`

C bounds-checking interface

The C library supports Annex K (*Bounds-checking interfaces*) of the C standard. It adds symbols, types, and functions in the header files `errno.h`, `stddef.h`, `stdint.h`, `stdio.h`, `stdlib.h`, `string.h`, `time.h` (`time32.h`, `time64.h`), and `wchar.h`.

To enable the interface, define the preprocessor extension `__STDC_WANT_LIB_EXT1__` to 1 prior to including any system header file. See *__STDC_WANT_LIB_EXT1__*, page 488.

As an added benefit, the compiler will issue warning messages for the use of unsafe functions for which the interface has a safer version. For example, using `strcpy` instead of the safer `strcpy_s` will make the compiler issue a warning message.

DLib_Threads.h

The `DLib_Threads.h` header file contains support for locks and thread-local storage (TLS) variables. This is useful for implementing thread support. For more information, see the header file.

iar_dlmalloc.h

The `iar_dlmalloc.h` header file contains support for the advanced (`dlmalloc`) heap handler. For more information, see *Heap considerations*, page 219.

LowLevelIOInterface.h

The header file `LowLevelInterface.h` contains declarations for the low-level I/O functions used by DLIB. See *The DLIB low-level I/O interface*, page 154.

stdio.h

These functions provide additional I/O functionality:

`fdopen`	Opens a file based on a low-level file descriptor.
`fileno`	Gets the low-level file descriptor from the file descriptor (`FILE*`).
`__gets`	Corresponds to `fgets` on `stdin`.
`getw`	Gets a `wchar_t` character from `stdin`.

putw	Puts a wchar_t character to stdout.
__ungetchar	Corresponds to ungetc on stdout.
__write_array	Corresponds to fwrite on stdout.

string.h

These are the additional functions defined in string.h:

strdup	Duplicates a string on the heap.
strcasecmp	Compares strings case-insensitive.
strncasecmp	Compares strings case-insensitive and bounded.
strnlen	Bounded string length.

time.h

There are two interfaces for using time_t and the associated functions time, ctime, difftime, gmtime, localtime, and mktime:

- The 32-bit interface supports years from 1900 up to 2035 and uses a 32-bit integer for time_t. The type and function have names like __time32_t, __time32, etc. This variant is mainly available for backwards compatibility.
- The 64-bit interface supports years from -9999 up to 9999 and uses a signed long long for time_t. The type and function have names like __time64_t, __time64, etc.

The interfaces are defined in three header files:

- time32.h defines __time32_t, time_t, __time32, time, and associated functions.
- time64.h defines __time64_t, time_t, __time64, time, and associated functions.
- time.h includes time32.h or time64.h depending on the definition of _DLIB_TIME_USES_64.

  If _DLIB_TIME_USES_64 is:

  - defined to 1, it will include time64.h.
  - defined to 0, it will include time32.h.
  - undefined, it will include time64.h.

In both interfaces, time_t starts at the year 1970.

An application can use either interface, and even mix them by explicitly using the 32 or 64-bit variants.

See also *__time32, __time64*, page 162.

`clock_t` is 8 bytes if `long` is 8 bytes and 64-bit `time.h` is used, otherwise it is 4 bytes.

By default, the time library does not support the timezone and daylight saving time functionality. To enable that functionality, use the linker option `--timezone_lib`. See *--timezone_lib*, page 362.

There are two functions that can be used for loading or force-loading the timezone and daylight saving time information from `__getzone`:

- `int _ReloadDstRules (void)`
- `int _ForceReloadDstRules (void)`

Both these functions return `0` for DST rules found and `-1` for DST rules not found.

NON-STANDARD IMPLEMENTATIONS

These functions do not work as specified by the C standard:

- `fopen_s` and `freopen`

  These functions will not propagate the `u` exclusivity attribute to the low-level interface.
- `towupper` and `towlower`

  These functions will only handle `A`, `...`, `Z` and `a`, `...`, `z`.
- `iswalnum`, `...`, `iswxdigit`

  These functions will only handle arguments in the range 0 to 127.
- The collate functions `strcoll` and `strxfrm` will not work as intended. The same applies to the C++ equivalent functionality.

SYMBOLS USED INTERNALLY BY THE LIBRARY

The system header files use intrinsic functions, symbols, pragma directives etc. Some are defined in the library and some in the compiler. These reserved symbols start with __ (double underscores) and should only be used by the library.

Use the compiler option `--predef_macros` to determine the value for any predefined symbols.

The symbols used internally by the library are not listed in this guide.

The linker configuration file

- Overview
- Declaring the build type
- Defining memories and regions
- Regions
- Section handling
- Section selection
- Using symbols, expressions, and numbers
- Structural configuration

Before you read this chapter you must be familiar with the concept of *sections*, see *Modules and sections*, page 96.

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:

- Declaring the build type

  informing the linker of whether the build is for a traditional ROM system or for a RAM system, helping the linker check that only suitable sections are placed in the different memory regions.
- 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.
- Special characters in names

  When specifying the name of a symbol or section that uses non-identifier characters, you can enclose the name in back quotes. Example: `` `My Name` ``.

Comments can be written either as C comments (`/*...*/`) or as C++ comments (`//...`).

Declaring the build type

Declaring the build type in the linker configuration files specifies to the linker whether the build is for a traditional ROM system (with, among other things, variable initialization at program start) or for a RAM system to be used for debugging (where other styles of initialization can be used).

build for directive

Syntax

```
build for { ram | rom };
```

Parameters

`ram` — The build is assumed to be a debugging or experimental setup, where some or all variable initialization can be performed at load time.

`rom` — The build is assumed to be a traditional ROM build, where all variable initialization is performed at program start.

Description

If you declare a build type of `rom`—and especially if you also declare which memory regions are ROM or RAM—the linker can perform better checking that only suitable sections are placed in the different memory regions. If you do not explicitly specify an `initialize` directive (see *initialize directive*, page 517), the linker will behave as if you had specified `initialize by copy { rw };`.

If you declare a build type of `ram`, the linker does not check which section types are placed in which memory region.

If you do not include the `build for` directive in the linker configuration file, the linker only performs limited checking. This is useful primarily for backward compatibility purposes.

See also

define region directive, page 504.

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 504.
- 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. You can also use this directive to declare whether a region contains RAM or ROM memory. This is primarily useful when building for a traditional ROM system. See *define region directive*, page 504.

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 508.

This section gives detailed information about each linker directive specific to defining memories and regions.

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 531.

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 [ ram | rom ] name = region-expr;
```

where *region-expr* is a region expression, see also *Regions*, page 507.

Parameters

`ram`	The region contains RAM memory.
`rom`	The region contains ROM memory.
`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—these ranges do not need to be consecutive or even in the same memory.

If you declare regions as being ROM or RAM, the linker can check that only suitable sections are placed in the regions if you are building a traditional ROM-based system (see *build for directive*, page 503).

Example

```
/* Define the 0x10000-byte code region ROM located at address
   0x10000 */
define rom region ROM = [from 0x10000 size 0x10000];
```

logical directive

Syntax

```
logical range-list = physical range-list
```

where *`range-list`* is one of

```
[ region-expr,... ]region-expr
[ region-expr,... ]from address-expr
```

Parameters

`region-expr`	A region expression, see also *Regions*, page 507.
`address-expr`	An address expression

Description

The `logical` directive maps logical addresses to physical addresses. The physical address is typically used when loading or burning content into memory, while the logical address is the one seen by your application. The physical address is the same as the logical address, if no `logical` directives are used, or if the address is in a range specified in a `logical` directive.

When generating ELF output, the mapping affects the physical address in program headers. When generating output in the Intel hex or Motorola S-records formats, the physical address is used.

Each address in the logical range list, in the order specified, is mapped to the corresponding address in the physical range list, in the order specified.

Unless one or both of the range lists end with the `from` form, the total size of the logical ranges and the physical ranges must be the same. If one side ends with the `from` form and not the other, the side that ends with the `from` form will include a final range of a size that makes the total sizes match, if possible. If both sides end with a `from` form, the ranges will extend to the highest possible address that makes the total sizes match.

Setting up a mapping from logical to physical addresses can affect how sections and other content are placed. No content will be placed to overlap more than one individual logical or physical range. Also, if there is a mapping from a different logical range to the corresponding physical range, any logical range for which no mapping to physical ranges has been specified—by not being mentioned in a `logical` directive—is excluded from placement.

All `logical` directives are applied together. Using one or using several directives to specify the same mapping makes no difference to the result.

Example

```
// Logical range 0x8000-0x8FFF maps to physical 0x10000-0x10FFF.
// No content can be placed in the logical range 0x10000-0x10FFF.
logical [from 0x8000 size 4K] = physical [from 0x10000 size 4K];

// Another way to specify the same mapping
logical [from 0x8000 size 4K] = physical from 0x10000;

// Logical range 0x8000-0x8FFF maps to physical 0x10000-0x10FFF.
// Logical range 0x10000-0x10FFF maps to physical 0x8000-0x8FFF.
// No logical range is excluded from placement because of
// this mapping.
logical [from  0x8000 size 4K] = physical [from 0x10000 size 4K];
logical [from 0x10000 size 4K] = physical [from  0x8000 size 4K];

// Logical range 0x1000-0x13FF maps to physical 0x8000-0x83FF.
// Logical range 0x1400-0x17FF maps to physical 0x9000-0x93FF.
// Logical range 0x1800-0x1BFF maps to physical 0xA000-0xA3FF.
// Logical range 0x1C00-0x1FFF maps to physical 0xB000-0xB3FF.
// No content can be placed in the logical ranges 0x8000-0x83FF,
// 0x9000-0x9FFF, 0xA000-0xAFFF, or 0xB000-0xBFFF.
logical [from 0x1000 size 4K] =
   physical [from 0x8000 size 1K repeat 4 displacement 4K];
```

```
// Another way to specify the same mapping.
logical [from 0x1000 to 0x13FF] = physical [from 0x8000 to
0x83FF];
logical [from 0x1400 to 0x17FF] = physical [from 0x9000 to
0x93FF];
logical [from 0x1800 to 0x1BFF] = physical [from 0xA000 to
0xA3FF];
logical [from 0x1C00 to 0x1FFF] = physical [from 0xB000 to
0xB3FF];
```

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 531.

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 expr`	*expr* is the start address of the memory range (inclusive).
`to expr`	*expr* is the end address of the memory range (inclusive).
`size expr`	*expr* is the size of the memory range.
`repeat expr`	*expr* defines several ranges in the same memory for the region literal.
`displacement expr`	*expr* is the 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 such a 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 504, and *Region expression*, page 508.

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 504

where *region-literal* is a region literal, see *Region literal*, page 507

and where *empty-region* is an empty region, see *Empty region*, page 509.

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 508.

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 in` directives place sets of sections with similar attributes into previously defined regions. See *place at directive*, page 521 and *place in directive*, page 523.

- Making sets of sections with special requirements

  The `block` directive makes it possible to create empty sections with specific or expanding sizes, specific 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 511, and *define overlay directive*, page 516.

- 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 517 and *do not initialize directive*, page 520.

- 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 521.

- Specifying the contents of linker-generated sections

  The `define section` directive can be used for creating specific sections with content and calculations that are only available at link time.
- Additional more specialized directives:

  `use init table` directive

This section gives detailed information about each linker directive specific to section handling.

define block directive

Syntax

```
define [ movable ] block name
  [ with param, param... ]
{
  extended-selectors
}
[ except
 {
   section-selectors
  } ];
```

where *param* can be one of:

```
size = expr
minimum size = expr
maximum size = expr
expanding size
alignment = expr
end alignment = expr
fixed order
alphabetical order
static base [basename]
```

and where the rest of the directive selects sections to include in the block, see *Section selection*, page 524.

Parameters

name	The name of the block to be defined.
`size`	Customizes the size of the block. By default, the size of a block is the sum of its parts dependent of its contents.
`minimum size`	Specifies a lower limit for the size of the block. The block is at least this large, even if its contents would otherwise not require it.

`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.
`expanding size`	The block will expand to use all available space in the memory range where it is placed.
`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.
`end alignment`	Specifies a minimum alignment for the end of the block. Normally, the end address of a block is determined by its start address and its size (which can depend on its contents), but if this parameter is used, the end address is increased to comply with the specified alignment if needed.
`fixed order`	Places sections in the specified order. Each *`extended-selector`* is added in a separate nested block, and these blocks are kept in the specified order.
`alphabetical order`	Places sections in alphabetical order by section name. Only *`section-selector`* patterns are allowed in `alphabetical order` blocks, for example, no nested blocks. All sections in a particular `alphabetical order` block must use the same kind of initialization (read-only, zero-init, copy-init, or no-init, and otherwise equivalent). You cannot use `__section_begin`, etc on individual sections contained in an `alphabetical order` block.
`static base` **[***`basename`***]**	**32-bit mode only**: Specifies that the static base with the name *`basename`* will be placed at the start of the block or in the middle of the block, as appropriate for the particular static base. The startup code must ensure that the register that holds the static base is initialized to the correct value. If there is only one static base, the name can be omitted. When linking for `--rwpi`, the *`basename`* for writable data must be `SB`, and when linking for `--ropi_cb`, the *`basename`* for read-only data must be `CB`.

Description

The `block` directive defines a contiguous area of memory that contains a possibly empty set of sections or other blocks. Blocks with no content are useful for allocating space for stacks or heaps. Blocks with content are usually used to group together sections that must to be consecutive.

You can access the start, end, and size of a block from an application by using the `__section_begin`, `__section_end`, or `__section_size` operators. If there is no block with the specified name, but there are sections with that name, a block will be created by the linker, containing all such sections.

`movable` blocks are for use with read-only and read-write position independence. Making blocks movable enables the linker to validate the application's use of addresses. Movable blocks are located in exactly the same way as other blocks, but the linker will check that the appropriate relocations are used when referring to symbols in movable blocks.

Blocks with expanding size are most often used for heaps or stacks.

Note: You cannot place a block with expanding size inside another block with expanding size, inside a block with a maximum size, or inside an overlay.

Example

```
/* Create a block with a minimum size for the heap that will use
all remaining space in its memory range */
define block HEAP with minimum size = 4K, expanding size,
alignment = 8 { };
```

See also

Interaction between the tools and your application, page 221. For an accessing example, see *define overlay directive*, page 516.

define section directive

Syntax

```
define [ root ] section name
  [ with alignment = sec-align ]
{
  section-content-item...
};
```

where each *section-content-item* can be one of:

```
udata8 { data | string };
sdata8 data [ ,data ] ...;
udata16 data [ ,data ] ...;
sdata16 data [ ,data ] ...;
udata24 data [ ,data ] ...;
sdata24 data [ ,data ] ...;
udata32 data [ ,data ] ...;
sdata32 data [ ,data ] ...;
udata64 data [ ,data ] ...;
sdata64 data [ ,data ] ...;
pad_to data-align;
[ public ] label:
if-item;
```

where *if-item* is:

```
if ( condition ) {
   section-content-item...
[} else if (condition] {
   section-content-item... ]...
[} else {
   section-content-item...]
}
```

Parameters

name	The name of the section.
sec-align	The alignment of the section, an expression.
root	Optional. If root is specified, the section is always included, even if it is not referenced.
udata8 **{***data*|*string***}**;	If the parameter is an expression (*data*), it generates an unsigned one-byte member in the section. The *data* expression is only evaluated during relocation and only if the value is needed. It causes a relocation error if the value of *data* is too large to fit in a byte. The possible range of values is 0 to 0xFF. If the parameter is a quoted string, it generates one one-byte member in the section for each character in the string.
sdata8 *data*;	As udata8 *data*, except that it generates a signed one-byte member. The possible range of values is –0x80 to 0x7F.
udata16 *data*;	As sdata8, except that it generates an unsigned two-byte member. The possible range of values is 0 to 0xFFFF.
sdata16 *data*;	As sdata8, except that it generates a signed two-byte member. The possible range of values is –0x8000 to 0x7FFF.
udata24 *data*;	As sdata8, except that it generates an unsigned three-byte member. The possible range of values is 0 to 0xFFFF'FF.

`sdata24 data;`	As `sdata8`, except that it generates a signed three-byte member. The possible range of values is –`0x8000'00` to `0x7FFF'FF`.
`udata32 data;`	As `sdata8`, except that it generates an unsigned four-byte member. The possible range of values is `0` to `0xFFFF'FFFF`.
`sdata32 data;`	As `sdata8`, except that it generates a signed four-byte member. The possible range of values is –`0x8000'0000` to `0x7FFF'FFFF`.
`udata64 data;`	As `sdata8`, except that it generates an unsigned eight-byte member. The possible range of values is `0` to `0xFFFF'FFFF'FFFF'FFFF`.
`sdata64 data;`	As `sdata8`, except that it generates a signed eight-byte member. The possible range of values is `-0x8000'0000'0000'0000` to `0x7FFF'FFFF'FFFF'FFFF`.
`pad_to data_align;`	Generates pad bytes to make the current offset from the start of the section to be aligned to the expression *`data-align`*.
`[public] label:`	Defines a label at the current offset from the start of the section. If `public` is specified, the label is visible to other program modules. If not, it is only visible to other data expressions in the linker configuration file.
`if-item`	Configuration-time selection of items.
`condition`	An expression.
`data`	An expression that is only evaluated during relocation and only if the value is needed.

Description

Use the `define section` directive to create sections with content that is not available from assembler language or C/C++. Examples of this are the results of stack usage analysis, the size of blocks, and arithmetic operations that do not exist as relocations.

Unknown identifiers in data expressions are assumed to be labels.

Note: Only data expressions can use labels, stack usage analysis results, etc. All the other expressions are evaluated immediately when the configuration file is read.

Example

```
define section data {
  /* The application entry in a 16-bit word, provided it is less
     than 256K and 4-byte aligned. */
  udata16 __iar_program_start >> 2;
  /* The maximum stack usage in the program entry category. */
  udata16 maxstack("Application entry");
  /* The size of the DATA block */
  udata32 size(block DATA);
};
```

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 524.

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: 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.

Code in overlaid memory areas cannot be debugged; the C-SPY Debugger cannot determine which code is currently loaded.

See also

Manual initialization, page 119.

initialize directive

Syntax

```
initialize { by copy | manually }
  [ with param, param... ]
{
  section-selectors
}
[ except
 {
   section-selectors
 } ];
```

where *param* can be one of:

```
packing = algorithm
simple ranges
complex ranges
no exclusions
```

For information about section selectors and except clauses, see *Section selection*, page 524.

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.
`algorithm`	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 consecutive bytes with the value zero. `packbits` - Compresses with the PackBits algorithm. This method generates good results for data with many identical consecutive bytes. `lz77` - Compresses with the Lempel-Ziv-77 algorithm. This method handles a larger variety of inputs well, but has a slightly larger decompressor. `auto` - 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. This is the default method for `initialize by copy`. `smallest` - This is a synonym for `auto`.

Description

The `initialize` directive splits each selected section into one section that holds initializer data and another section that holds the space for the initialized data. The section that holds the space for the initialized data retains the original section name, and the section that holds initializer data gets the name suffix `_init`. 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`), ILINK will automatically choose an appropriate packing algorithm for the initializers. To override this, specify a different `packing` method. 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.

Each decompressor has two variants: one that can only handle a single source and destination range at a time, and one that can handle more complex cases. By default, the linker chooses a decompressor variant based on whether the associated section placement directives specify a single or multi-range memory region. In general, this is the desired behavior, but you can use the `with complex ranges` or the `with simple ranges` modifier on an `initialize` directive to specify which decompressor variant to use. You can also use the command line option `--default_to_complex_ranges` to make `initialize` directives by default use complex ranges. The `simple ranges` decompressors are normally hundreds of bytes smaller than the `complex ranges` variants.

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.

Due to an internal dependence, generation of compressed initializers can also fail (with error LP021) if the address of the initialized area depends on the size of its initializers. To avoid this, place the initializers and the initialized area in different parts of the memory (for example, the initializers are placed in ROM and the initialized area in RAM).

If you specify the parameter `no exclusions`, an error is emitted if any sections are excluded (because they are needed for the initialization). `no exclusions` can only be used with `initialize by copy` (automatic initialization), not with `initialize manually`.

Unless `initialize manually` is used, ILINK will arrange for initialization to occur during system startup by including an initialization table. Startup code calls an initialization routine that reads this table and performs the necessary initializations.

Zero-initialized sections are not affected by the `initialize` directive.

The `initialize` directive is normally used for initialized variables, but can be used for copying any sections, for example, copying executable code from slow ROM to fast RAM, or for overlays. For another example, see *define overlay directive*, page 516.

Sections that are needed for initialization 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, in addition to logging the marking of section fragments to be included in the application, also logs the process of determining which sections are needed for initialization.

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 };
```

See also *Initialization at system startup*, page 102, and *do not initialize directive*, page 520.

do not initialize directive

Syntax

```
do not initialize
{
  section-selectors
}
[ except
 {
   section-selectors
 } ];
```

For information about section selectors and except clauses, see *Section selection*, page 524.

Description

Use the `do not initialize` directive to specify the sections that you do not want to be automatically zero-initialized by the system startup code. The directive can only be used on `zeroinit` sections.

Typically, this is useful if you want to handle zero-initialization in some other way for all or some `zeroinit` sections.

This can also be useful if you want to suppress zero-initialization of variables entirely. Normally, this is handled automatically for variables specified as `__no_init` in the source, but if you link with object files produced by older tools from IAR Systems or other tool vendors, you might need to suppress zero-initialization specifically for some sections.

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 102, and *initialize directive*, page 517.

keep directive

Syntax

```
keep
{
  [ { section-selectors | block name }
    [ , {section-selectors | block name }... ] ]
}
[ except
 {
   section-selectors
 } ];
```

For information about selectors and except clauses, see *Section selection*, page 524.

Description

The `keep` directive can be used for including blocks, overlays, or sections in the executable image that would otherwise be discarded because no references to them exist in the included parts of the application. Note that this directive always causes entire input sections to be included, and not just the relevant section fragment, when matching against a symbol name.

Furthermore, only sections from included modules are considered. The `keep` directive does not cause any additional modules to be included in your application.

To cause a module that defines a specific symbol to be included, or only the section fragment that defines a symbol, use the **Keep symbols** linker option (or the `--keep` option on the command line), or the linker directive `keep symbol`.

Example

```
keep { section .keep* } except {section .keep};
```

place at directive

Syntax

```
[ "name": ]
place [ noload ] at { address [ memory: ] address |
      start of region_expr [ with mirroring to mirror_address ] |
      end of region_expr [ with mirroring to mirror_address ] }

{
  extended-selectors
}
[ except
 {
   section-selectors
 } ];
```

For information about extended selectors and except clauses, see *Section selection*, page 524.

Parameters

name	Optional. If it is specified, it is used in the map file, in some log messages, and is part of the name of any ELF output sections resulting from the directive.
`noload`	Optional. If it is specified, it prevents the sections in the directive from being loaded to the target system. To use the sections, you must put them into the target system in some other way. `noload` can only be used when a *name* is specified.
memory: *address*	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 that results in a single-internal region. The start of the interval is used.
`end of` *region_expr*	A region expression that results in a single-internal region. The end of the interval is used.
mirror_address	If `with mirroring to` is specified, the contents of any sections are assumed to be mirrored to this address, therefore debug information and symbols will appear in the mirrored range, but the actual content bytes are placed as if `with mirroring to` was not specified. **Note:** This functionality is intended to support external (target-specific) mirroring.

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.

Note: `with mirroring to` can be used only together with `start of` and `end of`.

Example

```
/* Place the RO section .startup at the start of code_region */
"START": place at start of ROM { readonly section .startup };
```

See also

place in directive, page 523.

place in directive

Syntax

```
[ "name": ]
place [ noload ] in region-expr
      [ with mirroring to mirror_address ]
{
  extended-selectors
}
 [ except{
   section-selectors
 } ];
```

where `region-expr` is a region expression, see also *Regions*, page 507.

and where the rest of the directive selects sections to include in the block. See *Section selection*, page 524.

Parameters

name	Optional. If it is specified, it is used in the map file, in some log messages, and is part of the name of any ELF output sections resulting from the directive.
`noload`	Optional. If it is specified, it prevents the sections in the directive from being loaded to the target system. To use the sections, you must put them into the target system in some other way. `noload` can only be used when a *name* is specified.
mirror_address	If `with mirroring to` is specified, the contents of any sections are assumed to be mirrored to this address, therefore debug information and symbols will appear in the mirrored range, but the actual content bytes are placed as if `with mirroring to` was not specified. **Note:** This functionality is intended to support external (target-specific) mirroring.

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.

Note: When `with mirroring to` is specified, the *`region-expr`* must result in a single range.

Example

```
/* Place the read-only sections in the code_region */
"ROM": place in ROM { readonly };
```

See also *place at directive*, page 521.

use init table directive

Syntax

```
use init table name for
{
  section-selectors
}
[ except
 {
   section-selectors
 } ];
```

For information about section selectors and except clauses, see *Section selection*, page 524.

Parameters

name	The name of the `init` table.

Description

Normally, all initialization entries are generated into a single initialization table (called `Table`). Use this directive to cause some of the entries to be put into a separate table. You can then use this initialization table at another time, or under different circumstances, than the normal initialization table.

Initialization entries for all variables not mentioned in a `use init table` directive are put into the normal initialization table. By having multiple `use init table` directives you can have multiple initialization tables.

The start, end, and size of the `init` table can be accessed in the application program by using `__section_begin`, `__section_end`, or `__section_size` of `"Region$$name"`, respectively, or via the symbols `Region$$name$$Base`, `Region$$name$$Limit`, and `Region$$name$$Length`.

Example

```
use init table Core2 for { section *.core2};

/* __section_begin("Region$$Core2") can be used to get the start
   of the Core2 init table. */
```

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.

This section gives detailed information about each linker directive specific to section selection.

section-selectors

Syntax

```
[ section-selector [ , section-selector... ] ]
```

section-selector is:

```
[ section-attribute ][ section-type ]
[ symbol symbol-name ][ section section-name ]
[ object module-spec ]
```

section-attribute is:

```
ro [ code | data ] | rw [ code | data ] | zi
```

section-type is:

```
[ preinit_array | init_array ]
```

Parameters

`section-attribute` — Only sections with the specified attribute will be selected. `section-attribute` can consist of:

`ro|readonly`, for ROM sections.
`rw|readwrite`, for RAM sections.

In each category, sections can be further divided into those that contain code and those that contain data, resulting in four main categories:

`ro code`, for normal code
`ro data`, for constants
`rw code`, for code copied to RAM
`rw data`, for variables

`readwrite data` also has a subcategory—`zi|zeroinit`—for sections that are zero-initialized at application startup.

section-type	Only sections with that ELF section type will be selected. *section-type* can be: preinit_array, sections of the ELF section type SHT_PREINIT_ARRAY. init_array, sections of the ELF section type SHT_INIT_ARRAY.
symbol *symbol-name*	Only sections that define at least one public symbol that matches the symbol name pattern will be selected. *symbol-name* is the symbol name pattern. Two wildcards are allowed: ? matches any single character. * matches zero or more characters.
section *section-name*	Only sections whose names match the *section-name* will be selected. Two wildcards are allowed: ? matches any single character * matches zero or more characters.
object *module-spec*	Only sections that originate from library modules or object files that matches *module-spec* will be selected. *module-spec* can be in one of two forms: *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. If *libraryname* is :sys, the pattern will match only sections from the system library. *filename*, the name of an object file, 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 type, symbol name, section name, and the name of the module. Up to four of the five conditions can be omitted.

It is also possible to use only `{ }` without any section selectors, which can be useful when defining blocks.

Note: A section selector with narrower scope has higher priority than a more generic section selector. If more than one section selector matches for the same purpose, one of them must be more specific. A section selector is more specific than another one if in priority order:

- It specifies a symbol name with no wildcards and the other one does not.
- It specifies a section name or object name with no wildcards and the other one does not
- It specifies a section type and the other one does not
- There could be sections that match the other selector that also match this one, however, the reverse is not true.

Selector 1	Selector 2	More specific
`ro`	`ro code`	Selector 2
`symbol mysym`	`section foo`	Selector 1
`ro code section f*`	`ro section f*`	Selector 1
`section foo*`	`section f*`	Selector 1
`section *x`	`section f*`	Neither
`init_array`	`section f*`	Selector 1
`section .intvec`	`ro section .int*`	Selector 1
`section .intvec`	`object foo.o`	Neither

Table 41: Examples of section selector specifications

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 517, *do not initialize directive*, page 520, and *keep directive*, page 521.

extended-selectors

Syntax

```
[ extended-selector [ , extended-selector...]]
```

where *extended-selector* is:

```
[ first | last | midway ]
                   { section-selector |
                     block name [ inline-block-def ] |
                     overlay name }
```

where *inline-block-def* is:

```
[ block-params ] extended-selectors
```

Parameters

`first`	Places the selected sections, block, or overlay first in the containing placement directive, block, or overlay.
`last`	Places the selected sections, block or overlay last in the containing placement directive, block, or overlay.
`midway`	Places the selected sections, block, or overlay so that they are no further than half the maximum size of the containing block away from either edge of the block. Note that this parameter can only be used inside a block that has a maximum size.
`name`	The name of the block or overlay.

Description

Use `extended-selectors` to select content for inclusion in a placement directive, block, or overlay. In addition to using section selection patterns, you can also explicitly specify blocks or overlays for inclusion.

Using the `first` or `last` keyword, you can specify one pattern, block, or overlay that is to be placed first or last in the containing placement directive, block, or overlay. If you need more precise control of the placement order you can instead use a block with fixed order.

Blocks can be defined separately, using the `define block` directive, or inline, as part of an `extended-selector`.

The `midway` parameter is primarily useful together with a static base that can have both negative and positive offsets.

Example

```
define block First { ro section .f* }; /* Define a block holding
                                          any read-only section*/
                                          matching ".f*" */
define block Table { first block First, ro section .b };
                                       /* Define a block where
                                          the block First comes
                                          before the sections
                                          matching ".b*". */
```

You can also define the block `First` inline, instead of in a separate `define block` directive:

```
define block Table { first block First { ro section .f* },
                     ro section .b* };
```

See also

define block directive, page 511, *define overlay directive*, page 516, and *place at directive*, page 521.

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 530, and *export directive*, page 531.
- *Use expressions and numbers*

  In the linker configuration file, expressions and numbers are used for specifying addresses, sizes, etc. See *expressions*, page 531.

This section gives detailed information about each linker directive specific to defining symbols, expressions and numbers.

check that directive

Syntax

```
check that expression;
```

Parameters

expression	A boolean expression.

Description

You can use the `check that` directive to compare the results of stack usage analysis against the sizes of blocks and regions. If the expression evaluates to zero, an error is emitted.

Three extra operators are available for use only in `check that` expressions:

`maxstack(`*`category`*`)`	The stack depth of the deepest call chain for any call graph root function in the category.
`totalstack(`*`category`*`)`	The sum of the stack depths of the deepest call chains for each call graph root function in the category.
`size(`*`block`*`)`	The size of the block.

Example

```
check that maxstack("Program entry")
             + totalstack("interrupt")
             + 1K
           <= size(block CSTACK);
```

See also

Stack usage analysis, page 105.

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 531 and *Interaction between ILINK and the application*, page 122.

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 533

where *symbol* is a defined symbol, see *define symbol directive*, page 530 and *--config_def*, page 334

and where *func-operator* is one of these function-like operators, available in all expressions:

`aligndown(expr, align)`	The value of *expr* rounded down to the nearest multiple of *align*. *align* must be a power of two.
`alignup(expr, align)`	The value of *expr* rounded up to the nearest multiple of *align*. *align* must be a power of two.
`end(region)`	The highest address in the region.
`isdefinedsymbol(name)`	True (`1`) if the symbol *name* is defined, otherwise False (`0`).
`isempty(region)`	True (`1`) if the region is empty, otherwise False (`0`).
`max(expr [, expr... ] )`	The largest of the parameters.
`min(expr [, expr... ] )`	The smallest of the parameters.
`size(region)`	The total size of all ranges in the region.
`start(region)`	The lowest address in the region.

where *align* and *expr* are expressions, and *region* is a region expression, see *Region expression*, page 508.

func-operator can also be one of these operators, which are only available in expressions for the size or alignment of a block or overlay, in `check that` expressions, and in data expressions in `define section` directives:

`imp(name)`	If *name* is the name of a symbol with a constant value, this operator is that value. This operator can be used together with `#pragma public_equ` (see *public_equ*, page 419) to import values from modules in your application, for example the size of a particular struct type.
`tlsalignment()`	The alignment of the thread-local storage area.
`tlssize()`	The size of the thread-local storage area.

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).

keep symbol directive

Syntax

`keep symbol` *`name`*`;`

Parameters

name	The name of the symbol.

Description

Normally, the linker keeps a symbol only if it is needed by your application. Use this directive to ensure that a symbol is always included in the final application.

See also

keep directive, page 521 and --*keep*, page 347.

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 534.
- 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 535.
- Signaling an error for unsupported cases

This section gives detailed information about each linker directive specific to structural configuration.

error directive

Syntax

```
error string
```

Parameters

string	The error message.

Description

An `error` directive can be used for signaling an error if the directive occurs in the active part of a conditional directive.

Example

```
error "Unsupported configuration"
```

if directive

Syntax

```
if (expr) {
  directives
[ } else if (expr) {
  directives ]
[ } else {
  directives ]
}
```

where *expr* is an expression, see *expressions*, page 531.

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 text inside a non-selected part of an `if` directive is not checked for syntax. The only requirements for such text, is that it can be tokenized, and that any open brace (`{`) token has a matching close brace (`}`) token.

Example

See *Empty region*, page 509.

include directive

Syntax

`include "filename";`

Parameters

filename	A path 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 parts, each in a separate file. For instance, there might be parts that you need to change often and parts that you seldom edit.

Normally, the linker searches for configuration include files in the system configuration directory. You can use the `--config_search` linker option to add more directories to search.

See also

--*config_search*, page 334

Section reference

- Summary of sections and blocks
- Descriptions of sections and blocks

For more information, see *Modules and sections*, page 96.

Summary of sections and blocks

This table lists the ELF 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.
`.data`	Holds static and global initialized variables.
`.data_init`	Holds initial values for `.data` sections when the linker directive `initialize` is used.
`.exc.text`	Holds exception-related code.
`HEAP`	Holds the heap used for dynamically allocated data.
`__iar_tls$$DATA`	Holds the TLS area for the primary thread.
`__iar_tls$$INITDATA`	Holds initial values for the TLS area.
`.iar.dynexit`	Holds the `atexit` table.
`.iar.locale_table`	Holds the locale table for the selected locales.
`.init_array`	Holds a table of dynamic initialization functions.
`.intvec`	Holds the reset vector table
`IRQ_STACK`	Holds the stack for interrupt requests, IRQ, and exceptions.
`.noinit`	Holds `__no_init` static and global variables.
`.preinit_array`	Holds a table of dynamic initialization functions.
`.prepreinit_array`	Holds a table of dynamic initialization functions.
`.rodata`	Holds constant data.
`.tbss`	Holds thread-local zero-initialized static and global variables for the primary thread.
`.tdata`	Holds thread-local initialized static and global variables for the primary thread.

Table 42: Section summary

Section	Description
.text	Holds the program code.
.textrw	Holds __ramfunc declared program code.
.textrw_init	Holds initializers for the .textrw declared section.
Veneer$$CMSE	Holds secure gateway veneers.

Table 42: Section summary (Continued)

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 *Placing code and data—the linker configuration file*, page 99.

.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 stack memory*, page 118.

.data

Description
: Holds static and global initialized variables. In object files, this includes the initial values. When the linker directive `initialize` is used, a corresponding `.data_init` section is created for each `.data` section, holding the possibly compressed initial values.

Memory placement
: This section can be placed anywhere in memory.

.data_init

Description
: Holds the possibly compressed initial values for `.data` sections. This section is created by the linker if the `initialize` linker directive is used.

Memory placement
: This section can be placed anywhere in memory.

.exc.text

Description
: Holds code that is only executed when your application handles an exception.

Memory placement
: In the same memory as `.text`.

See also
: *Exception handling*, page 209.

HEAP

Description
: Holds the heap used for dynamically allocated data in memory, 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 heap memory*, page 118.

__iar_tls$$DATA

Description

Holds the two sections .tdata and .tbss. Together they hold the thread-local storage area for the primary thread. The main use for this section is to use a size operator (__section_size, see *Dedicated section operators*, page 202) on it to obtain the size of the thread-local storage area. You can also use the operator __iar_tls$$DATA$$Align to obtain the alignment of the thread-local storage.

This section is created by the linker if the linker option --threaded_lib is used.

See also

Managing a multithreaded environment, page 165

__iar_tls$$INITDATA

Description

Holds initial values for the thread-local storage area. The main use for this section is to copy it to a thread's thread-local storage area when the thread is created. The difference between the size of this section and the total size of the thread-local storage area (the section __iar_tls$$DATA) is the number of bytes in the thread-local storage area that should be initialized to zero.

This section is created by the linker if the linker option --threaded_lib is used.

See also

Managing a multithreaded environment, page 165

.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 118.

.iar.locale_table

Description

Holds the locale table for the selected locales.

Memory placement

This section can be placed anywhere in memory.

See also

Locale, page 164.

.init_array

Description

Holds pointers to routines to call for initializing one or more C++ objects with static storage duration.

Memory placement

This section can be placed anywhere in memory.

.intvec

Description

Holds the reset vector table and exception vectors which contain branch instructions to `cstartup`, interrupt service routines etc.

Memory placement

The placement of this section is device-dependent. See the manufacturer's hardware manual.

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 stack, page 218.

.noinit

Description

Holds static and global `__no_init` variables.

Memory placement

This section can be placed anywhere in memory.

.preinit_array

Description

Like `.init_array`, but is used by the library to make some C++ initializations happen before the others.

Memory placement

This section can be placed anywhere in memory.

See also

.init_array, page 541.

.prepreinit_array

Description
Like `.init_array`, but is used when C static initialization is rewritten as dynamic initialization. Performed before all C++ dynamic initialization.

Memory placement
This section can be placed anywhere in memory.

See also
.init_array, page 541.

.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.

.tbss

Description
Holds thread-local zero-initialized static and global variables for the primary thread. If the linker option `--threaded_lib` is not used, these are treated as `.bss`.

See also
Managing a multithreaded environment, page 165.

.tdata

Description
Holds thread-local initialized static and global variables for the primary thread. If the linker option `--threaded_lib` is not used, these are treated as `.data`.

See also
Managing a multithreaded environment, page 165.

.text

Description
Holds program code, including the code for system initialization.

Memory placement
This section can be placed anywhere in memory.

.textrw

Description
Holds `__ramfunc` declared program code.

Memory placement — This section can be placed anywhere in memory.

See also — *__ramfunc*, page 395.

.textrw_init

Description — Holds initializers for the `.textrw` declared sections.

Memory placement — This section can be placed anywhere in memory.

See also — *__ramfunc*, page 395.

Veneer$$CMSE

Description — This section contains secure gateway veneers created automatically by the linker for each entry function, as determined by the extended keyword `__cmse_nonsecure_entry`.

Memory placement — This section should be placed in an NSC (non-secure callable) memory region. NSC regions can be programmed using an SAU (security attribution unit) or an IDAU (implementation-defined attribute unit). For information about how to program the SAU or IDAU, see the documentation for your Armv8-M core.

See also — *Arm TrustZone®*, page 234, *--cmse*, page 284, *__cmse_nonsecure_entry*, page 389, and *--import_cmse_lib_out*, page 347

The stack usage control file

- Overview
- Stack usage control directives
- Syntactic components

Before you read this chapter, see *Stack usage analysis*, page 105.

Overview

A stack usage control file consists of a sequence of directives that control stack usage analysis. You can use C ("/*...*/") and C++ ("//...") comments in these files.

The default filename extension for stack usage control files is suc.

C++ NAMES

When you specify the name of a C++ function in a stack usage control file, you must use the name exactly as used by the linker. Both the number and names of parameters, as well as the names of types must match. However, most non-significant white-space differences are accepted. In particular, you must enclose the name in quote marks because all C++ function names include non-identifier characters.

You can also use wildcards in function names. "#*" matches any sequence of characters, and "#?" matches a single character. This makes it possible to write function names that will match any instantiation of a template function.

Examples:

```
"operator new(unsigned int)"
"std::ostream::flush()"
"operator <<(std::ostream &, char const *)"
"void _Sort<#*>(#*, #*, #*)"
```

Stack usage control directives

This section gives detailed reference information about each stack usage control directive.

call graph root directive

Syntax
```
call graph root [ category ] : func-spec [, func-spec... ];
```

Parameters

See the information on syntactic components:

category, page 549

func-spec, page 549

Description

Specifies that the listed functions are call graph roots. You can optionally specify a call graph root category. Call graph roots are listed under their category in the *Stack Usage* chapter in the linker map file.

The linker will normally issue a warning for functions needed in the application that are not call graph roots and which do not appear to be called.

Example
```
call graph root [task]: MyFunc10, MyFunc11;
```

See also

call_graph_root, page 406.

exclude directive

Syntax
```
exclude func-spec [, func-spec... ];
```

Parameters

See the information on syntactic components:

func-spec, page 549

Description

Excludes the specified functions, and call trees originating with them, from stack usage calculations.

Example
```
exclude MyFunc5, MyFunc6;
```

function directive

Syntax
```
[ override ] function [ category ] func-spec : stack-size
[ , call-info... ];
```

Parameters

See the information on syntactic components:

category, page 549

func-spec, page 549

call-info, page 550

stack-size, page 550

Description

Specifies what the maximum stack usage is in a function and which other functions that are called from that function.

Normally, an error is issued if there already is stack usage information for the function, but if you start with `override`, the error will be suppressed and the information supplied in the directive will be used instead of the previous information.

Example

```
function MyFunc1: 32,
  calls MyFunc2,
  calls MyFunc3, MyFunc4: 16;

function [interrupt] MyInterruptHandler: 44;
```

max recursion depth directive

Syntax

`max recursion depth` *`func-spec`* `:` *`size`*`;`

Parameters

See the information on syntactic components:

func-spec, page 549

size, page 551

Description

Specifies the maximum number of iterations through any of the cycles in the recursion nest of which the function is a member.

A recursion nest is a set of cycles in the call graph where each cycle shares at least one node with another cycle in the nest.

Stack usage analysis will base its result on the max recursion depth multiplied by the stack usage of the deepest cycle in the nest. If the nest is not entered on a point along one of the deepest cycles, no stack usage result will be calculated for such calls.

Example

```
max recursion depth MyFunc12: 10;
```

no calls from directive

Syntax `no calls from module-spec to func-spec [, func-spec... ];`

Parameters See the information on syntactic components:

func-spec, page 549

module-spec, page 549

Description When you provide stack usage information for some functions in a module without stack usage information, the linker warns about functions that are referenced from the module but not listed as called. This is primarily to help avoid problems with C runtime routines, calls to which are generated by the compiler, beyond user control.

If there actually is no call to some of these functions, use the `no calls from` directive to selectively suppress the warning for the specified functions. You can also disable the warning entirely (`--diag_suppress` or **Project>Options>Linker>Diagnostics>Suppress these diagnostics**).

Example

```
no calls from [file.o] to MyFunc13, MyFun14;
```

possible calls directive

Syntax `possible calls calling-func : called-func [ , called-func... ];`

Parameters See the information on syntactic components:

func-spec, page 549

Description Specifies an exhaustive list of possible destinations for all indirect calls in one function. Use this for functions which are known to perform indirect calls and where you know exactly which functions that might be called in this particular application. Consider using the `#pragma calls` directive if the information about which functions that might be called is available when compiling.

Example

```
possible calls MyFunc7: MyFunc8, MyFunc9;
```

When the function does not perform any calls, the list is empty:

```
possible calls MyFunc8: ;
```

See also *calls*, page 405.

Syntactic components

This section describes the syntactical components that can be used by the stack usage control directives.

category

Syntax
`[ name ]`

Description
A call graph root category. You can use any name you like. Categories are not case-sensitive.

Example
category examples:

```
[interrupt]
[task]
```

func-spec

Syntax
[*?]* *name* **[** *module-spec* **]**

Description
Specifies the name of a symbol, and for module-local symbols, the name of the module it is defined in. Normally, if `func-spec` does not match a symbol in the program, a warning is emitted. Prefixing with `?` suppresses this warning.

Example
func-spec examples:

```
xFun
MyFun [file.o]
?"fun1(int)"
```

module-spec

Syntax
`[name` **[** `(`*name*`)` **]**`]`

Description
Specifies the name of a module, and optionally, in parentheses, the name of the library it belongs to. To distinguish between modules with the same name, you can specify:

- The complete path of the file ("`D:\C1\test\file.o`")
- As many path elements as are needed at the end of the path ("`test\file.o`")
- Some path elements at the start of the path, followed by `"..."`, followed by some path elements at the end ("`D:\...\file.o`").

Note: When using multi-file compilation (`--mfc`), multiple files are compiled into a single module, named after the first file.

Example

module-spec examples:

```
[file.o]
[file.o(lib.a)]
["D:\C1\test\file.o"]
```

name

Description

A name can be either an identifier or a quoted string.

The first character of an identifier must be either a letter or one of the characters "_", "$", or ".". The rest of the characters can also be digits.

A quoted string starts and ends with " and can contain any character. Two consecutive " characters can be used inside a quoted string to represent a single ".

Example

name examples:

```
MyFun
file.o
"file-1.o"
```

call-info

Syntax

calls *func-spec* **[** , *func-spec*... **][** : *stack-size* **]**

Description

Specifies one or more called functions, and optionally, the stack size at the calls.

Example

call-info examples:

```
calls MyFunc1 : stack 16
calls MyFunc2, MyFunc3, MyFunc4
```

stack-size

Syntax

[stack **]** *size*
(**[** stack **]** size)

Description

Specifies the size of a stack frame. A stack may not be specified more than once.

Example

stack-size examples:

```
24
stack 28
```

size

Description

A decimal integer, or `0x` followed by a hexadecimal integer. Either alternative can optionally be followed by a suffix indicating a power of two (`K`=2^{10}, `M`=2^{20}, `G`=2^{30}, `T`=2^{40}, `P`=2^{50}).

Example

size examples:

```
24
0x18
2048
2K
```

IAR utilities

- 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—ielfdump—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 ELF Relocatable Object Creator—iexe2obj—creates a relocatable ELF object file from an executable ELF object file.
- Descriptions of options—detailed reference information about each command line option available for the different utilities.

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 `iarchive` 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 *IDE Project Management and Building Guide for Arm.*

INVOCATION SYNTAX

The invocation syntax for the archive builder is:

```
iarchive [command] [libraryfile] [objectfiles] [options]
```

Parameters

The parameters are:

Parameter	Description
command	A command line option that defines the operation to be performed. If the *command* is omitted, `--create` is used by default. You can specify the *command* anywhere on the command line.
libraryfile	The library file to be operated on. If specified like this, it must appear before the first object file, if any. You can also specify the library file using the option `-o`.
objectfiles	One or more object files as arguments to the command. Note that some commands take no object file arguments.
options	Optional command line options that modify the behavior of the archive tool. These options can be placed anywhere on the command line.

Table 43: 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.

Table 44: iarchive commands summary

Command line option	Description
`--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 44: iarchive commands summary (Continued)

For more information, see *Descriptions of options*, page 572.

SUMMARY OF IARCHIVE OPTIONS

This table summarizes the `iarchive` command line options:

Command line option	Description
`-f`	Extends the command line.
`--f`	Extends the command line, optionally with a dependency.
`--fake_time`	Generates library files with identical timestamps.
`--no_bom`	Omits the byte order mark from UTF-8 output files.
`--output, -o`	Specifies the library file.
`--text_out`	Specifies the encoding for text output files.
`--utf8_text_in`	Uses the UTF-8 encoding for text input files.
`--verbose, -V`	Reports all performed operations.
`--version`	Sends tool output to the console and then exits.
`--vtoc`	Produces a verbose list of files in the library.

Table 45: iarchive options summary

For more information, see *Descriptions of options*, page 572.

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 Visual Studio 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 558.
outputfile	An absolute ELF executable image, or if one of the relevant command line options is specified, an image file in another format.

Table 46: ielftool parameters

See also *Rules for specifying a filename or directory as parameters*, page 274.

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 raw binary.
`--bin-multi`	Produces output to multiple raw binary files.
`--checksum`	Generates a checksum.
`--fill`	Specifies fill requirements.
`--front_headers`	Outputs headers in the beginning of the file.
`--ihex`	Sets the format of the output file to 32-bit linear Intel Extended hex.
`--ihex-len`	Sets the number of data bytes in Intel Hex records.
`--offset`	Adds (or subtracts) an offset to all addresses in the generated output file.
`--parity`	Generates parity bits.
`--self_reloc`	Not for general use.
`--silent`	Sets silent operation.
`--simple`	Sets the format of the output file to Simple-code.
`--simple-ne`	As `--simple`, but without an entry record.
`--srec`	Sets the format of the output file to Motorola S-records.
`--srec-len`	Sets 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.
`--titxt`	Sets the format of the output file to Texas Instruments TI-TXT.
`--verbose, -V`	Prints all performed operations.
`--version`	Sends tool output to the console and then exits.

Table 47: ielftool options summary

For more information, see *Descriptions of options*, page 572.

SPECIFYING IELFTOOL ADDRESS RANGES

At the most basic level, an address range for `ielftool` consists of two hexadecimal numbers—`0x8000-0x87FF`—which includes both `0x8000` and `0x87FF`.

You can specify ELF symbols that are present in the processed ELF file as a start or end address using `__checksum_begin-__checksum_end`. This range begins on the byte that has the address value of the `__checksum_begin` symbol and ends (inclusive) on the byte that has the address value of the `__checksum_end` symbol. Symbol values of `0x40` and `0x3FD` would equate to specifying `0x40-0x3FD`.

You can add offsets to symbolic values using `__start+3-__end+0x10`. The calculation is done in modulo 32-bits, therefore adding `0xFFFFFFFF` is equivalent to subtracting 1.

You can specify blocks from an `.icf` file that are present in the processed ELF file using `{BLOCKNAME}`. A block started on `0x400` and ending (inclusively) on `0x535`, would equate to specifying `0x400-0x535`.

You can combine several address ranges, as long as they do not overlap, separated by `0x800-1FFF {FARCODE_BLOCK}`.

You can specify `__FLASH_BASE-__FLASH_END` as a legal range (as long as there is no overlap).

The IAR ELF Dumper—ielfdump

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 input_file [output_file]
```

Note: `ielfdumparm` is a command line tool which is not primarily intended to be used in the IDE.

Parameters

The parameters are:

Parameter	Description
input_file	An ELF relocatable or executable file to use as input.
output_file	A file or directory where the output is emitted. If absent and no `--output` option is specified, output is directed to the console.

Table 48: ielfdumparm parameters

See also *Rules for specifying a filename or directory as parameters*, page 274.

SUMMARY OF IELFDUMP OPTIONS

This table summarizes the `ielfdumparm` command line options:

Command line option	Description
`-a`	Generates output for all sections except string table sections.
`--all`	Generates output for all input sections regardless of their names or numbers.
`--code`	Dumps all sections that contain executable code.
`--disasm_data`	Dumps data sections as code sections.
`-f`	Extends the command line.
`--f`	Extends the command line, optionally with a dependency.
`--no_bom`	Omits the Byte Order Mark from UTF-8 output files.
`--no_header`	Suppresses production of a list header in the output.
`--no_rel_section`	Suppresses dumping of `.rel`/`.rela` sections.
`--no_strtab`	Suppresses dumping of string table sections.
`--no_utf8_in`	Do not assume UTF-8 for non-IAR ELF files.
`--output, -o`	Specifies an output file.
`--range`	Disassembles only addresses in the specified range.
`--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.
`--segment, -g`	Generates output for segments with specified numbers.

Table 49: ielfdumparm options summary

Command line option	Description
`--source`	Includes source with disassembled code in executable files.
`--text_out`	Specifies the encoding for text output files.
`--use_full_std_template_names`	Uses full short full names for some Standard C++ templates.
`--utf8_text_in`	Uses the UTF-8 encoding for text input files.
`--version`	Sends tool output to the console and then exits.

Table 49: ielfdumparm options summary (Continued)

For more information, see *Descriptions of options*, page 572.

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 50: iobjmanip parameters

See also *Rules for specifying a filename or directory as parameters*, page 274.

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.
`--f`	Extends the command line, optionally with a dependency.
`--no_bom`	Omits the Byte Order Mark from UTF-8 output files.
`--remove_file_path`	Removes path information from the file symbol.
`--remove_section`	Removes one or more section.
`--rename_section`	Renames a section.
`--rename_symbol`	Renames a symbol.
`--strip`	Removes debug information.
`--text_out`	Specifies the encoding for text output files.
`--utf8_text_in`	Uses the UTF-8 encoding for text input files.
`--version`	Sends tool output to the console and then exits.

Table 51: iobjmanip options summary

For more information, see *Descriptions of options*, page 572.

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 *nr* parameters but got *nr*

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 *name* removed due to transitive dependency on *name*

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.

To keep symbols from your symbols file in your final application, the symbols must be referred to, either from your source code or by using the linker option `--keep`.

INVOCATION SYNTAX

The invocation syntax for the IAR Absolute Symbol Exporter is:

```
isymexport [options] inputfile outputfile
```

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 565.
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 for controlling which symbols are included, and if desired, for also renaming some of the symbols.

Table 52: isymexport parameters

See also *Rules for specifying a filename or directory as parameters*, page 274.

In the IDE, to add the export of library symbols, choose **Project>Options>Build Actions** and specify your command line in the **Post-build command line** text field, for example:

```
$TOOLKIT_DIR$\bin\isymexport.exe "$TARGET_PATH$"
"$PROJ_DIR$\const_lib.symbols"
```

SUMMARY OF ISYMEXPORT OPTIONS

This table summarizes the `isymexport` command line options:

Command line option	Description
`--edit`	Specifies a steering file.
`--export_locals`	Exports local symbols.
`-f`	Extends the command line.
`--f`	Extends the command line, optionally with a dependency.
`--generate_vfe_header`	Declares that the image does not contain any virtual function calls to potentially discarded functions.
`--no_bom`	Omits the Byte Order Mark from UTF-8 output files.
`--ram_reserve_ranges`	Generates symbols for the areas in RAM that the image uses.
`--reserve_ranges`	Generates symbols to reserve the areas in ROM and RAM that the image uses.
`--show_entry_as`	Exports the entry point of the application with the given name.
`--text_out`	Specifies the encoding for text output files.
`--utf8_text_in`	Uses the UTF-8 encoding for text input files.
`--version`	Sends tool output to the console and then exits.

Table 53: isymexport options summary

For more information, see *Descriptions of options*, page 572.

STEERING FILES

A steering file can be used for controlling which symbols are included, and if desired, for also renaming some of the symbols. 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.

When you use a steering file, only actively exported symbols will be available in the output file. Therefore, a steering file without `show` directives will generate an output file without symbols.

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 */
```

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 as 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. Therefore, 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_* as Y_*_X
```

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
```

Show-root directive

Syntax

`show-root` *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, marked as `root`, unless this is overridden by a later `hide` directive.

When linking with the module produced by `isymexport`, the symbol will be included in the final executable file, even if no references to the symbol are present in the build.

Example

```
/* Export myVar making sure that it is included when linking */
show-root myVar
```

Show-weak directive

Syntax

```
show-weak 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 as a weak symbol unless this is overridden by a later `hide` directive.

When linking, no error will be reported if the new code contains a definition for a symbol with the same name as the exported symbol.

Note: Any internal references in the `isymexport` input file are already resolved and cannot be affected by the presence of definitions in the new code.

Example

```
/* Export myFunc as a weak definition */
show-weak myFunc
```

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.

Es015: ambiguous pattern match: *symbol* matches more than one rename pattern

A symbol in the input file matches more than one `rename` pattern.

Es016: the entry point symbol is already exported

The option `--show_entry_as` was used with a name that already exists in the input file.

The IAR ELF Relocatable Object Creator—iexe2obj

The IAR ELF Relocatable Object Creator, `iexe2obj`, creates a relocatable ELF object file from an executable ELF object file.

INVOCATION SYNTAX

The invocation syntax for `iexe2obj` is:

```
iexe2obj 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 option must be specified. See *Summary of iexe2obj options*, page 571
inputfile	An executable ELF object file.
outputfile	The name of the resulting relocatable ELF object file with all the requested operations applied.

Table 54: iexe2obj parameters

See also *Rules for specifying a filename or directory as parameters*, page 274.

BUILDING THE INPUT FILE

The input file must be linked with the linker option `--no_entry`, using object files compiled with `--rwpi`, `--ropi`, and `--ropi_cb`. See also *--ropi_cb*, page 317.

A function symbol `FUNC`, that should have a wrapper, must be preserved by the linker when it builds the input file. You can achieve this either by using the keyword `__root` in the declaration of `FUNC` or by using the linker command line option `--keep FUNC`.

Code and constant data

The input file can contain at most one *non-writable*, *executable* section that will be placed in the output file. To enable placing the executable section in execute-only memory, you must use the option `--no_literal_pool` both when compiling and when linking.

The input file can contain at most one *non-writable*, *non-executable* section that will be placed in the output file. The start address of the section will be used as a constants base address, `CB`.

Writable data

The input file can contain at most one *writable*, *non-executable* section that will be placed in the output file. The start address of the section will be used as a static base address, `SB`.

The writable data section might need dynamic initialization, in which case `iexe2obj` will create a function (`__sti_routine`) that is called during dynamic initialization of the client application. For this to work, a label `__init` is needed (as defined in the library `rt7MQx_tl`), and the linker configuration file used for creating your input file must contain:

```
define block INIT with alignment=4,fixed order {
  section .init_start,
  section .init_a,
  section .init_b,
  section .init_end.
};
```

The linker might issue a warning (Lp005) for mixing sections with content and sections without content. If that warning concerns sections `.data` and `.bss`, it can be ignored.

SUMMARY OF IEXE2OBJ OPTIONS

Command line option	Description
`--hide_symbols`	Hides all symbols from the input file

Table 55: iexe2obj options summary

Command line option	Description
`--keep_mode_symbols`	Copies mode symbols from the input file to the output file
`--prefix`	Sets a prefix for symbol and section names
`--wrap`	Generates wrapper functions for function symbols in *`inputfile`* that should be callable by clients of *`outputfile`*.

Table 55: iexe2obj options summary (Continued)

Descriptions of options

This section gives detailed reference information about each command line option available for the different utilities.

-a

Syntax `-a`

For use with `ielfdumparm`

Description Use this option as a shortcut for `--all --no_strtab`.

This option is not available in the IDE.

--all

Syntax `--all`

For use with `ielfdumparm`

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.

This option is not available in the IDE.

--bin

Syntax
`--bin[=`*`range`*`]`

Parameters
See *Specifying ielftool address ranges*, page 559.

For use with
`ielftool`

Description
Sets the format of the output file to raw binary, a binary format that includes only the raw bytes, with no address information. If no range is specified, the output file will include all the bytes from the lowest address for which there is content in the ELF file to the highest address for which there is content. If a range is specified, only bytes from that range are included. Note that in both cases, any gaps for which there is no content will be generated as zeros.

Note: If a range with no content is specified, no output file is created.

To set related options, choose:

Project>Options>Output converter

--bin-multi

Syntax
`--bin-multi[=`*`range`*`[;`*`range`*`...]]`

Parameters
See *Specifying ielftool address ranges*, page 559.

For use with
`ielftool`

Description
Use this option to produce one or more raw binary output files. If no ranges are specified, a raw binary output file is generated for each range for which there is content in the ELF file. If ranges are specified, a raw binary output file is generated for each range specified for which there is content. In each case, the name of each output file will include the start address of its range. For example, if the output file is specified as `out.bin` and the ranges `0x0-0x1F` and `0x8000-0x8147` are output, there will be two files, named `out-0x0.bin` and `out-0x8000.bin`.

This option is not available in the IDE.

--checksum

Syntax

```
--checksum {symbol[{+|-}offset]|address}:size,
algorithm[:[1|2][a|m|z][W|L|Q][x][r][R][o][i|p]]
[,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 The offset will be added (or subtracted if a negative offset (`-`) is specified) to the symbol. Address expressions using `+` and `-` are supported in a limited fashion. For example: `(start+7)-(end-2)`.

address The absolute address where the checksum value should be stored.

size The number of bytes in the checksum—`1`, `2`, or `4`. The number cannot be larger than the size of the checksum symbol.

algorithm The checksum algorithm used. Choose between:

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 `0x1021`); used by default.

crc32, CRC32 (generating polynomial `0x04C11DB7`).

crc64iso, CRC64iso (generating polynomial `0x1B`).

crc64ecma, CRC64ECMA (generating polynomial `0x42F0E1EBA9EA3693`).

crc=n, CRC with a generating polynomial of *n*.

`1|2` If specified, choose between:

`1`, specifies one's complement.

`2`, specifies two's complement.

`a|m|z` Reverses the order of the bits for the checksum. Choose between:

`a`, reverses the input bytes (but nothing else).

`m`, reverses the input bytes and the final checksum.

`z`, reverses the final checksum (but nothing else).

Note that using `a` and `z` in combination has the same effect as `m`.

`W|L|Q` Specifies the size of the unit for which a checksum should be calculated. Choose between:

`W`, calculates a checksum on 16 bits in every iteration.

`L`, calculates a checksum on 32 bits in every iteration.

`Q`, calculates a checksum on 64 bits in every iteration.

If you do not specify a unit size, 8 bits will be used by default.

The input byte sequence will processed as:

- 8-bit checksum unit size—byte0, byte1, byte2, byte3, etc.
- 16-bit checksum unit size—byte1, byte0, byte3, byte2, etc.
- 32-bit checksum unit size—byte3, byte2, byte1, byte0, byte7, byte6, byte5, byte4, etc.
- 64-bit checksum unit size—byte7, byte6, byte5, byte4, byte3, byte2, byte1, byte0, byte15, byte14, etc.

Note: The checksum unit size only affects the order in which the input byte sequence is processed. It does not affect the size of the checksum symbol, the polynomial, the initial value, the width of the processor's address bus, etc.

Most software CRC implementations use a checksum unit size of 1 byte (8 bits). The `W`, `L`, and `Q` parameters are almost exclusively used when a software CRC implementation has to match the checksum computed by the hardware CRC implementation. If you are not trying to cooperate with a hardware CRC implementation, the `W`, `L`, or `Q` parameter will simply compute a different checksum, because it processes the input byte sequence in a different order.

`x` Reverses the byte order of the checksum. This only affects the checksum value.

`r` Reverses the byte order of the input data. This has no effect unless the number of bits per iteration has been set using the `L` or `W` parameters.

`R`	Traverses the checksum range(s) in reverse order. If the range is, for example, `0x100-0xFFF;0x2000-0x2FFF`, the checksum calculation will normally start on `0x100` and then calculate every byte up to and including `0xFFF`, followed by calculating the byte on `0x2000` and continue to `0x2FFF`. Using the `R` parameter, the calculation instead starts on `0x2FFF` and continues by calculating every byte down to `0x2000`, then from `0xFFF` down to and including `0x100`.
`o`	Outputs the Rocksoft model specification for the checksum.
`i|p`	Use either `i` or `p`, if the *`start`* value is bigger than `0`. Choose between: `i`, initializes the checksum value with the start value. `p`, prefixes the input data with a word of size *`size`* that contains the *`start`* value.
`start`	By default, the initial value of the checksum is `0`. If necessary, use *`start`* to supply a different initial value. If not `0`, then either `i` or `p` must be specified.
`range`	*`range`* is one or more memory ranges for which the checksum will be calculated. It is typically advisable to use symbols or blocks if the memory range can change. If you use explicit addresses, for example, `0x8000-0x8347`, and the code then changes, you need to update the end address to the new value. If you instead use `{CODE}` or a symbol located at the end of the code, you do not need to update the `--checksum` command. See also *Specifying ielftool address ranges*, page 559.

For use with

`ielftool`

Description

Use this option to calculate a checksum with the specified algorithm for the specified ranges. If you have an external definition for the checksum—for example, a hardware CRC implementation—use the appropriate parameters to the `--checksum` option to match the external design. In this case, learn more about that design in the hardware documentation. 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.

Example

This example shows how to use the crc16 algorithm with the start value `0` over the address range `0x8000-0x8FFF`:

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

The input data i read from `sourceFile.out`, and the resulting checksum value of size 2 bytes will be stored at the symbol `__checksum`. The modified ELF file is saved as `destinationFile.out` leaving `sourceFile.out` untouched.

In the next example, a symbol is used for specifying the start of the range:

```
ielftool --checksum=__checksum:2,crc16;__checksum_begin-0x8FFF
sourceFile.out destinationFile.out
```

If `BLOCK1` occupies `0x4000-0x4337` and `BLOCK2` occupies `0x8000-0x87FF`, this example will compute the checksum for the bytes on `0x4000` to `0x4337` and from `0x8000` to `0x87FF`:

```
ielftool --checksum __checksum:2,crc16;{BLOCK1};{BLOCK2}
BlxTest.out BlxTest2.out
```

See also

Checksum calculation for verifying image integrity, page 222

Specifying ielftool address ranges, page 559

To set related options, choose:

Project>Options>Linker>Checksum

--code

Syntax

`--code`

For use with

`ielfdumparm`

Description

Use this option to dump all sections that contain executable code—sections with the ELF section attribute `SHF_EXECINSTR`.

This option is not available in the IDE.

--create

Syntax

`--create` *libraryfile objectfile1 ... objectfileN*

Parameters

libraryfile	The library file that the command operates on.
objectfile1 ... objectfileN	The object file(s) to build the library from. The arguments can also be archive files, in which case each member in the archive file is processed as if specified separately.

See also *Rules for specifying a filename or directory as parameters*, page 274

For use with

`iarchive`

Description

Use this command to build a new library from a set of object files (modules) and/or archive files. The modules are added to the library in the order that they are specified on the command line.

If no command is specified on the command line, `--create` is used by default.

This option is not available in the IDE.

--delete, -d

Syntax

`--delete` *libraryfile objectfile1 ... objectfileN*
`-d` *libraryfile objectfile1 ... objectfileN*

Parameters

libraryfile	The library file that the command operates on.
objectfile1 ... objectfileN	The object file(s) that the command operates on.

See also *Rules for specifying a filename or directory as parameters*, page 274

For use with

`iarchive`

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.

This option is not available in the IDE.

--disasm_data

Syntax
`--disasm_data`

For use with
`ielfdumparm`

Description
Use this command to instruct the dumper to dump data sections as if they were code sections.

This option is not available in the IDE.

--edit

Syntax
`--edit` *`steering_file`*

Parameters
See *Rules for specifying a filename or directory as parameters*, page 274

For use with
`isymexport`

Description
Use this option to specify a steering file for controlling which symbols are included in the `isymexport` output file, and if desired, also for renaming some of the symbols.

See also
Steering files, page 565.

This option is not available in the IDE.

--export_locals

Syntax
`--export_locals [=`*`symbol_prefix`*`]`

Parameters

`symbol_prefix`	A custom prefix to the names of exported symbols that replaces the default prefix `LOCAL`.

For use with
`isymexport`

Description
Use this option to export local symbols from a ROM image file, in addition to absolute symbols. The default name of the exported symbol is `LOCAL_`*`filename_symbolname`*. Use the optional parameter *`symbol_prefix`* to replace `LOCAL` with your custom prefix.

Example

When exported from the ROM image file, the symbol `symb` in the source file `myFile.c` becomes `LOCAL_myFile_c_symb`.

This option is not available in the IDE.

--extract, -x

Syntax

```
--extract libraryfile [objectfile1 ... objectfileN]
-x libraryfile [objectfile1 ... objectfileN]
```

Parameters

libraryfile	The library file that the command operates on.
objectfile1 ... objectfileN	The object file(s) that the command operates on.

See also *Rules for specifying a filename or directory as parameters*, page 274

For use with

`iarchive`

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.

This option is not available in the IDE.

-f

Syntax

```
-f filename
```

Parameters

See *Rules for specifying a filename or directory as parameters*, page 274

For use with

`iarchive`, `ielfdumparm`, `iobjmanip`, and `isymexport`.

Description

Use this option to make the tool 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.

This option is not available in the IDE.

--f

Syntax `--f filename`

Parameters See *Rules for specifying a filename or directory as parameters*, page 274

For use with `iarchive`, `ielfdumparm`, `iobjmanip`, and `isymexport`.

Description Use this option to make the tool 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.

If you also specify `--dependencies` on the command line for the tool, extended command line files specified using `--f` will generate a dependency, but those specified using `-f` will not generate a dependency.

See also *-f*, page 580.

This option is not available in the IDE.

--fake_time

Syntax `--fake_time`

For use with `iarchive`

Description Use this option to generate library files with identical timestamps. The value used is `0x5CF00000`, which corresponds to approximately 30th May 2019 at 18:08:32 (the exact time will vary depending on the time settings). This option enables you to generate identical libraries for identical object files. Without this option, the timestamp will generate unique library files from the same input files.

This option is not available in the IDE.

--fill

Syntax

`--fill [v;]`*`pattern`*`;`*`range`*`[;`*`range`*`...]`

Parameters

`v`	Generates virtual fill for the fill command. Virtual fill is filler bytes that are included in checksumming, but that are not included in the output file. The primary use for this is certain types of hardware where bytes that are not specified by the image have a known value—typically, `0xFF` or `0x0`.
`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.
`range`	Specifies the address range for the fill. Note that each address must be 4-byte aligned.

See also *Specifying ielftool address ranges*, page 559.

For use with

`ielftool`

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

--front_headers

Syntax `--front_headers`

For use with `ielftool`

Description Use this option to output ELF program and section headers in the beginning of the file, instead of at the end.

This option is not available in the IDE.

--generate_vfe_header

Syntax `--generate_vfe_header`

For use with `isymexport`

Description Use this option to declare that the image does not contain any virtual function calls to potentially discarded functions.

When the linker performs virtual function elimination, it discards virtual functions that appear not to be needed. For the optimization to be applied correctly, there must be no virtual function calls in the image that affect the functions that are discarded.

See also *Virtual function elimination*, page 127.

To set this options, use:

Project>Options>Linker>Extra Options

--hide_symbols

Syntax `--hide_symbols`

For use with `iexe2obj`

Description Use this option to hide all symbols from the input file.

This option is not available in the IDE.

--ihex

Syntax `--ihex`

For use with `ielftool`

Description

Sets the format of the output file to 32-bit linear Intel Extended hex, a hexadecimal text format defined by Intel.

Note: Intel Extended cannot express addresses larger than 2^{32}-1. If your application contains such addresses, you must use another format.

To set related options, choose:

Project>Options>Linker>Output converter

--ihex-len

Syntax `--ihex-len=`*`length`*

Parameters

`length`	The number of data bytes in the record.

For use with `ielftool`

Description

Sets the maximum number of data bytes in an Intel Hex record. This option can only be used together with the `--ihex` option. By default, the number of data bytes in an Intel Hex record is 16.

This option is not available in the IDE.

--keep_mode_symbols

Syntax `--keep_mode_symbols`

For use with `iexe2obj`

Description

Use this option to copy mode symbols from the input file to the output file. This is used, for example, by the disassembler.

This option is not available in the IDE.

--no_bom

Syntax `--no_bom`

For use with `iarchive`, `ielfdumparm`, `iobjmanip`, and `isymexport`

Description Use this option to omit the Byte Order Mark (BOM) when generating a UTF-8 output file.

See also --*text_out*, page 598 and *Text encodings*, page 266

This option is not available in the IDE.

--no_header

Syntax `--no_header`

For use with `ielfdumparm`

Description By default, a standard list header is added before the actual file content. Use this option to suppress output of the list header.

This option is not available in the IDE.

--no_rel_section

Syntax `--no_rel_section`

For use with `ielfdumparm`

Description By default, whenever the content of a section of a relocatable file is generated as output, the associated section, if any, is also included in the output. Use this option to suppress output of the relocation section.

This option is not available in the IDE.

--no_strtab

Syntax `--no_strtab`

For use with `ielfdumparm`

Description Use this option to suppress dumping of string table sections (sections of type `SHT_STRTAB`).

This option is not available in the IDE.

--no_utf8_in

Syntax `--no_utf8_in`

For use with `ielfdumparm`

Description The dumper can normally determine whether ELF files produced by IAR tools use the UTF-8 text encoding or not, and produce the correct output. For ELF files produced by non-IAR tools, the dumper will assume UTF-8 encoding unless this option is used, in which case the encoding is assumed to be according to the current system default locale.

Note: This only makes a difference if any characters beyond 7-bit ASCII are used in paths, symbols, etc.

See also *Text encodings*, page 266

This option is not available in the IDE.

--offset

Syntax `--offset [-]`*`offset`*

Parameters

`offset`	The offset will be added (or subtracted if - is specified) to all addresses in the generated output file.

For use with `ielftool`

Description Use this option to add or subtract an offset to the address of each output record in the generated output file. The option only works on Motorola S-records, Intel Hex, TI-Txt, and Simple-Code. The option has no effect when generating an ELF file or when binary files (`--bin` contain no address information) are generated. No content, including the entry point, will be changed by using this option, only the addresses in the output format.

Example

```
--offset 0x30000
```

This will add an offset of `0x30000` to all addresses. As a result, content that was linked at address `0x4000` will be placed at `0x34000`.

This option is not available in the IDE.

--output, -o

Syntax

```
-o {filename|directory}
--output {filename|directory}
```

Parameters

See *Rules for specifying a filename or directory as parameters*, page 274

For use with

`iarchive` and `ielfdumparm`.

Description

`iarchive`

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.

`ielfdumparm`

By default, output from the dumper is directed to the console. Use this option to direct the output to a file instead. The default name of the output file is the name of the input file with an added `id` filename extension

You can also specify the output file by specifying a file or directory following the name of the input file.

This option is not available in the IDE.

--parity

Syntax

```
--parity{symbol[+offset]|address}:size,algo:flashbase[:flags];range[;range...]
```

Parameters

symbol	The name of the symbol where the parity bytes should be stored. Note that it must exist in the symbol table in the input ELF file.
offset	An offset to the symbol. By default, `0`.
address	The absolute address where the parity bytes should be stored.
size	The maximum number of bytes that the parity generation can use. An error will be issued if this value is exceeded. Note that the size must fit in the specified symbol in the ELF file.
algo	Choose between: `odd`, uses odd parity. `even`, uses even parity.
flashbase	The start address of the flash memory. Parity bits will not be generated for the addresses between *`flashbase`* and the start address of the range. If *`flashbase`* and the start address of the range coincide, parity bits will be generated for all addresses
flags	Choose between: `r`, reverses the byte order within each word. `L`, processes 4 bytes at a time. `W`, processes 2 bytes at a time. `B`, processes 1 byte at a time.
range	The address range over which the parity bytes should be generated. See also *Specifying ielftool address ranges*, page 559.

For use with

`ielftool`

Description

Use this option to generate parity bytes over specified ranges. The range is traversed left to the right and the parity bits are generated using the odd or even algorithm. The parity bits are finally stored in the specified symbol where they can be accessed by your application.

This option is not available in the IDE.

--prefix

Syntax

`--prefix prefix`

Parameters

prefix	A prefix for symbol and section names

For use with

`iexe2obj`

Description

By default, the base name of the output file is used as a prefix for symbol and section names that are defined in wrappers. Use this option to set a custom prefix for these symbols and section names.

See also

--*wrap*, page 602

This option is not available in the IDE.

--ram_reserve_ranges

Syntax

`--ram_reserve_ranges[=symbol_prefix]`

Parameters

symbol_prefix	The prefix of symbols created by this option.

For use with

`isymexport`

Description

Use this option to generate symbols for the areas in RAM that the image uses. One symbol will be generated for each such area. The name of each symbol is based on the name of the area and is prefixed by the optional parameter *symbol_prefix*.

Generating symbols that cover an area in this way prevents the linker from placing other content at the affected addresses. This can be useful when linking against an existing image.

If `--ram_reserve_ranges` is used together with `--reserve_ranges`, the RAM areas will get their prefix from the `--ram_reserve_ranges` option and the non-RAM areas will get their prefix from the `--reserve_ranges` option.

See also

--*reserve_ranges*, page 593.

This option is not available in the IDE.

--range

Syntax

`--range start-end`

Parameters

start-end	Disassemble code where the start address is greater than or equal to *start*, and where the end address is less than *end*.

For use with

`ielfdumparm`

Description

Use this option to specify a range for which code from an executable will be dumped.

This option is not available in the IDE.

--raw

Syntax

`--raw`

For use with

`ielfdumparm`

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.

Note: Raw-binary does not have any problems with 64-bit addresses.

This option is not available in the IDE.

--remove_file_path

Syntax

`--remove_file_path`

For use with

`iobjmanip`

Description

Use this option to make `iobjmanip` remove information about the directory structure of the project source tree from the generated object file, which means that the file symbol in the ELF object file is modified.

This option must be used in combination with `--remove_section ".comment"`.

This option is not available in the IDE.

--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`.

For use with

`iobjmanip`

Description

Use this option to make `iobjmanip` omit the specified section when generating the output file.

This option is not available in the IDE.

--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.

For use with

`iobjmanip`

Description

Use this option to make `iobjmanip` rename the specified section when generating the output file.

This option is not available in the IDE.

--rename_symbol

Syntax

`--rename_symbol oldname =newname`

Parameters

oldname	The symbol to be renamed.
newname	The new name of the symbol.

For use with

`iobjmanip`

Description

Use this option to make `iobjmanip` rename the specified symbol when generating the output file.

This option is not available in the IDE.

--replace, -r

Syntax

```
--replace libraryfile objectfile1 ... objectfileN
-r libraryfile objectfile1 ... objectfileN
```

Parameters

libraryfile	The library file that the command operates on.
objectfile1 ... objectfileN	The object file(s) that the command operates on. The arguments can also be archive files, in which case each member in the archive file is processed as if specified separately.

See also *Rules for specifying a filename or directory as parameters*, page 274

For use with `iarchive`

Description

Use this command to replace or add object files (modules) and/or archive files to an existing library. The modules specified on the command line either replace existing modules in the library—if they have the same name—or are appended to the library.

This option is not available in the IDE.

--reserve_ranges

Syntax

`--reserve_ranges[=symbol_prefix]`

Parameters

symbol_prefix	The prefix of symbols created by this option.

For use with `isymexport`

Description

Use this option to generate symbols for the areas in ROM and RAM that the image uses. One symbol will be generated for each such area. The name of each symbol is based on the name of the area and is prefixed by the optional parameter *symbol_prefix*.

Generating symbols that cover an area in this way prevents the linker from placing other content at the affected addresses. This can be useful when linking against an existing image.

If `--reserve_ranges` is used together with `--ram_reserve_ranges`, the RAM areas will get their prefix from the `--ram_reserve_ranges` option and the non-RAM areas will get their prefix from the `--reserve_ranges` option.

See also

--*ram_reserve_ranges*, page 589.

This option is not available in the IDE.

--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.

For use with

ielfdumparm

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 */
```

This option is not available in the IDE.

--segment, -g

Syntax

--segment *segment_number*[,...]

-g *segment_number*[,...]

Parameters

segment_number The number of a segment whose contents will be included in the output.

For use with

ielfdumparm

Description

Use this option to select specific segments—parts of an executable image indicated by program headers—for inclusion in the output.

This option is not available in the IDE.

--self_reloc

Syntax `--self_reloc`

For use with `ielftool`

Description This option is intentionally not documented as it is not intended for general use.

This option is not available in the IDE.

--show_entry_as

Syntax `--show_entry_as` *`name`*

Parameters

`name`	The name to give to the program entry point in the output file.

For use with `isymexport`

Description Use this option to export the entry point of the application given as input under the name *`name`*.

This option is not available in the IDE.

--silent

Syntax `--silent`

For use with `ielftool`

Description Causes the tool to operate without sending any messages to the standard output stream.

By default, the tool sends various messages via the standard output stream. You can use this option to prevent this. The tool 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`

For use with `ielftool`

Description

Sets the format of the output file to Simple-code, a binary format that includes address information.

Note: Simple-code can express addresses larger than 2^{32}-1. If your application contains such addresses, a Simple-code file with a higher version number will be generated. Such files can only be read by Simple-code readers that can handle this higher version.

To set related options, choose:

Project>Options>Output converter

--simple-ne

Syntax `--simple-ne`

For use with `ielftool`

Description

Sets the format of the output file to Simple code, but no entry record is generated.

To set related options, choose:

Project>Options>Output converter

--source

Syntax `--source`

For use with `ielfdumparm`

Description

Use this option to make `ielftool` include source for each statement before the code for that statement, when dumping code from an executable file. To make this work, the executable image must be built with debug information, and the source code must still be accessible in its original location.

This option is not available in the IDE.

--srec

Syntax `--srec`

For use with `ielftool`

Description Sets the format of the output file to Motorola S-records, a hexadecimal text format defined by Motorola. Note that you can use the `ielftool` options `--srec-len` and `--srec-s3only` to modify the exact format used.

Note: Motorola S-records cannot express addresses larger than 2^{32}-1. If your application contains such addresses, you must use another format.

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.

For use with `ielftool`

Description Sets the maximum number of data bytes in an S-record. This option can only be used together with the `--srec` option. By default, the number of data bytes in an S-record is 16.

This option is not available in the IDE.

--srec-s3only

Syntax `--srec-s3only`

For use with `ielftool`

Description Restricts the S-record output to contain only a subset of records, that is S0, 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`

For use with
`iobjmanip` and `ielftool`.

Description
Use this option to remove all sections containing debug information before the output file is written. `iobjmanip` will also remove the names of all module-local function, variable, and section symbols.

Note: `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

--symbols

Syntax
`--symbols libraryfile`

Parameters

libraryfile	The library file that the command operates on.

See also *Rules for specifying a filename or directory as parameters*, page 274

For use with
`iarchive`

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.

This option is not available in the IDE.

--text_out

Syntax
`--text_out{utf8|utf16le|utf16be|locale}`

Parameters

`utf8`	Uses the UTF-8 encoding

`utf16le`	Uses the UTF-16 little-endian encoding
`utf16be`	Uses the UTF-16 big-endian encoding
`locale`	Uses the system locale encoding

For use with

`iarchive`, `ielfdumparm`, `iobjmanip`, and `isymexport`

Description

Use this option to specify the encoding to be used when generating a text output file.

The default for the list files is to use the same encoding as the main source file. The default for all other text files is UTF-8 with a Byte Order Mark (BOM).

If you want text output in UTF-8 encoding without BOM, you can use the option `--no_bom` as well.

See also

--no_bom, page 585 and *Text encodings*, page 266

This option is not available in the IDE.

--titxt

Syntax

```
--titxt
```

For use with

`ielftool`

Description

Sets the format of the output file to Texas Instruments TI–TXT, a hexadecimal text format defined by Texas Instruments.

Note: Texas Instruments TI–TXT can express addresses larger than 2^{32}-1.

To set related options, choose:

Project>Options>Output converter

--toc, -t

Syntax

```
--toc libraryfile
-t libraryfile
```

Parameters

`libraryfile`	The library file that the command operates on.

See also *Rules for specifying a filename or directory as parameters*, page 274

For use with	iarchive
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.

This option is not available in the IDE.

--use_full_std_template_names

Syntax	--use_full_std_template_names
For use with	ielfdumparm
Description	Normally, the names of some standard C++ templates are used in the output in an abbreviated form in the demangled names of symbols, for example, "std::string" instead of "std::basic_string<char, std::char_traits<char>, std_::allocator<char>>". Use this option to make ielfdump use the unabbreviated form.

This option is not available in the IDE.

--utf8_text_in

Syntax	--utf8_text_in
For use with	iarchive, ielfdumparm, iobjmanip, and isymexport
Description	Use this option to specify that the tool shall use the UTF-8 encoding when reading a text input file with no Byte Order Mark (BOM).

Note: This option does not apply to source files.

See also *Text encodings*, page 266

This option is not available in the IDE.

--verbose, -V

Syntax

```
--verbose
-V (iarchive only)
```

For use with

`iarchive` and `ielftool`.

Description

Use this option to make the tool 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.

--version

Syntax

```
--version
```

For use with

`iarchive`, `ielfdumparm`, `ielftool`, `iobjmanip`, `isymexport`

Description

Use this option to make the tool send version information to the console and then exit.

This option is not available in the IDE.

--vtoc

Syntax

`--vtoc` *`libraryfile`*

Parameters

`libraryfile`	The library file that the command operates on.

See also *Rules for specifying a filename or directory as parameters*, page 274

For use with

`iarchive`

Description

Use this command to list the names, sizes, and modification times 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.

This option is not available in the IDE.

--wrap

Syntax	`--wrap` *`symbol`*

Parameters

`symbol`	A function symbol that should be callable by clients of the output file of `iexe2obj`.

For use with	`iexe2obj`
Description	Use this option to generate a wrapper function for function symbols.

This option is not available in the IDE.

Implementation-defined behavior for Standard C++

- Descriptions of implementation-defined behavior for C++
- Implementation quantities

If you are using C instead of C++, see *Implementation-defined behavior for Standard C*, page 623 or *Implementation-defined behavior for C89*, page 643, respectively.

Descriptions of implementation-defined behavior for C++

This section follows the same order as the C++ 14 standard. Each item includes references to the ISO chapter and section (in parenthesis) that explains the implementation-defined behavior.

Note: The IAR Systems implementation adheres to a freestanding implementation of Standard C++ 14. This means that parts of a standard library can be excluded from the implementation.

1 GENERAL

Diagnostics (1.3.6)

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.

Required libraries for freestanding implementation (1.4)

See *C++ header files*, page 493 and *Not supported C/C++ functionality*, page 497, respectively, for information about which Standard C++ system headers that the IAR C/C++ Compiler does not support.

Bits in a byte (1.7)

A byte contains 8 bits.

Interactive devices (1.9)

The streams `stdin`, `stdout`, and `stderr` are treated as interactive devices.

Number of threads in a program under a freestanding implementation (1.10)

By default, the IAR Systems runtime environment does not support more than one thread of execution. With an optional third-party RTOS, it might support several threads of execution.

2 LEXICAL CONVENTIONS

Mapping physical source file characters to the basic source character set (2.2)

The source character set is the same as the physical source file multibyte character set. By default, the standard ASCII character set is used. However, it can be UTF-8, UTF-16, or the system locale. See *Text encodings*, page 266.

Physical source file characters (2.2)

The source character set is the same as the physical source file multibyte character set. By default, the standard ASCII character set is used. However, it can be UTF-8, UTF-16, or the system locale. See *Text encodings*, page 266.

Converting characters from a source character set to the execution character set (2.2)

The source character set is the set of legal characters that can appear in source files. It is dependent on the chosen encoding for the source file. See *Text encodings*, page 266. By default, the source character set is Raw.

The execution character set is the set of legal characters that can appear in the execution environment. These are the execution character sets for character constants and string literals, and their encoding types:

Execution character set	Encoding type
`L`	UTF-32
`u`	UTF-16
`U`	UTF-32

Table 56: Execution character sets and their encodings

Execution character set	Encoding type
`u8`	UTF-8
`none`	The source character set

Table 56: Execution character sets and their encodings (Continued)

The DLIB runtime environment needs a multibyte character scanner to support a multibyte execution character set. See *Locale*, page 164.

Required availability of the source of translation units to locate template definitions (2.2)

When locating the template definition related to template instantiations, the source of the translation units that define the template is not required.

The execution character set and execution wide-character set (2.3)

The values of the members of the execution character set are the values of the ASCII character set, which can be augmented by the values of the extra characters in the source file character set. The source file character set is determined by the chosen encoding for the source file. See *Text encodings*, page 266.

The wide character set consists of all the code points defined by ISO/IEC 10646.

Mapping header names to headers or external source files (2.9)

The header name is interpreted and mapped into an external source file in the most intuitive way. In both forms of the `#include` preprocessing directive, the character sequences that specify header names are interpreted exactly in the same way as for other source constructs. They are then mapped to external header source file names.

The value of multi-character literals (2.14.3)

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 is issued if the value cannot be represented in an integer constant.

The value of wide-character literals with single c-char that are not in the execution wide-character set (2.14.3)

All possible c-chars have a representation in the execution wide-character set.

The value of wide-character literal containing multiple characters (2.14.3)

A diagnostic message is issued, and all but the first c-char is ignored.

The semantics of non-standard escape sequences (2.14.3)

No non-standard escape sequences are supported.

The value of character literal outside range of corresponding type (2.14.3)

The value is truncated to fit the type.

The encoding of universal character name not in execution character set (2.14.3)

A diagnostic message is issued.

The choice of larger or smaller value of floating-point literal (2.14.4)

For a floating-point literal whose scaled value cannot be represented as a floating-point value, the nearest even floating point-value is chosen.

The distinctness of string literals (2.14.5)

All string literals are distinct except when the linker option `--merge_duplicate_sections` is used.

Concatenation of various types of string literals (2.14.5)

Differently prefixed string literal tokens cannot be concatenated, except for those specified by the ISO C++ standard.

3 BASIC CONCEPTS

Defining main in a freestanding environment (3.6.1)

The `main` function must be defined.

Startup and termination in a freestanding environment (3.6.1)

See *Application execution—an overview*, page 64 and *System startup and termination*, page 150, for descriptions of the startup and termination of applications.

Parameters to main (3.6.1)

The only two permitted definitions for `main` are:

```
int main()
int main(int, char **)
```

Linkage of main (3.6.1)

The `main` function has external linkage.

Dynamic initialization of static objects before main (3.6.2)

Static objects are initialized before the first statement of `main`, except when the linker option `--manual_dynamic_initialization` is used.

Dynamic initialization of threaded local objects before entry (3.6.2)

By default, the IAR systems runtime environment does not support more than one thread of execution. With an optional third-party RTOS, it might support several threads of execution.

Thread-local objects are treated as static objects except when the linker option `--threaded_lib` is used. Then they are initialized by the RTOS.

Use of an invalid pointer (3.7.4.2)

Any other use of an invalid pointer than indirection through it and passing it to a deallocation function works as for a valid pointer.

Relaxed or strict pointer safety for the implementation (3.7.4.3)

The IAR Systems implementation of Standard C++ has relaxed pointer safety.

The value of trivially copyable types (3.9)

All bits in basic types are part of the value representation. Padding between basic types is copied verbatim.

Representation and signage of char (3.9.1)

A plain `char` is treated as an `unsigned char`. See *--char_is_signed*, page 283 and *--char_is_unsigned*, page 283.

Extended signed integer types (3.9.1)

No extended signed integer types exist in the implementation.

Value representation of floating-point types (3.9.1)

See *Basic data types—floating-point types*, page 374.

Value representation of pointer types (3.9.2)

See *Pointer types*, page 376.

Alignment (3.11)

See *Alignment*, page 367.

Alignment additional values (3.11)

See *Alignment*, page 367.

alignof expression additional values (3.11)

See *Alignment*, page 367.

4 STANDARD CONVERSIONS

lvalue-to-rvalue conversion for objects that contain an invalid pointer (4.1)

The conversion is made as if the pointer was valid.

The value of the result of unsigned to signed conversion (4.7)

When an integer value is converted to a value of signed integer type, but cannot be represented by the destination type, the value is truncated to the number of bits of the destination type and then reinterpreted as a value of the destination type.

The result of inexact floating-point conversion (4.8)

When a floating-point value is converted to a value of a different floating-point type, and the value is within the range of the destination type but cannot be represented exactly, the value is rounded to the nearest floating-point value by default.

The value of the result of an inexact integer to floating-point conversion (4.9)

When an integer value is converted to a value of a floating-point type, and the value is within the range of the destination type but cannot be represented exactly, the value is rounded to the nearest floating-point value by default.

The rank of extended signed integer types (4.13)

The implementation has no extended signed integer types.

5 EXPRESSIONS

Passing argument of class type through ellipsis (5.2.2)

The result is a diagnostic and is then treated as a trivially copyable object.

The derived type for typeid (5.2.8)

The type of a `typeid` expression is an expression with dynamic type `std::type_info`.

Conversion from a pointer to an integer (5.2.10)

See *Casting*, page 377.

Conversion from an integer to a pointer (5.2.10)

See *Casting*, page 377.

Converting a function pointer to an object pointer and vice versa (5.2.10)

See *Casting*, page 377.

sizeof applied to fundamental types other than char, signed char, and unsigned char (5.3.3)

See *Basic data types—integer types*, page 369, *Basic data types—floating-point types*, page 374, and *Pointer types*, page 376.

Support for over-aligned types (5.3.4)

Over-aligned types are supported in `new` expressions.

The type of ptrdiff_t (5.7)

See *ptrdiff_t*, page 378.

The result of right shift of negative value (5.8)

In a bitwise right shift operation of the form `E1 >> E2`, if `E1` is of signed type and has a negative value, the value of the result is the integral part of the quotient `E1/(2**E2)`, except when `E1` is `-1`.

7 DECLARATIONS

The meaning of the attribute declaration (7)

There are no other attributes supported than what is specified in the C++ standard. See *Extended keywords*, page 383, for supported attributes and ways to use them with objects.

Access to an object that has volatile-qualified type (7.1.6.1)

See *Declaring objects volatile*, page 380.

The underlying type for enumeration (7.2)

See *The enum type*, page 370.

The meaning of the asm declaration (7.4)

An `asm` declaration enables the direct use of assembler instructions.

The semantics of linkage specifiers (7.5)

Only the string-literals `"C"` and `"C++"` can be used in a linkage specifier.

Linkage of objects to other languages than C (7.5)

The IAR Systems implementation of Standard C++ does not support linkage to other languages than C.

The behavior of attribute-scoped tokens (7.6.1)

The use of an attribute-scoped token is not supported.

The behavior of non-standard attributes (7.6.1)

There are no other attributes supported other than what is specified in the C++ standard. See *Extended keywords*, page 383, for a list supported attributes and ways to use them with objects.

8 DECLARATORS

The string resulting from __func__ (8.4.1)

The value of `__func__` is the C++ function name.

9 CLASSES

Allocation of bitfields within a class object (9.6)

See *Bitfields*, page 370.

14 TEMPLATES

The semantics of linkage specification on templates (14)

Only the string-literals `"C"` and `"C++"` can be used in a linkage specifier.

15 EXCEPTION HANDLING

Stack unwinding before calling std::terminate() (15.3, 15.5.1)

When no suitable catch handler is found, the stack is not unwound before calling `std::terminate()`.

Stack unwinding before calling std::terminate() when a noexcept specification is violated (15.5.1)

When a `noexcept` specification is violated, the stack is not unwound before calling `std::terminate()`.

Bad throw in std::unexpected (15.5.2)

If `std::unexpected` throws an exception that is not allowed by the exception specification for the function that caused the original exception specification violation, and that exception specification includes `std::bad_exception`, then the thrown exception is replaced by a `std::bad_exception` and the search for another handler continues.

16 PREPROCESSING DIRECTIVES

The numeric values of character literals in #if directives (16.1)

Numeric values of character literals in the `#if` and `#elif` preprocessing directives match the values that they have in other expressions.

Negative value of character literal in preprocessor (16.1)

A plain `char` is treated as an `unsigned char`. See *--char_is_signed*, page 283 and *--char_is_unsigned*, page 283. If a `char` is treated as a signed character, then character literals in `#if` and `#elif` preprocessing directives can be negative.

Search locations for < > header (16.2)

See *Include file search procedure*, page 263.

The search procedure for included source file (16.2)

See *Include file search procedure*, page 263.

Search locations for "" header (16.2)

See *Include file search procedure*, page 263.

The sequence of places searched for a header (16.2)

See *Include file search procedure*, page 263.

Nesting limit for #include directives (16.2)

The amount of available memory sets the limit.

#pragma (16.6)

See *Recognized pragma directives (6.10.6)*, page 631.

The definition and meaning of __STDC__ (16.8)

`__STDC__` is predefined to 1.

The text of __DATE__ when date of translation is not available (16.8)

The date of the translation is always available.

The text of __TIME__ when time of translation is not available (16.8)

The time of the translation is always available.

The definition and meaning of __STDC_VERSION__ (16.8)

`__STDC_VERSION__` is predefined to `201112L`.

17 LIBRARY INTRODUCTION

Headers for a freestanding implementation (17.6.1.3)

See *DLIB runtime environment—implementation details*, page 491.

Linkage of names from Standard C library (17.6.2.3)

Declarations from the C library have `"C"` linkage.

Functions in Standard C++ library that can be recursively reentered (17.6.5.8)

Functions can be recursively reentered, unless specified otherwise by the ISO C++ standard.

Exceptions thrown by standard library functions that do not have an exception specification (17.6.5.12)

These functions do not throw any additional exceptions.

error_category for errors originating outside of the operating system (17.6.5.14)

There is no additional error category.

18 LANGUAGE SUPPORT LIBRARY

Definition of NULL (18.2)

`NULL` is predefined as `0`.

The type of ptrdiff_t (18.2)

See *ptrdiff_t*, page 378.

The type of size_t (18.2)

See *size_t*, page 377.

Exit status (18.5)

Control is returned to the `__exit` library function. See *__exit*, page 157.

The return value of bad_alloc::what (18.6.2.1)

The return value is a pointer to `"bad allocation"`.

The return value of bad_array_new_length::what (18.6.2.2)

The return value is a pointer to `"bad allocation"`.

The return value of type_info::name() (18.7.1)

The return value is a pointer to a C string containing the name of the type.

The return value of bad_cast::what (18.7.2)

The return value is a pointer to `"bad cast"`.

The return value of bad_typeid::what (18.7.3)

The return value is a pointer to `"bad typeid"`.

The result of exception::what (18.8.1)

The return value is a pointer to `"unknown"`.

The return value of bad_exception::what (18.8.2)

The return value is a pointer to `"bad exception"`.

The use of non-POF functions as signal handlers (18.10)

Non-Plain Old Functions (POF) can be used as signal handlers if no uncaught exceptions are thrown in the handler, and if the execution of the signal handler does not trigger undefined behavior.

20 GENERAL UTILITIES LIBRARY

get_pointer_safety returning pointer_safety::relaxed or pointer_safety::preferred when the implementation has relaxed pointer safety (20.7.4)

The function `get_pointer_safety` always returns `std::pointer_safety::relaxed`.

Support for over-aligned types (20.7.9.1, 20.7.11)

Over-aligned types are supported.

The exception type when a shared_ptr constructor fails (20.8.2.2.1)

Only `std::bad_alloc` is thrown.

The assignability of placeholder objects (20.9.9.1.4)

Placeholder objects are `CopyAssignable`.

Support for extended alignment (20.10.7.6)

Extended alignment is supported.

Rounding or truncating values to the required precision when converting between time_t values and time_point objects (20.12.7.1)

Values are truncated to the required precision when converting between `time_t` values and `time_point` objects.

21 STRINGS LIBRARY

The type of streampos (21.2.3.1)

The type of `streampos` is `std::fpos<mbstate_t>`.

The type of streamoff (21.2.3.1)

The type of `streamoff` is `long`.

Supported multibyte character encoding rules (21.2.3.1)

See *Locale*, page 164.

The type of u16streampos (21.2.3.2)

The type of `u16streampos` is `streampos`.

The return value of char_traits<char16_t>::eof (21.2.3.2)

The return value of `char_traits<char16_t>::eof` is `EOF`.

The type of u32streampos (21.2.3.3)

The type of `u32streampos` is `streampos`.

The return value of char_traits<char32_t>::eof (21.2.3.3)

The return value of `char_traits<char32_t>::eof` is `EOF`.

The type of wstreampos (21.2.3.4)

The type of `wstreampos` is `streampos`.

The return value of char_traits<wchar_t>::eof (21.2.3.3)

The return value of `char_traits<wchar_t>::eof` is `EOF`.

22 LOCALIZATION LIBRARY

Locale object being global or per-thread (22.3.1)

There is one global locale object for the entire application.

Locale names (22.3.1.2)

See *Locale*, page 164.

The effects on the C locale of calling locale::global (22.3.1.5)

Calling this function with an unnamed locale has no effect.

The value of ctype<char>::table_size (22.4.1.3)

The value of `ctype<char>::table_size` is 256.

Additional formats for time_get::do_get_date (22.4.5.1.2)

No additional formats are accepted for `time_get::do_get_date`.

time_get::do_get_year and two-digit year numbers (22.4.5.1.2)

Two-digit year numbers are accepted by `time_get::do_get_year`. Years from 0 to 68 are parsed as meaning 2000 to 2068, and years from 69 to 99 are parsed as meaning 1969 to 1999.

Formatted character sequences generated by time_put::do_put in the C locale (22.4.5.3.1)

The behavior is the same as that of the library function `strftime`.

Mapping from name to catalog when calling messages::do_open (22.4.7.1.2)

No mapping occurs because this function does not open a catalog.

Mapping to message when calling messages::do_get (22.4.7.1.2)

No mapping occurs because this function does not open a catalog. `dflt` is returned.

Mapping to message when calling messages::do_close (22.4.7.1.2)

The function cannot be called because no catalog can be open.

23 CONTAINERS LIBRARY

The type of array::iterator (23.3.2.1)

The type of `array::iterator` is `T *`.

The type of array::const_iterator (23.3.2.1)

The type of `array::const_iterator` is `T const *`.

The default number of buckets in unordered_map (23.5.4.2)

The IAR C/C++ Compiler for Arm makes a default construction of the `unordered_map` before inserting the elements.

The default number of buckets in unordered_multimap (23.5.5.2)

The IAR C/C++ Compiler for Arm makes a default construction of the `unordered_multimap` before inserting the elements.

The default number of buckets in unordered_set (23.5.6.2)

The IAR C/C++ Compiler for Arm makes a default construction of the `unordered_set` before inserting the elements.

The default number of buckets in unordered_multiset (23.5.7.2)

The IAR C/C++ Compiler for Arm makes a default construction of the `unordered_multiset` before inserting the elements.

25 ALGORITHMS LIBRARY

The underlying source of random numbers for random_shuffle (25.3.12)

The underlying source is `rand()`.

27 INPUT/OUTPUT LIBRARY

The behavior of iostream classes when traits::pos_type is not streampos or when traits::off_type is not streamoff (27.2.2)

No specific behavior has been implemented for this case.

The effects of calling ios_base::sync_with_stdio after any input or output operation on standard streams (27.5.3.4)

Previous input/output is not handled in any special way.

Argument values to construct basic_ios::failure (27.5.5.4)

When `basic_ios::clear` throws an exception, it throws an exception of type `basic_ios::failure` constructed with the `badbit`/`failbit`/`eofbit` set.

The basic_stringbuf move constructor and the copying of sequence pointers (27.8.2.1)

The constructor copies the sequence pointers.

The effects of calling basic_streambuf::setbuf with non-zero arguments (27.8.2.4)

This function has no effect.

The basic_filebuf move constructor and the copying of sequence pointers (27.9.1.2)

The constructor copies the sequence pointers.

The effects of calling basic_filebuf::setbuf with non-zero arguments (27.9.1.5)

This will offer the buffer to the C stream by calling `setvbuf()` with the associated file. If anything goes wrong, the stream is reinitialized.

The effects of calling basic_filebuf::sync when a get area exists (27.9.1.5)

A `get` area cannot exist.

28 REGULAR EXPRESSIONS LIBRARY

The type of regex_constants::error_type (28.5.3)

The type is an `enum`. See *The enum type*, page 370.

29 ATOMIC OPERATIONS LIBRARY

The values of various ATOMIC_..._LOCK_FREE macros (29.4)

In cases where atomic operations are supported, these macros will have the value 2. See *Atomic operations*, page 497.

30 THREAD SUPPORT LIBRARY

The presence and meaning of native_handle_type and native_handle (30.2.3)

The `thread` system header is not supported.

ANNEX D (NORMATIVE): COMPATIBILITY FEATURES

The type of ios_base::streamoff (D.6)

The type of `ios_base::streamoff` is `std::streamoff`.

The type of ios_base::streampos (D.6)

The type of `ios_base::streampos` is `std::streampos`.

Implementation quantities

The IAR Systems implementation of C++ is, like all implementations, limited in the size of the applications it can successfully process.

These limitations apply:

C++ feature	Limitation
Nesting levels of compound statements, iteration control structures, and selection control structures.	Limited only by memory.
Nesting levels of conditional inclusion.	Limited only by memory.
Pointer, array, and function declarators (in any combination) modifying a class, arithmetic, or incomplete type in a declaration.	Limited only by memory.
Nesting levels of parenthesized expressions within a full-expression.	Limited only by memory.
Number of characters in an internal identifier or macro name.	Limited only by memory.

Table 57: C++ implementation quantities

C++ feature	Limitation
Number of characters in an external identifier.	Limited only by memory.
External identifiers in one translation unit.	Limited only by memory.
Identifiers with block scope declared in a block.	Limited only by memory.
Macro identifiers simultaneously defined in one translation unit.	Limited only by memory.
Parameters in one function definition.	Limited only by memory.
Arguments in one function call.	Limited only by memory.
Parameters in one macro definition.	Limited only by memory.
Arguments in one macro invocation.	Limited only by memory.
Characters in one logical source line.	Limited only by memory.
Characters in a string literal (after concatenation).	Limited only by memory.
Size of an object.	Limited only by memory.
Nesting levels for `#include` files.	Limited only by memory.
Case labels for a switch statement (excluding those for any nested switch statements).	Limited only by memory.
Data members in a single class.	Limited only by memory.
Enumeration constants in a single enumeration.	Limited only by memory.
Levels of nested class definitions in a single member-specification.	Limited only by memory.
Functions registered by `atexit`.	Limited by heap memory in the built application.
Functions registered by `at_quick_exit`.	Limited by heap memory in the built application.
Direct and indirect base classes.	Limited only by memory.
Direct base classes for a single class.	Limited only by memory.
Members declared in a single class.	Limited only by memory.
Final overriding virtual functions in a class, accessible or not.	Limited only by memory.
Direct and indirect virtual bases of a class.	Limited only by memory.
Static members of a class.	Limited only by memory.
Friend declarations in a class.	Limited only by memory.
Access control declarations in a class.	Limited only by memory.
Member initializers in a constructor definition.	Limited only by memory.

Table 57: C++ implementation quantities (Continued)

C++ feature	Limitation
Scope qualifiers of one identifier.	Limited only by memory.
Nested external specifications.	Limited only by memory.
Recursive `constexpr` function invocations.	1000. This limit can be changed by using the compiler option `--max_cost_constexpr_call`.
Full-expressions evaluated within a core constant expression.	Limited only by memory.
Template arguments in a template declaration.	Limited only by memory.
Recursively nested template instantiations, including substitution during template argument deduction (14.8.2).	64 for a specific template. This limit can be changed by using the compiler option `--pending_instantiations`.
Handlers per try block.	Limited only by memory.
Throw specifications on a single function declaration.	Limited only by memory.
Number of placeholders (20.9.9.1.4).	20 placeholders from `_1` to `_20`.

Table 57: C++ implementation quantities (Continued)

Implementation-defined behavior for Standard C

- Descriptions of implementation-defined behavior

If you are using C89 instead of Standard C, see *Implementation-defined behavior for C89*, page 643.

Descriptions of implementation-defined behavior

This section follows the same order as the C standard. Each item includes references to the ISO chapter and section (in parenthesis) that explains the implementation-defined behavior.

Note: The IAR Systems implementation adheres to a freestanding implementation of Standard C. This means that parts of a standard library can be excluded in the implementation.

J.3.1 TRANSLATION

Diagnostics (3.10, 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.

White-space characters (5.1.1.2)

At translation phase three, each non-empty sequence of white-space characters is retained.

J.3.2 ENVIRONMENT

The character set (5.1.1.2)

The source character set is the same as the physical source file multibyte character set. By default, the standard ASCII character set is used. However, it can be UTF-8, UTF-16, or the system locale. See *Text encodings*, page 266.

Main (5.1.2.1)

The function called at program startup is called `main`. No prototype is declared for `main`, and the only definition supported for `main` is:

```
int main(void)
```

To change this behavior, see *System initialization*, page 153.

The effect of program termination (5.1.2.1)

Terminating the application returns the execution to the startup code (just after the call to `main`).

Alternative ways to define main (5.1.2.2.1)

There is no alternative ways to define the `main` function.

The argv argument to main (5.1.2.2.1)

The `argv` argument is not supported.

Streams as interactive devices (5.1.2.3)

The streams `stdin`, `stdout`, and `stderr` are treated as interactive devices.

Multithreaded environment (5.1.2.4)

By default, the IAR Systems runtime environment does not support more than one thread of execution. With an optional third-party RTOS, it might support several threads of execution.

Signals, their semantics, and the default handling (7.14)

In the DLIB runtime environment, the set of supported signals is the same as in Standard C. A raised signal will do nothing, unless the `signal` function is customized to fit the application.

Signal values for computational exceptions (7.14.1.1)

In the DLIB runtime environment, there are no implementation-defined values that correspond to a computational exception.

Signals at system startup (7.14.1.1)

In the DLIB runtime environment, there are no implementation-defined signals that are executed at system startup.

Environment names (7.22.4.6)

In the DLIB runtime environment, there are no implementation-defined environment names that are used by the `getenv` function.

The system function (7.22.4.8)

The `system` function is not supported.

J.3.3 IDENTIFIERS

Multibyte characters in identifiers (6.4.2)

Additional multibyte characters may appear in identifiers depending on the chosen encoding for the source file. The supported multibyte characters must be translatable to one Universal Character Name (UCN).

Significant characters in identifiers (5.2.4.1, 6.4.2)

The number of significant initial characters in an identifier with or without external linkage is guaranteed to be no less than 200.

J.3.4 CHARACTERS

Number of bits in a byte (3.6)

A byte contains 8 bits.

Execution character set member values (5.2.1)

The values of the members of the execution character set are the values of the ASCII character set, which can be augmented by the values of the extra characters in the source file character set. The source file character set is determined by the chosen encoding for the source file. See *Text encodings*, page 266.

Alphabetic escape sequences (5.2.2)

The standard alphabetic escape sequences have the values `\a–7`, `\b–8`, `\f–12`, `\n–10`, `\r–13`, `\t–9`, and `\v–11`.

Characters outside of the basic executive character set (6.2.5)

A character outside of the basic executive character set that is stored in a `char` is not transformed.

Plain char (6.2.5, 6.3.1.1)

A plain `char` is treated as an `unsigned char`. See *--char_is_signed*, page 283 and *--char_is_unsigned*, page 283.

Source and execution character sets (6.4.4.4, 5.1.1.2)

The source character set is the set of legal characters that can appear in source files. It is dependent on the chosen encoding for the source file. See *Text encodings*, page 266. By default, the source character set is Raw.

The execution character set is the set of legal characters that can appear in the execution environment. These are the execution character set for character constants and string literals and their encoding types:

Execution character set	Encoding type
`L`	UTF-32
`u`	UTF-16
`U`	UTF-32
`u8`	UTF-8
`none`	The source character set

Table 58: Execution character sets and their encodings

The DLIB runtime environment needs a multibyte character scanner to support a multibyte execution character set. See *Locale*, page 164.

Integer character constants with more than one character (6.4.4.4)

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.

Wide character constants with more than one character (6.4.4.4)

A wide character constant that contains more than one multibyte character generates a diagnostic message.

Locale used for wide character constants (6.4.4.4)

See *Source and execution character sets (6.4.4.4, 5.1.1.2)*, page 626.

Concatenating wide string literals with different encoding types (6.4.5)

Wide string literals with different encoding types cannot be concatenated.

Locale used for wide string literals (6.4.5)

See *Source and execution character sets (6.4.4.4, 5.1.1.2)*, page 626.

Source characters as executive characters (6.4.5)

All source characters can be represented as executive characters.

Encoding of wchar_t, char16_t, and char32_t (6.10.8.2)

`wchar_t` has the encoding UTF-32, `char16_t` has the encoding UTF-16, and `char32_t` has the encoding UTF-32.

J.3.5 INTEGERS

Extended integer types (6.2.5)

There are no extended integer types.

Range of integer values (6.2.6.2)

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.

For information about the ranges for the different integer types, see *Basic data types—integer types*, page 369.

The rank of extended integer types (6.3.1.1)

There are no extended integer types.

Signals when converting to a signed integer type (6.3.1.3)

No signal is raised when an integer is converted to a signed integer type.

Signed bitwise operations (6.5)

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, except for the operator >> which will behave as an arithmetic right shift.

J.3.6 FLOATING POINT

Accuracy of floating-point operations (5.2.4.2.2)

The accuracy of floating-point operations is unknown.

Accuracy of floating-point conversions (5.2.4.2.2)

The accuracy of floating-point conversions is unknown.

Rounding behaviors (5.2.4.2.2)

There are no non-standard values of `FLT_ROUNDS`.

Evaluation methods (5.2.4.2.2)

There are no non-standard values of `FLT_EVAL_METHOD`.

Converting integer values to floating-point values (6.3.1.4)

When an integer value is converted to a floating-point value that cannot exactly represent the source value, the round-to-nearest rounding mode is used (`FLT_ROUNDS` is defined to 1).

Converting floating-point values to floating-point values (6.3.1.5)

When a floating-point value is converted to a floating-point value that cannot exactly represent the source value, the round-to-nearest rounding mode is used (`FLT_ROUNDS` is defined to 1).

Denoting the value of floating-point constants (6.4.4.2)

The round-to-nearest rounding mode is used (`FLT_ROUNDS` is defined to 1).

Contraction of floating-point values (6.5)

Floating-point values are contracted. However, there is no loss in precision and because signaling is not supported, this does not matter.

Default state of FENV_ACCESS (7.6.1)

The default state of the pragma directive `FENV_ACCESS` is `OFF`.

Additional floating-point mechanisms (7.6, 7.12)

There are no additional floating-point exceptions, rounding-modes, environments, and classifications.

Default state of FP_CONTRACT (7.12.2)

The default state of the pragma directive `FP_CONTRACT` is `ON` unless the compiler option `--no_default_fp_contract` is used.

J.3.7 ARRAYS AND POINTERS

Conversion from/to pointers (6.3.2.3)

For information about casting of data pointers and function pointers, see *Casting*, page 377.

ptrdiff_t (6.5.6)

For information about `ptrdiff_t`, see *ptrdiff_t*, page 378.

J.3.8 HINTS

Honoring the register keyword (6.7.1)

User requests for register variables are not honored.

Inlining functions (6.7.4)

User requests for inlining functions increases the chance, but does not make it certain, that the function will actually be inlined into another function. See *Inlining functions*, page 90.

J.3.9 STRUCTURES, UNIONS, ENUMERATIONS, AND BITFIELDS

Sign of 'plain' bitfields (6.7.2, 6.7.2.1)

For information about how a 'plain' `int` bitfield is treated, see *Bitfields*, page 370.

Possible types for bitfields (6.7.2.1)

All integer types can be used as bitfields in the compiler's extended mode, see *-e*, page 293.

Atomic types for bitfields (6.7.2.1)

Atomic types cannot be used as bitfields.

Bitfields straddling a storage-unit boundary (6.7.2.1)

Unless `__attribute__((packed))` (a GNU language extension) is used, a bitfield is always placed in one—and one only—storage unit, and thus does not straddle a storage-unit boundary.

Allocation order of bitfields within a unit (6.7.2.1)

For information about how bitfields are allocated within a storage unit, see *Bitfields*, page 370.

Alignment of non-bitfield structure members (6.7.2.1)

The alignment of non-bitfield members of structures is the same as for the member types, see *Alignment*, page 367.

Integer type used for representing enumeration types (6.7.2.2)

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.

J.3.10 QUALIFIERS

Access to volatile objects (6.7.3)

Any reference to an object with `volatile` qualified type is an access, see *Declaring objects volatile*, page 380.

J.3.11 PREPROCESSING DIRECTIVES

Locations in #pragma for header names (6.4, 6.4.7)

These pragma directives take header names as parameters at the specified positions:

```
#pragma include_alias ("header", "header")
#pragma include_alias (<header>, <header>)
```

Mapping of header names (6.4.7)

Sequences in header names are mapped to source file names verbatim. A backslash '\' is not treated as an escape sequence. See *Overview of the preprocessor*, page 471.

Character constants in constant expressions (6.10.1)

A character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set.

The value of a single-character constant (6.10.1)

A single-character constant may only have a negative value if a plain character (`char`) is treated as a signed character, see *--char_is_signed*, page 283.

Including bracketed filenames (6.10.2)

For information about the search algorithm used for file specifications in angle brackets <>, see *Include file search procedure*, page 263.

Including quoted filenames (6.10.2)

For information about the search algorithm used for file specifications enclosed in quotes, see *Include file search procedure*, page 263.

Preprocessing tokens in #include directives (6.10.2)

Preprocessing tokens in an `#include` directive are combined in the same way as outside an `#include` directive.

Nesting limits for #include directives (6.10.2)

There is no explicit nesting limit for `#include` processing.

inserts \ in front of \u (6.10.3.2)

`#` (stringify argument) inserts a `\` character in front of a Universal Character Name (UCN) in character constants and string literals.

Recognized pragma directives (6.10.6)

In addition to the pragma directives described in the chapter *Pragma directives*, the following directives are recognized and will have an indeterminate effect. If a pragma directive is listed both in the chapter *Pragma directives* and here, the information provided in the chapter *Pragma directives* overrides the information here.

`alias_def`

`alignment`

`alternate_target_def`

`baseaddr`

basic_template_matching

building_runtime

can_instantiate

codeseg

constseg

cplusplus_neutral

cspy_support

cstat_dump

dataseg

define_type_info

do_not_instantiate

early_dynamic_initialization

exception_neutral

function

function_category

function_effects

hdrstop

important_typedef

ident

implements_aspect

init_routines_only_for_needed_variables

initialization_routine

inline_template

instantiate

keep_definition

library_default_requirements

library_provides

library_requirement_override

memory

`module_name`

`no_pch`

`no_vtable_use`

`once`

`pop_macro`

`preferred_typedef`

`push_macro`

`separate_init_routine`

`set_generate_entries_without_bounds`

`system_include`

`uses_aspect`

`vector`

`warnings`

Default __DATE__ and __TIME__ (6.10.8)

The definitions for `__TIME__` and `__DATE__` are always available.

J.3.12 LIBRARY FUNCTIONS

Additional library facilities (5.1.2.1)

Most of the standard library facilities are supported. Some of them—the ones that need an operating system—require a low-level implementation in the application. For more information, see *The DLIB runtime environment*, page 129.

Diagnostic printed by the assert function (7.2.1.1)

The `assert()` function prints:

filename:*linenr expression* `-- assertion failed`

when the parameter evaluates to zero.

Representation of the floating-point status flags (7.6.2.2)

There is no representation of floating-point status flags.

Feraiseexcept raising floating-point exception (7.6.2.3)

For information about the `feraiseexcept` function raising floating-point exceptions, see *Floating-point environment*, page 375.

Strings passed to the setlocale function (7.11.1.1)

For information about strings passed to the `setlocale` function, see *Locale*, page 164.

Types defined for float_t and double_t (7.12)

The `FLT_EVAL_METHOD` macro can only have the value `0`.

Domain errors (7.12.1)

No function generates other domain errors than what the standard requires.

Return values on domain errors (7.12.1)

Mathematic functions return a floating-point `NaN` (not a number) for domain errors.

Underflow errors (7.12.1)

Mathematic functions set `errno` to the macro `ERANGE` (a macro in `errno.h`) and return zero for underflow errors.

fmod return value (7.12.10.1)

The `fmod` function sets `errno` to a domain error and returns a floating-point `NaN` when the second argument is zero.

remainder return value (7.12.10.2)

The `remainder` function sets `errno` to a domain error and returns a floating-point NaN when the second argument is zero.

The magnitude of remquo (7.12.10.3)

The magnitude is congruent modulo `INT_MAX`.

remquo return value (7.12.10.3)

The `remquo` function sets `errno` to a domain error and returns a floating-point NaN when the second argument is zero.

signal() (7.14.1.1)

The signal part of the library is not supported.

Note: The default implementation of `signal` does not perform anything. Use the template source code to implement application-specific signal handling. See *signal*, page 161 and *raise*, page 159, respectively.

NULL macro (7.19)

The `NULL` macro is defined to `0`.

Terminating newline character (7.21.2)

Stream functions recognize either `newline` or `end of file (EOF)` as the terminating character for a line.

Space characters before a newline character (7.21.2)

Space characters written to a stream immediately before a newline character are preserved.

Null characters appended to data written to binary streams (7.21.2)

No null characters are appended to data written to binary streams.

File position in append mode (7.21.3)

The file position is initially placed at the beginning of the file when it is opened in append-mode.

Truncation of files (7.21.3)

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 *Briefly about input and output (I/O)*, page 130.

File buffering (7.21.3)

An open file can be either block-buffered, line-buffered, or unbuffered.

A zero-length file (7.21.3)

Whether a zero-length file exists depends on the application-specific implementation of the low-level file routines.

Legal file names (7.21.3)

The legality of a filename depends on the application-specific implementation of the low-level file routines.

Number of times a file can be opened (7.21.3)

Whether a file can be opened more than once depends on the application-specific implementation of the low-level file routines.

Multibyte characters in a file (7.21.3)

The encoding of multibyte characters in a file depends on the application-specific implementation of the low-level file routines.

remove() (7.21.4.1)

The effect of a remove operation on an open file depends on the application-specific implementation of the low-level file routines. See *Briefly about input and output (I/O)*, page 130.

rename() (7.21.4.2)

The effect of renaming a file to an already existing filename depends on the application-specific implementation of the low-level file routines. See *Briefly about input and output (I/O)*, page 130.

Removal of open temporary files (7.21.4.3)

Whether an open temporary file is removed depends on the application-specific implementation of the low-level file routines.

Mode changing (7.21.5.4)

`freopen` closes the named stream, then reopens it in the new mode. The streams `stdin`, `stdout`, and `stderr` can be reopened in any new mode.

Style for printing infinity or NaN (7.21.6.1, 7.29.2.1)

The style used for printing infinity or `NaN` for a floating-point constant is `inf` and `nan` (`INF` and `NAN` for the `F` conversion specifier), respectively. The n-char-sequence is not used for `nan`.

%p in printf() (7.21.6.1, 7.29.2.1)

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.

Reading ranges in scanf (7.21.6.2, 7.29.2.1)

A `-` (dash) character is always treated as a range symbol.

%p in scanf (7.21.6.2, 7.29.2.2)

The `%p` conversion specifier, scan pointer, to `scanf()` reads a hexadecimal number and converts it into a value with the type `void *`.

File position errors (7.21.9.1, 7.21.9.3, 7.21.9.4)

On file position errors, the functions `fgetpos`, `ftell`, and `fsetpos` store `EFPOS` in `errno`.

An n-char-sequence after nan (7.22.1.3, 7.29.4.1.1)

An n-char-sequence after a NaN is read and ignored.

errno value at underflow (7.22.1.3, 7.29.4.1.1)

`errno` is set to `ERANGE` if an underflow is encountered.

Zero-sized heap objects (7.22.3)

A request for a zero-sized heap object will return a valid pointer and not a null pointer.

Behavior of abort and exit (7.22.4.1, 7.22.4.5)

A call to `abort()` or `_Exit()` will not flush stream buffers, not close open streams, and not remove temporary files.

Termination status (7.22.4.1, 7.22.4.4, 7.22.4.5, 7.22.4.7)

The termination status will be propagated to `__exit()` as a parameter. `exit()`, `_Exit()`, and `quick_exit` use the input parameter, whereas abort uses `EXIT_FAILURE`.

The system function return value (7.22.4.8)

The `system` function returns -1 when its argument is not a null pointer.

Range and precision of clock_t and time_t (7.27)

The range and precision of `clock_t` is up to your implementation. The range and precision of `time_t` is 19000101 up to 20351231 in tics of a second if the 32-bit `time_t` is used. It is -9999 up to 9999 years in tics of a second if the 64-bit `time_t` is used. See *time.h*, page 499

The time zone (7.27.1)

The local time zone and daylight savings time must be defined by the application. For more information, see *time.h*, page 499.

The era for clock() (7.27.2.1)

The era for the `clock` function is up to your implementation.

TIME_UTC epoch (7.27.2.5)

The epoch for `TIME_UTC` is up to your implementation.

%Z replacement string (7.27.3.5, 7.29.5.1)

By default, ":" or "" (an empty string) is used as a replacement for `%Z`. Your application should implement the time zone handling. See *__time32, __time64*, page 162.

Math functions rounding mode (F.10)

The functions in `math.h` honor the rounding direction mode in `FLT-ROUNDS`.

J.3.13 ARCHITECTURE

Values and expressions assigned to some macros (5.2.4.2, 7.20.2, 7.20.3)

There are always 8 bits in a byte.

`MB_LEN_MAX` is at the most 6 bytes depending on the library configuration that is used.

For information about sizes, ranges, etc for all basic types, see *Data representation*, page 367.

The limit macros for the exact-width, minimum-width, and fastest minimum-width integer types defined in `stdint.h` have the same ranges as `char`, `short`, `int`, `long`, and `long long`.

The floating-point constant `FLT_ROUNDS` has the value 1 (to nearest) and the floating-point constant `FLT_EVAL_METHOD` has the value 0 (treat as is).

Accessing another thread's autos or thread locals (6.2.4)

The IAR Systems runtime environment does not allow multiple threads. With a third-party RTOS, the access will take place and work as intended as long as the accessed item has not gone out of its scope.

The number, order, and encoding of bytes (6.2.6.1)

See *Data representation*, page 367.

Extended alignments (6.2.8)

For information about extended alignments, see *data_alignment*, page 406.

Valid alignments (6.2.8)

For information about valid alignments on fundamental types, see the chapter *Data representation*.

The value of the result of the sizeof operator (6.5.3.4)

See *Data representation*, page 367.

J.4 LOCALE

Members of the source and execution character set (5.2.1)

By default, the compiler accepts all one-byte characters in the host's default character set. The chapter *Encodings* describes how to change the default encoding for the source character set, and by that the encoding for plain character constants and plain string literals in the execution character set.

The meaning of the additional characters (5.2.1.2)

Any multibyte characters in the extended source character set is translated into the following encoding for the execution character set:

Execution character set	Encoding
`L` typed	UTF-32
`u` typed	UTF-16
`U` typed	UTF-32
`u8` typed	UTF-8
none typed	The same as the source character set

Table 59: Translation of multibyte characters in the extended source character set

It is up to your application with the support of the library configuration to handle the characters correctly.

Shift states for encoding multibyte characters (5.2.1.2)

No shift states are supported.

Direction of successive printing characters (5.2.2)

The application defines the characteristics of a display device.

The decimal point character (7.1.1)

For a library with the configuration Normal or Tiny, the default decimal-point character is a '.'. For a library with the configuration Full, the chosen locale defines what character is used for the decimal point.

Printing characters (7.4, 7.30.2)

The set of printing characters is determined by the chosen locale.

Control characters (7.4, 7.30.2)

The set of control characters is determined by the chosen locale.

Characters tested for (7.4.1.2, 7.4.1.3, 7.4.1.7, 7.4.1.9, 7.4.1.10, 7.4.1.11, 7.30.2.1.2, 7.30.5.1.3, 7.30.2.1.7, 7.30.2.1.9, 7.30.2.1.10, 7.30.2.1.11)

The set of characters tested for the character-based functions are determined by the chosen locale. The set of characters tested for the `wchar_t`-based functions are the UTF-32 code points `0x0` to `0x7F`.

The native environment (7.11.1.1)

The native environment is the same as the "C" locale.

Subject sequences for numeric conversion functions (7.22.1, 7.29.4.1)

There are no additional subject sequences that can be accepted by the numeric conversion functions.

The collation of the execution character set (7.24.4.3, 7.29.4.4.2)

Collation is not supported.

Message returned by strerror (7.24.6.2)

The messages returned by the `strerror` function depending on the argument is:

<table>
<tr><th>Argument</th><th>Message</th></tr>
<tr><td>EZERO</td><td>no error</td></tr>
<tr><td>EDOM</td><td>domain error</td></tr>
<tr><td>ERANGE</td><td>range error</td></tr>
<tr><td>EFPOS</td><td>file positioning error</td></tr>
<tr><td>EILSEQ</td><td>multi-byte encoding error</td></tr>
<tr><td><0 || >99</td><td>unknown error</td></tr>
<tr><td>all others</td><td>error nnn</td></tr>
</table>

Table 60: Message returned by strerror()—DLIB runtime environment

Formats for time and date (7.27.3.5, 7.29.5.1)

Time zone information is as you have implemented it in the low-level function `__getzone`.

Character mappings (7.30.1)

The character mappings supported are `tolower` and `toupper`.

Character classifications (7.30.1)

The character classifications that are supported are `alnum`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, and `xdigit`.

Implementation-defined behavior for C89

- Descriptions of implementation-defined behavior

If you are using Standard C instead of C89, see *Implementation-defined behavior for Standard C*, page 623.

Descriptions of implementation-defined behavior

The descriptions follow 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 DLIB runtime environment, see *System initialization*, page 153.

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. It is dependent on the chosen encoding for the source file. See *Text encodings*, page 266. By default, the source character set is Raw.

The execution character set is the set of legal characters that can appear in the execution environment. These are the execution character set for character constants and string literals and their encoding types:

Execution character set	Encoding type
`L`	UTF-32
`u`	UTF-16
`U`	UTF-32
`u8`	UTF-8
`none`	The source character set

Table 61: Execution character sets and their encodings

The DLIB runtime environment needs a multibyte character scanner to support a multibyte execution character set. See *Locale*, page 164.

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)

See *Locale*, page 164.

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—integer types*, page 369, 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, except for the operator >> which will behave as an arithmetic right shift.

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 IEC 60559. A typical floating-point number is built up of a sign-bit (`s`), a biased exponent (`e`), and a mantissa (`m`).

See *Basic data types—floating-point types*, page 374, 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 377, for information about `size_t`.

Conversion from/to pointers (6.3.4)

See *Casting*, page 377, for information about casting of data pointers and function pointers.

ptrdiff_t (6.3.6, 7.1.1)

See *ptrdiff_t*, page 378, 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—integer types*, page 369, for information about the alignment requirement for data objects.

Sign of 'plain' bitfields (6.5.2.1)

A 'plain' `int` bitfield is treated as an `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 cannot straddle a storage-unit boundary for the chosen bitfield integer type.

Integer type chosen to represent enumeration types (6.5.2.2)

The chosen integer type for a specific enumeration type depends on the enumeration constants defined for the enumeration type. The chosen integer type is the smallest possible.

QUALIFIERS

Access to volatile objects (6.5.3)

Any reference to an object with volatile qualified type is an access.

DECLARATORS

Maximum numbers of declarators (6.5.4)

The number of declarators is not limited. The number is limited only by the available memory.

STATEMENTS

Maximum number of case statements (6.6.4.2)

The number of case statements (case values) in a switch statement is not limited. The number is limited only by the available memory.

PREPROCESSING DIRECTIVES

Character constants and conditional inclusion (6.8.1)

The character set used in the preprocessor directives is the same as the execution character set. The preprocessor recognizes negative character values if a 'plain' character is treated as a `signed` character.

Including bracketed filenames (6.8.2)

For file specifications enclosed in angle brackets, the preprocessor does not search directories of the parent files. A parent file is the file that contains the `#include` directive. Instead, it begins by searching for the file in the directories specified on the compiler command line.

Including quoted filenames (6.8.2)

For file specifications enclosed in quotes, the preprocessor directory search begins with the directories of the parent file, then proceeds through the directories of any grandparent files. Thus, searching begins relative to the directory containing the source file currently being processed. If there is no grandparent file and the file 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. If a pragma directive is listed both in the chapter *Pragma directives* and here, the information provided in the chapter *Pragma directives* overrides the information here.

```
alignment

baseaddr

basic_template_matching

building_runtime

can_instantiate

codeseg

constseg

cspy_support

dataseg

define_type_info

do_not_instantiate

early_dynamic_initialization

function

function_effects

hdrstop

important_typedef

instantiate

keep_definition

library_default_requirements
```

`library_provides`

`library_requirement_override`

`memory`

`module_name`

`no_pch`

`once`

`system_include`

`vector`

`warnings`

Default __DATE__ and __TIME__ (6.8.8)

The definitions for `__TIME__` and `__DATE__` are always available.

LIBRARY FUNCTIONS FOR THE IAR DLIB RUNTIME ENVIRONMENT

Note: Some items in this list only apply when file descriptors are supported by the library configuration. For more information about runtime library configurations, see the chapter *The DLIB runtime environment*.

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: The default implementation of `signal` does not perform anything. Use the template source code to implement application-specific signal handling. See *signal*, page 161 and *raise*, page 159, respectively.

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 the file position indicator of an append-mode stream is initially positioned at the beginning or the end of the file, depends on the application-specific implementation of the low-level file routines.

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 *Briefly about input and output (I/O)*, page 130.

The characteristics of the file buffering is that the implementation supports files that are unbuffered, line buffered, or fully buffered.

Whether a zero-length file actually exists depends on the application-specific implementation of the low-level file routines.

Rules for composing valid file names depends on the application-specific implementation of the low-level file routines.

Whether the same file can be simultaneously open multiple times depends on the application-specific implementation of the low-level file routines.

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 *Briefly about input and output (I/O)*, page 130.

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 *Briefly about input and output (I/O)*, page 130.

%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 *getenv*, page 157.

system() (7.10.4.5)

How the command processor works depends on how you have implemented the `system` function. See *system*, page 162.

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 62: Message returned by strerror()—DLIB runtime environment

The time zone (7.12.1)

The local time zone and daylight savings time implementation is described in *__time32, __time64*, page 162.

clock() (7.12.2.1)

From where the system clock starts counting depends on how you have implemented the `clock` function. See *clock*, page 156.

A

B

C

D

E

F

G

H

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Symbols

Numerics