IAR Embedded Workbench®

IAR C/C++ Development Guide

Compiling and linking

for the Renesas

RX Family

DRX-7

IAR SYSTEMS

EDITION NOTICE
Seventh edition: January 2015

Part number: DRX-7

This guide applies to version 2.x of IAR Embedded Workbench® for the Renesas RX family.

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

Internal reference: M17, csrct2010.1, V_110411, IJOA.

Brief contents

Contents

Tables

Preface

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

Who should read this guide

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

REQUIRED KNOWLEDGE

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

- The architecture and instruction set of the RX microcontroller family (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 31.

How to use this guide

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

PART 2. REFERENCE INFORMATION

- *External interface details* provides reference information about how the compiler and linker interact with their environment—the invocation syntax, methods for passing options to the compiler and linker, environment variables, the include file

search procedure, and the different types of compiler and linker output. The chapter also describes how the diagnostic system works.

- *Compiler options* explains how to set options, gives a summary of the options, and contains detailed reference information for each compiler option.
- *Linker options* gives a summary of the options, and contains detailed reference information for each linker option.
- *Data representation* describes the available data types, pointers, and structure types. This chapter also gives information about type and object attributes.
- *Extended keywords* gives reference information about each of the RX-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 RX-specific low-level features.
- *The preprocessor* gives a brief overview of the preprocessor, including reference information about the different preprocessor directives, symbols, and other related information.
- *Library functions* gives an introduction to the C or C++ library functions, and summarizes the header files.
- *The linker configuration file* describes the purpose of the linker configuration file and describes its contents.
- *Section reference* gives reference information about the use of sections.
- *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 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, is available in the booklet Quick Reference (available in the product box) and the *Installation and Licensing Guide*.
- Getting started using IAR Embedded Workbench and the tools it provides, is available in the guide *Getting Started with IAR Embedded Workbench®*.
- Using the IDE for project management and building, is available in the *IDE Project Management and Building Guide.*
- Using the IAR C-SPY® Debugger, is available in the *C-SPY® Debugging Guide for RX.*
- Programming for the IAR C/C++ Compiler for RX and linking using the IAR ILINK Linker, is available in the *IAR C/C++ Development Guide for RX.*
- Programming for the IAR Assembler for RX, is available in the *IAR Assembler User Guide for RX.*
- Using the IAR DLIB Library, is available in the *DLIB Library Reference information*, available in the online help system.
- Porting application code and projects created with a previous version of the IAR Embedded Workbench for RX, is available in the *IAR Embedded Workbench® Migration Guide*.
- Porting application code and projects created with the Renesas High-performance Embedded Workshop for RX, is available in the guide *Migrating from Renesas HEW tool chain for RX to IAR Embedded Workbench® for RX*.
- Migrating from an older UBROF-based product version to a newer version that uses the ELF/DWARF object format, is available in the guide *IAR Embedded Workbench® Migrating from UBROF to ELF/DWARF*.
- Developing safety-critical applications using the MISRA C guidelines, is available in the *IAR Embedded Workbench® MISRA C:2004 Reference Guide* or the *IAR Embedded Workbench® MISRA C:1998 Reference Guide*.

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

THE ONLINE HELP SYSTEM

The context-sensitive online help contains:

- Information about project management, editing, and building in the IDE
- Information about debugging using the IAR C-SPY® Debugger

- Reference information about the menus, windows, and dialog boxes in the IDE
- Compiler reference information
- 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.

FURTHER READING

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

- Barr, Michael, and Andy Oram, ed. *Programming Embedded Systems in C and C++*. O'Reilly & Associates.
- Harbison, Samuel P. and Guy L. Steele (contributor). *C: A Reference Manual.* Prentice Hall.
- Josuttis, Nicolai M. *The C++ Standard Library: A Tutorial and Reference.* Addison-Wesley.
- Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language. Prentice Hall*.
- Labrosse, Jean J. *Embedded Systems Building Blocks: Complete and Ready-To-Use Modules in C.* R&D Books.
- Lippman, Stanley B. and Josée Lajoie. *C++ Primer*. Addison-Wesley.
- Mann, Bernhard. C für Mikrocontroller. Franzis-Verlag. [Written in German.]
- Meyers, Scott. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs.* Addison-Wesley.
- Meyers, Scott. *More Effective C++.* Addison-Wesley.
- Meyers, Scott. *Effective STL.* Addison-Wesley.
- Stroustrup, Bjarne. *The C++ Programming Language.* Addison-Wesley.
- Stroustrup, Bjarne. *Programming Principles and Practice Using C++.* Addison-Wesley.
- Sutter, Herb. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions.* Addison-Wesley.

WEB SITES

Recommended web sites:

- The Renesas web site, **www.renesas.com**, that contains information and news about the RX microcontrollers.
- 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**.
- Finally, the Embedded C++ Technical Committee web site, **www.caravan.net/ec2plus**, that contains information about the Embedded C++ standard.

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 `rx\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench 7.n\rx\doc`.

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 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 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.
`[a|b|c]`	An optional part of a command with alternatives.
`{a|b|c}`	A mandatory part of a command with alternatives.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
italic	• A cross-reference within this guide or to another guide. • Emphasis.
…	An ellipsis indicates that the previous item can be repeated an arbitrary number of times.

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

Table 2: Naming conventions used in this guide

Part 1. Using the build tools

This part of the *IAR C/C++ Development Guide for RX* 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
- 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 RX-based embedded applications. The tools allow you to develop your application in C, C++, or in assembler language.

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

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

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

IAR C/C++ COMPILER

The IAR C/C++ Compiler for RX 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 RX-specific facilities.

IAR ASSEMBLER

The IAR Assembler for RX 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 RX uses the same mnemonics and operand syntax as the Renesas RX Assembler, which simplifies the migration of existing code. For more information, see the *IAR Assembler User Guide for RX*.

THE IAR ILINK LINKER

The IAR ILINK Linker for RX 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 RX—`ielfdumprx`—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.

EXTERNAL TOOLS

For information about how to extend the tool chain in the IDE, see the *IDE Project Management and Building Guide*.

IAR language overview

The IAR C/C++ Compiler for RX 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 C99. Hereafter, this standard is referred to as *Standard C* in this guide.
  - C89—also known as C94, C90, C89, and ANSI C. This standard is required when MISRA C is enabled.

- C++, a modern object-oriented programming language with a full-featured library well suited for modular programming. Any of these standards can be used:
  - Embedded C++ (EC++)—a subset of the C++ programming standard, which is intended for embedded systems programming. It is defined by an industry consortium, the Embedded C++ Technical committee. See the chapter *Using C++*.
  - IAR Extended Embedded C++ (EEC++)—EC++ with additional features such as full template support, multiple inheritance, namespace support, the new cast operators, as well as the Standard Template Library (STL).

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

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

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

For information about how the compiler handles the implementation-defined areas of the languages, see the chapter *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 RX*.

Device support

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

SUPPORTED RX DEVICES

The IAR C/C++ Compiler for RX supports all devices based on the standard Renesas RX microcontroller. The single-precision hardware floating-point unit (FPU) is also supported.

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 `rx\inc` directory. Make sure to include the appropriate include file in your application source files. If you need additional I/O header files, they can be created using one of the provided ones as a template.

Linker configuration files

The `rx\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 83, 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 `rx\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 RX*.

EXAMPLES FOR GETTING STARTED

The `rx\examples` directory contains examples of working applications to give you a smooth start with your development.

Special support for embedded systems

This section briefly describes the extensions provided by the compiler to support specific features of the RX microcontroller.

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 249 for additional information.

For more information about the extended keywords, see the chapter *Extended keywords*. See also, *Data storage*, page 59 and *Functions*, page 69.

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 very 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 145.

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.

CPU FEATURES AND CONSTRAINTS

Some of the basic features of the RX microcontroller are:

- Variable byte order for data access
- A multiplier/divider
- MAC units
- Byte variable-length instructions
- A floating-point unit
- Fast interrupts
- Alignment constraints.

The compiler supports this by means of compiler options, extended keywords, pragma directives etc.

MAPPING OF INTERNAL AND EXTERNAL MEMORY

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

As an embedded software developer, you must understand the features of the different 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 accessed seldom but must maintain their value after power off, so they should be saved in EEPROM or flash memory.

For efficient memory usage, the compiler provides several mechanisms for controlling placement of functions and data objects in memory. For more information, see *Controlling data and function placement in memory*, page 207. 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 83.

COMMUNICATION WITH PERIPHERAL UNITS

If external devices are connected to the microcontroller, you might need to initialize and control the signalling interface, for example by using chip select pins, and detect and handle external interrupt signals. Typically, this must be initialized and controlled at runtime. The normal way to do this is to use special function registers (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 41. For an example, see *Accessing special function registers*, page 219.

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 microcontroller 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 *Primitives for interrupts, concurrency, and OS-related programming*, page 70.

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.

As an embedded software developer, you must ensure that the startup code is located at the dedicated memory addresses, or can be accessed using a pointer from the vector

table. This means that startup code and the initial vector table must be placed in non-volatile memory, such as ROM, EPROM, or flash.

A C/C++ application further needs to initialize all global variables. This initialization is handled by the linker and the system startup code in conjunction. For more information, see *Application execution—an overview*, page 50.

REAL-TIME OPERATING SYSTEMS

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

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

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

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

INTEROPERABILITY WITH OTHER BUILD TOOLS

The IAR compiler and linker provide support for the RX ABI, the Application Binary Interface for RX. For more information about this interface specification, see the `www.renesas.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 RX ABI standard.

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

The build process—an overview

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

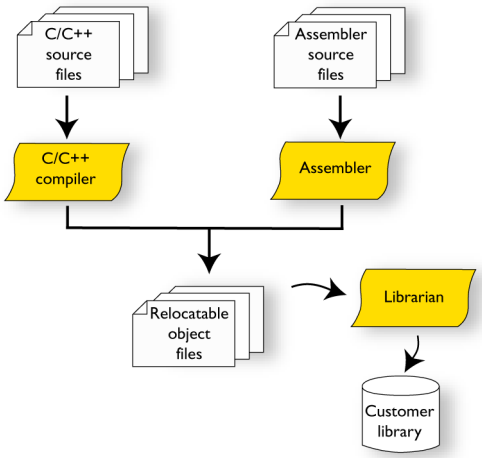
To get familiar with the process in practice, you should run one or more of the tutorials available from the IAR Information Center.

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

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

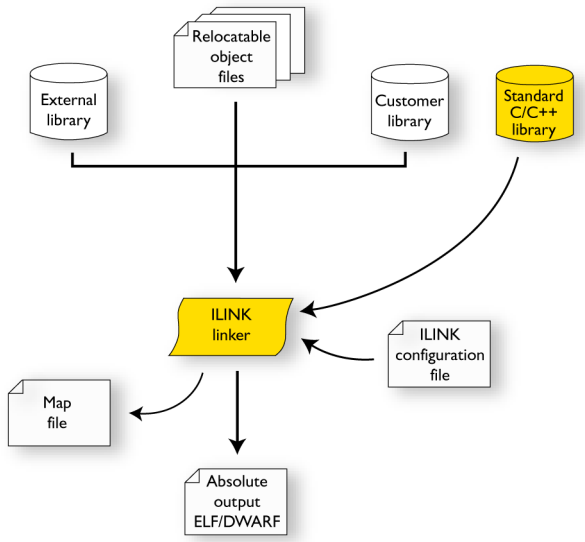
The relocatable modules, in object files and libraries, produced by the IAR compiler and assembler cannot be executed as is. To become an executable application, they must be *linked.*

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

The IAR ILINK Linker (`ilinkrx.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.

During the 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.

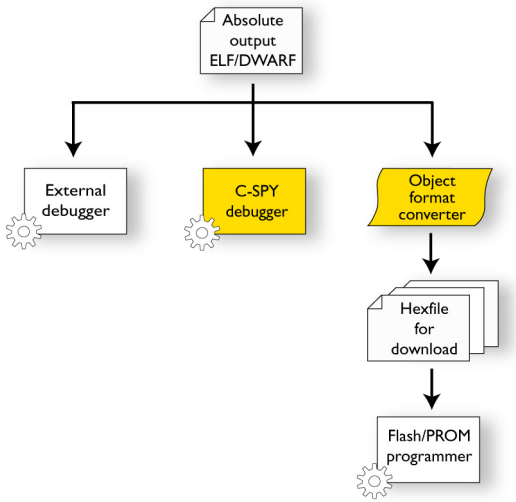
For more information about the procedure performed by the linker, see *The linking process in detail*, page 81.

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 414.

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. The initialization phase can for simplicity be divided into:

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

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

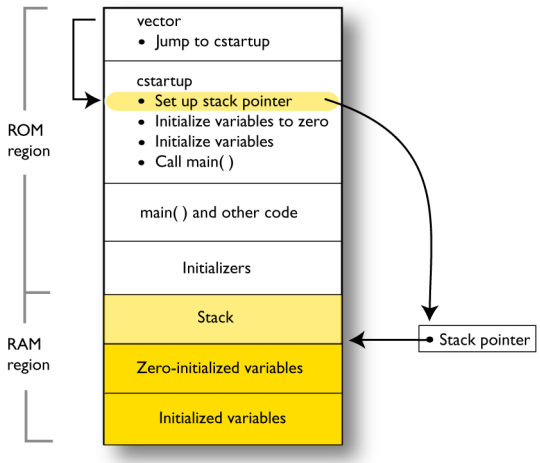
  Typically, this includes assuring that every global (statically linked) C/C++ symbol receives its proper initialization value before the `main` function is called.
- Application initialization

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

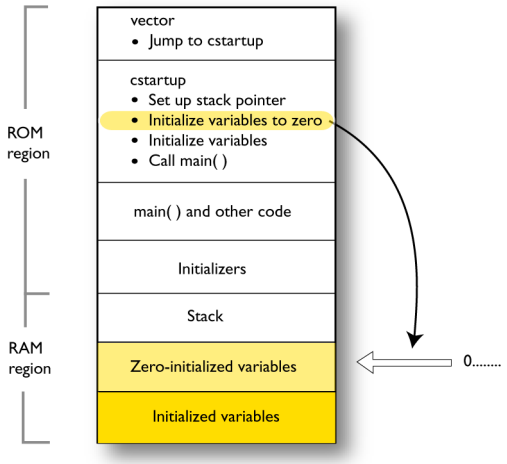
For a ROM/flash-based system, constants and functions are already placed in ROM. All symbols placed in RAM must be initialized before the `main` function is called. The linker has already divided the available RAM into different areas for variables, stack, heap, etc.

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

1 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 (outside the actual stack):

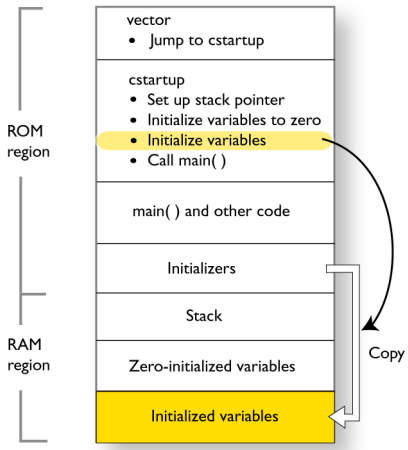


2 Then, memories that should be zero-initialized are cleared, in other words, filled with zeros:

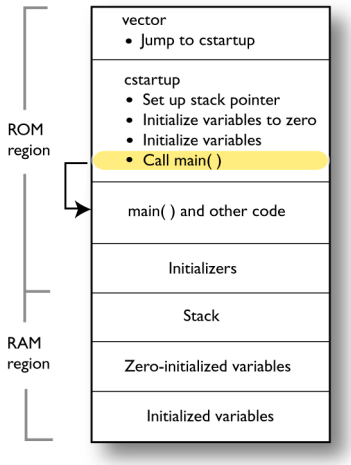


Typically, this is data referred to as *zero-initialized data*; variables declared as, for example, `int i = 0;`

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



4 Finally, the `main` function is called:



For more information about each stage, see *System startup and termination*, page 119. For more information about initialization of data, see *Initialization at system startup*, page 86.

THE EXECUTION PHASE

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

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

THE TERMINATION PHASE

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

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

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

For more information about this, see *System termination*, page 122.

Building applications—an overview

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

```
iccrx myfile.c
```

You must also specify some critical options, see *Basic project configuration*, page 55.

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

```
ilinkrx 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, choose **Tools>Options> Messages** and select the option **Show build messages: All**.

Basic project configuration

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

You need to make settings for:

- Core
- Byte order
- Position independence (read-only or read-write)
- Data model
- Size of `int` data type
- Size of `double` floating-point type
- Optimization settings
- Runtime environment
- Customizing the ILINK configuration, see the chapter *Linking your application*.

In addition to these settings, many other options and settings can 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*, respectively.

PROCESSOR CONFIGURATION

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

Core

The compiler supports all RX architectures.

In the IDE, choose **Project>Options>General Options>Target** and choose an appropriate device from the **Device** drop-down list. The core and device options will then be automatically selected.

Use the `--core={rxv1|rxv2}` option to select the core for which the code will be generated.

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

Byte order

For data access, the RX architecture allows a choice between the big- and little-endian byte order. All user and library modules in your application must use the same byte order.

In the IDE, choose **Project>Options>General Options>Byte order** to set the byte order for data.

Use the `--endian` option to specify the byte order for data for your project; see *--endian*, page 250, for syntax information.

ROPI/RWPI

Most applications are designed to be placed at a fixed position in memory. However, sometimes it is useful to instead decide at runtime where to place the application, for example in certain systems where applications are loaded dynamically.

Use the `--ropi` and `--rwpi` options to configure the compiler to generate position-independent code and data.

In the IDE, choose **Project>Options>General Options>Target>Code and read-only data** and/or **Project>Options>General Options>Target>Read-write data** to generate position-independent code and data.

Read about the advantages and drawbacks with ROPI/RWPI in *Position-independent code and data*, page 191.

DATA MODEL

One of the characteristics of the RX microcontroller is a trade-off in how memory is accessed, ranging from cheap access to small memory areas, up to more expensive access methods that can access any location.

In the compiler, you can set a default memory access method by selecting a data model. These data models are supported:

- The *Near* data model can access the highest and lowest 32 Kbytes of memory

- The *Far* data model can access the highest and lowest 8 Mbytes of memory
- The *Huge* data model can access the entire 32-bit address area.

The chapter *Data storage* covers data models in greater detail. The chapter also covers how to override the default access method for individual variables.

SIZE OF INT DATA TYPE

The `int` data type can be represented with either 16 or 32 (default) bits. Use the compiler option `--int` to change the default size if you are migrating code written for another microcontroller that uses a 16-bit `int` size. See *--int*, page 252.

SIZE OF DOUBLE FLOATING-POINT TYPE

Floating-point values are represented by 32- and 64-bit numbers in standard IEEE 754 format. If you use the compiler option `--double={32|64}`, you can choose whether data declared as `double` should be represented with 32 bits or 64 bits. The data type `float` is always represented using 32 bits.

OPTIMIZATION FOR SPEED AND SIZE

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

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

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

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

RUNTIME ENVIRONMENT

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

To set up an efficient runtime environment you need a good understanding of the various features, see the chapter *The DLIB runtime environment*.

Setting up for the runtime environment in the IDE

The library is automatically chosen according to the settings you make in **Project>Options>General Options**, on the pages **Target**, **Library Configuration**, **Library Options**. A correct include path is automatically set up for the system header files and for the device-specific include files.

Note that for the DLIB library there are different configurations— Normal and Full—which include different levels of support for locale, file descriptors, multibyte characters, etc. See *Library configurations*, page 124, for more information.

Setting up for the runtime environment from the command line

You do not have to specify a library file explicitly, as ILINK automatically uses the correct library file.

A library configuration file that matches the library object file is automatically used. To explicitly specify a library configuration, use the `--dlib_config` option.

In addition to these options you might want to specify any application-specific linker options or the include path to application-specific header files by using the `-I` option, for example:

```
-I MyApplication\inc
```

For information about the prebuilt library object files, see *Using prebuilt libraries*, page 109.

Setting library and runtime environment options

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

- The formatters used by the functions `printf`, `scanf`, and their variants, see *Choosing formatters for printf and scanf*, page 112.
- The size of the stack and the heap, see *Setting up stack memory*, page 99, and *Setting up heap memory*, page 99, respectively.

Data storage

- Introduction
- Memory types
- Data models
- Storage of auto variables and parameters
- Dynamic memory on the heap

Introduction

The RX microcontroller has one continuous memory space for both code and data, ranging from `0x00000000` to `0xFFFFFFFF`. Different types of memory can be placed in the memory range. A typical application will have ROM memory in the upper address interval, and RAM in the lower address interval.

Both code and data can be efficiently read. Physically, data and code reside on different memory buses, but the address spaces are disjoint

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 *The user mode and supervisor mode stacks*, page 190 and *Storage of auto variables and parameters*, page 66.
- 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. For more information, see *Data models*, page 64 and *Memory types*, page 60.
- Dynamically allocated data.

  An application can allocate data on the *heap*, where the data remains valid until it is explicitly released back to the system by the application. This type of memory is useful when the number of objects is not known until the application executes. Note

that there are potential risks connected with using dynamically allocated data in systems with a limited amount of memory, or systems that are expected to run for a long time. For more information, see *Dynamic memory on the heap*, page 67.

Memory types

This section describes the concept of *memory types* used for accessing data by the compiler. It also discusses pointers in the presence of multiple memory types. For each memory type, the capabilities and limitations are discussed.

INTRODUCTION TO MEMORY TYPES

The compiler uses different memory types to access data that is placed in different areas of the memory. There are different methods for reaching memory areas, and they have different costs when it comes to code space, execution speed, and register usage. The access methods range from generic but expensive methods that can access the full memory space, to cheap methods that can access limited memory areas. Each memory type corresponds to one memory access method. If you map different memories—or part of memories—to memory types, the compiler can generate code that can access data efficiently.

For example, the memory accessed using 16-bit addressing is called data16 memory.

To choose a default memory type that your application will use, select a *data model*. However, it is possible to specify—for individual variables—different memory types. This makes it possible to create an application that can contain a large amount of data, and at the same time make sure that variables that are used often are placed in memory that can be efficiently accessed.

Below is an overview of the various memory types.

data16

The data16 memory consists of the highest and the lowest 32 Kbytes of data memory. This is the address ranges `0x00000000-0x00007FFF` and `0xFFFF8000-0xFFFFFFFF`.

A data16 object can only be placed in data16 memory, and the size of such an object is limited to 32 Kbytes-1. If you use objects of this type, the code generated by the compiler to access them becomes slightly smaller. This means a smaller footprint for the application, and faster execution at runtime.

data24

The data24 memory consists of the highest and the lowest 8 Mbytes of data memory. In hexadecimal notation, this is the address ranges `0x00000000-0x007FFFFF` and `0xFF800000-0xFFFFFFFF`.

A data24 object can only be placed in data24 memory, and the size of such an object is limited to 8 Mbytes-1.

data32

Using this memory type, you can place the data objects anywhere in the data memory space. Also, unlike the other memory types, there is no limitation on the size of the objects that can be placed in this memory type.

The data32 memory type uses 4-byte addresses, which can make the code slightly larger.

The compiler will optimize direct accesses (using literal addresses) so that the size penalty for using different memory types becomes smaller.

sbrel

Sbrel memory uses a static base register, relative to which all accesses are made. Sbrel memory requires RWPI, and special linker directives that define a movable block. The movable block can be placed anywhere in memory. Sbrel is the default memory type when RWPI is enabled.

In all other respects, sbrel memory is identical to data32 memory.

USING DATA MEMORY ATTRIBUTES

The compiler provides a set of *extended keywords*, which can be used as *data memory attributes*. These keywords let you override the default memory type for individual data objects, which means that you can place data objects in other memory areas than the default memory. This also means that you can fine-tune the access method for each individual data object, which results in smaller code size.

This table summarizes the available memory types and their corresponding keywords:

Memory type	Keyword	Address range	Default in data model
Data16	`__data16`	`0x00000000-0x00007FFF` and `0xFFFF8000-0xFFFFFFFF`	Near
Data24	`__data24`	`0x00000000-0x007FFFFF` and `0xFF800000-0xFFFFFFFF`	Far
Data32	`__data32`	`0x00000000-0xFFFFFFFF`	Huge
Sbrel	`__sbrel`	`0x00000000-0xFFFFFFFF`	*when using RWPI*

Table 3: Memory types and their corresponding memory attributes

All data pointers are 32 bits.

The keywords are only available if 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 249 for additional information.

For more information about each keyword, see *Descriptions of extended keywords*, page 311.

Syntax

The keywords follow the same syntax as the type qualifiers `const` and `volatile`. The memory attributes are *type attributes* and therefore they must be specified both when variables are defined and in the declaration, see *General syntax rules for extended keywords*, page 307.

The following declarations place the variables `i` and `j` in data16 memory. The variables `k` and `l` will also be placed in data16 memory. The position of the keyword does not have any effect in this case:

```
__data16 int i, j;
int __data16 k, l;
```

Note that the keyword affects both identifiers. If no memory type is specified, the default memory type is used.

The `#pragma type_attribute` directive can also be used for specifying the memory attributes. The advantage of using pragma directives for specifying keywords is that it offers you a method to make sure that the source code is portable. Refer to the chapter *Pragma directives* for details about how to use the extended keywords together with pragma directives.

Type definitions

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

```
/* Defines via a typedef */
typedef char __data32 Byte;
typedef Byte *BytePtr;
Byte aByte;
BytePtr aBytePointer;

/* Defines directly */
__data32 char aByte;
char __data32 *aBytePointer;
```

POINTERS AND MEMORY TYPES

Pointers are used for referring to the location of data. In general, a pointer has a type. For example, a pointer that has the type `int *` points to an integer.

In the compiler, a pointer also points to some type of memory. The memory type is specified using a keyword before the asterisk. For example, a pointer that points to an integer stored in data16 memory is declared by:

```
int __data16 * MyPtr;
```

Note that the location of the pointer variable `MyPtr` is not affected by the keyword. In the following example, however, the pointer variable `MyPtr2` is placed in data16 memory. Like `MyPtr`, `MyPtr2` points to a character in data24 memory.

```
char __data24 * __data16 MyPtr2;
```

STRUCTURES AND MEMORY TYPES

For structures, the entire object is placed in the same memory type. It is not possible to place individual structure members in different memory types.

In the example below, the variable `gamma` is a structure placed in data16 memory.

```
struct MyStruct
{
  int mAlpha;
  int mBeta;
};

__data16 struct MyStruct gamma;
```

This declaration is incorrect:

```
struct MyStruct
{
  int mAlpha;
  __data16 int mBeta; /* Incorrect declaration */
};
```

MORE EXAMPLES

The following is a series of examples with descriptions. First, some integer variables are defined and then pointer variables are introduced. Finally, a function accepting a pointer to an integer in data16 memory is declared. The function returns a pointer to an integer

in data24 memory. It makes no difference whether the memory attribute is placed before or after the data type.

`int MyA;`	A variable defined in default memory determined by the data model in use.
`int __data16 MyB;`	A variable in data16 memory.
`__data24 int MyC;`	A variable in data24 memory.
`int * MyD;`	A pointer stored in default memory. The pointer points to an integer in default memory.
`int __data16 * MyE;`	A pointer stored in default memory. The pointer points to an integer in data16 memory.
`int __data16 * __data24 MyF;`	A pointer stored in data24 memory pointing to an integer stored in data16 memory.
`int __data24 * MyFunction(` `  int __data16 *);`	A declaration of a function that takes a parameter which is a pointer to an integer stored in data16 memory. The function returns a pointer to an integer stored in data24 memory.

C++ AND MEMORY TYPES

Instances of C++ classes are placed into a memory (just like all other objects) either implicitly, or explicitly using memory type attributes or other IAR language extensions. Non-static member variables, like structure fields, are part of the larger object and cannot be placed individually into specified memories.

In non-static member functions, the non-static member variables of a C++ object can be referenced via the `this` pointer, explicitly or implicitly. The `this` pointer is of the default data pointer type unless class memory is used, see *Using IAR attributes with Classes*, page 181.

Static member variables can be placed individually into a data memory in the same way as free variables.

For more information about C++ classes, see *Using IAR attributes with Classes*, page 181.

Data models

Technically, the data model specifies the default memory type. This means that the data model controls the default placement of static and global variables, and constant literals.

The data model only specifies the default memory type. It is possible to override this for individual variables and pointers. For information about how to specify a memory type for individual objects, see *Using data memory attributes*, page 61.

Note: Your choice of data model does not affect the placement of code.

SPECIFYING A DATA MODEL

Three data models are implemented: Near, Far, and Huge. These models are controlled by the `--data_model` option. Each model has a default memory type. If you do not specify a data model option, the compiler will use the Far data model.

Your project can only use one data model at a time, and the same model must be used by all user modules and all library modules. However, you can override the default memory type for individual data objects by explicitly specifying a memory attribute, see *Using data memory attributes*, page 61.

This table summarizes the different data models:

Data model name	Default memory attribute	Pointer attribute	Placement of data
Near	`__data16`	`__data32`	Low 32 Kbytes or high 32 Kbytes
Far (default)	`__data24`	`__data32`	Low 8 Mbytes or high 8 Mbytes
Huge	`__data32`	`__data32`	The entire 4 Gbytes of memory

Table 4: Data model characteristics

See the *IDE Project Management and Building Guide* for information about setting options in the IDE.

Use the `--data_model` option to specify the data model for your project; see *--data_model*, page 243.

The impact of the different data models on the code size depends on the amount of data with static duration. There is no principal difference in the generated code. On higher optimization levels the difference is even smaller, because of the global clustering optimization.

The RX microcontroller has no mode for direct addressing. This means that addresses of static objects must be loaded into a register before the data can be read from memory. The size of these address loads will increase if you change to a larger data model. However, on high optimization levels, the compiler will use a base address to all objects with static duration data in the module, and use relative addressing to access them.

For this reason, the size of the generated code does not depend very much on your choice of data model, but you should nevertheless always use the smallest data model that you need.

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; the rest are placed on the stack. From a semantic point of view, this is equivalent. The main differences are that accessing registers is faster, and that less memory is required compared to when variables are located on the stack.

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

THE STACK

The stack can contain:

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

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

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

See also *Stack considerations*, page 190 and *Setting up stack memory*, page 99.

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. 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 very useful for applications where the amount of data is not known until runtime.

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

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

See also *Setting up heap memory*, page 99.

POTENTIAL PROBLEMS

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

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

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

There is also the matter of fragmentation; this means a heap where small sections of free memory is separated by memory used by allocated objects. It is not possible to allocate

a new object if no piece of free memory is large enough for the object, even though the sum of the sizes of the free memory exceeds the size of the object.

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

Functions

- Function-related extensions
- Executing functions in RAM
- Primitives for interrupts, concurrency, and OS-related programming
- Inlining functions

Function-related extensions

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

- Execute functions in RAM
- Use primitives for interrupts, concurrency, and OS-related programming
- Control function inlining
- Facilitate function optimization
- Access hardware features.

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

For more information about optimizations, see *Efficient coding for embedded applications*, page 203. For information about the available intrinsic functions for accessing hardware operations, see the chapter *Intrinsic functions*.

Executing functions in RAM

In little-endian mode, 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 119. (The keyword cannot be used in big-endian mode.)

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: initializer by cstartup */
  static int myc[] = { 10, 20 };

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

  msg(hello);
}
```

For more details, see *Initializing code—copying ROM to RAM*, page 102.

Primitives for interrupts, concurrency, and OS-related programming

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

- The extended keywords: `__interrupt`, `__nested`, `__task`, `__fast_interrupt`, `__monitor`
- The pragma directive `#pragma vector`
- The intrinsic functions: `__enable_interrupt`, `__disable_interrupt`.

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 microcontroller 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 RX microcontroller supports many interrupt sources. For each interrupt source, an interrupt routine can be written. Each interrupt routine is associated with a vector number, which is specified in the RX microcontroller documentation from the chip manufacturer. If you want to handle several different interrupts using the same interrupt routine, you can specify several interrupt vectors.

Interrupt vectors and the interrupt vector table

For the RX microcontroller, the `INTB` (interrupt table) register points to the start of the interrupt vector table and is placed in the `.inttable` section. The interrupt vector number is the index into the interrupt vector table.

By default, the vector table is populated with a *default interrupt handler* which calls the `abort` function. 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.

The header file `io`*`device`*`.h`, where *`device`* corresponds to the selected device, contains predefined names for the existing interrupt vectors.

Defining an interrupt function—an example

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

```
#include "iorx62n.h"

#pragma vector = VECT_CMT0_CMI0  /* Symbol defined in I/O header
                                    file */
__interrupt void MyInterruptRoutine(void)
{
  /* Do something */
}
```

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.

Adding an exception handler

To overload a default exception handler such as an undefined or non-maskable interrupt, you define a user function with one of the names specified in the template file `fixedint.c`. These are:

```
__interrupt void __floating_point_handler();
__interrupt void __NMI_handler();
__interrupt void __privileged_handler();
__interrupt void __undefined_handler();
```

However, `__floating_point_handler` might already be overloaded with an "unimplemented processing handler" that emulates floating-point for subnormal arguments and results.

To overload the default floating-point exception handler, you must use this special linker mechanism:

If you are *not* using the unimplemented processing handler described above, specify the linker option:

```
--redirect __float_placeholder=_my_float_handler
```

where *`my_float_handler`* is the name you choose for this handler.

However, if you *are* using the unimplemented processing handler described above, specify the linker option:

```
--redirect __floating_point_handler=_my_float_handler
```

The unimplemented processing handler will call `__floating_point_handler` if the exception was not caused by unimplemented processing.

FAST INTERRUPT FUNCTIONS

A fast interrupt function is very fast and has the highest priority. A fast interrupt uses the `FREIT` return mechanism and the `FINTV` register as a vector. Use the intrinsic function `__set_FINTV_register` to initialize this vector register, see *`__set_FINTV_register`*, page 347.

To specify a fast interrupt function, use the `__fast_interrupt` keyword; see *`__fast_interrupt`*, page 313.

NESTED INTERRUPTS

Interrupts are automatically disabled by the RX microcontroller prior to entering an interrupt handler. To make nested interrupts possible, in other words, interrupts within interrupts, the keyword `__nested` must be used in addition to `__interrupt` or `__fast_interrupt`. The interrupt function entrance sequence will then enable interrupts the first thing that occurs.

For more information, see *__nested*, page 314.

MONITOR FUNCTIONS

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

To define a monitor function, you can use the `__monitor` keyword. For more information, see *__monitor*, page 313.

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

Example of implementing a semaphore in C

In the following example, a binary semaphore—that is, a mutex—is implemented using one static variable and two monitor functions. A monitor function works like a critical region, that is no interrupt can occur and the process itself cannot be swapped out. A semaphore can be locked by one process, and is used for preventing processes from simultaneously using resources that can only be used by one process at a time, for example a USART. The `__monitor` keyword assures that the lock operation is atomic; in other words it cannot be interrupted.

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


/* Function to test whether the lock is open, and if so take it.
 * Returns 1 on success and 0 on failure.
 */

__monitor int TryGetLock(void)
{
  if (sTheLock == 0)
  {
    /* Success, nobody has the lock. */

    sTheLock = 1;
    return 1;
  }
  else
  {
    /* Failure, someone else has the lock. */

    return 0;
  }
}


/* Function to unlock the lock.
 * It is only callable by one that has the lock.
 */

__monitor void ReleaseLock(void)
{
  sTheLock = 0;
}


/* Function to take the lock. It will wait until it gets it. */

void GetLock(void)
{
  while (!TryGetLock())
  {
    /* Normally, a sleep instruction is used here. */
  }
}
```

```
/* An example of using the semaphore. */

void MyProgram(void)
{
  GetLock();

  /* Do something here. */

  ReleaseLock();
}
```

Example of implementing a semaphore in C++

In C++, it is common to implement small methods with the intention that they should be inlined. However, the compiler does not support inlining of functions and methods that are declared using the `__monitor` keyword.

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

```
#include <intrinsics.h>

// Class for controlling critical blocks.
class Mutex
{
public:
  Mutex()
  {
    // Get hold of current interrupt state.
    mState = __get_interrupt_state();

    // Disable all interrupts.
    __disable_interrupt();
  }

  ~Mutex()
  {
    // Restore the interrupt state.
    __set_interrupt_state(mState);
  }

private:
  __istate_t mState;
};
```

```
class Tick
{
public:
  // Function to read the tick count safely.
  static long GetTick()
  {
    long t;

    // Enter a critical block.
    {
      Mutex m; // Interrupts are disabled while m is in scope.

      // Get the tick count safely,
      t = smTickCount;
    }
    // and return it.
    return t;
  }

private:
  static volatile long smTickCount;
};

volatile long Tick::smTickCount = 0;

extern void DoStuff();

void MyMain()
{
  static long nextStop = 100;

  if (Tick::GetTick() >= nextStop)
  {
    nextStop += 100;
    DoStuff();
  }
}
```

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 329.
- `--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 211.

For more information about the function inlining optimization, see *Function inlining*, page 214.

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. Further, the linker eliminates duplicate sections and sections that are not required.

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 40.

VENEERS

The RX microcontroller uses veneers when calling a function where the 24-bit offsets do not reach. The veneer introduces code which makes the call successfully reach the destination. This code can be inserted between any caller and called function.

ILINK inserts veneers automatically when they are needed.

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.

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

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

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

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

A section is the smallest linkable unit; but if possible, ILINK can exclude smaller units—section fragments—from the final application. For more information, see *Keeping modules*, page 98, and *Keeping symbols and sections*, page 98.

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. See the chapter *Section reference* for more information about each section.

You can group sections together for placement by using blocks. See *define block directive*, page 375.

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 RX ABI (RX Application Binary Interface) compliant. Be aware that this also might require a compiler utility library from the same vendor.

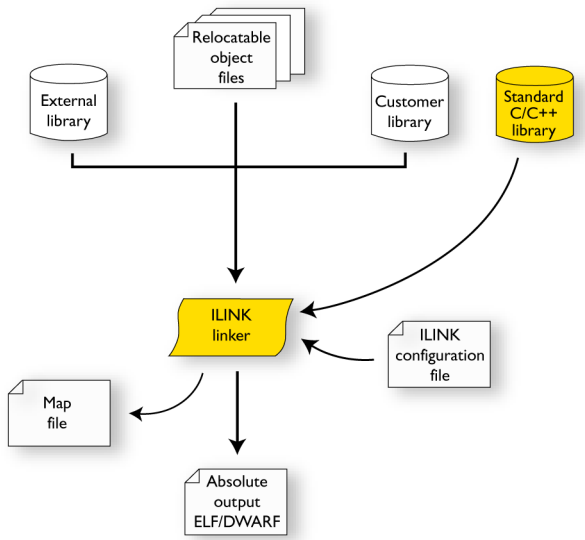
The linker is used for the link process. It normally performs the following procedure (note that some of the steps can be turned off by command line options or by directives in the linker configuration file):

- 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.
- Determine which sections/section fragments from the included modules to include in the application. Only those sections/section fragments that are actually needed by the application are included. There are several ways to determine of which sections/section fragments that are needed, for example, the `__root` object attribute, the `#pragma required` directive, and the `keep` linker directive. In case of duplicate sections, only one is included.
- Where appropriate, arrange for the initialization of initialized variables and code in RAM. The `initialize` directive causes the linker to create extra sections to enable copying from ROM to RAM. Each section that will be initialized by copying is divided into two sections, one for the ROM part and one for the RAM part. If

manual initialization is not used, the linker also arranges for the startup code to perform the initialization.

- Determine where to place each section according to the section placement directives in the *linker configuration file*. Sections that are to be initialized by copying appear twice in the matching against placement directives, once for the ROM part and once for the RAM part, with different attributes.
- 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 messages and logging messages on `stdout` and `stderr`. The log messages are useful for understanding why an application

was linked as it was. For example, why a module or section (or section fragment) was included.

Note: To see the actual content of an ELF object file, use `ielfdumprx`. See *The IAR ELF Dumper—ielfdump*, page 416.

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

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

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

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

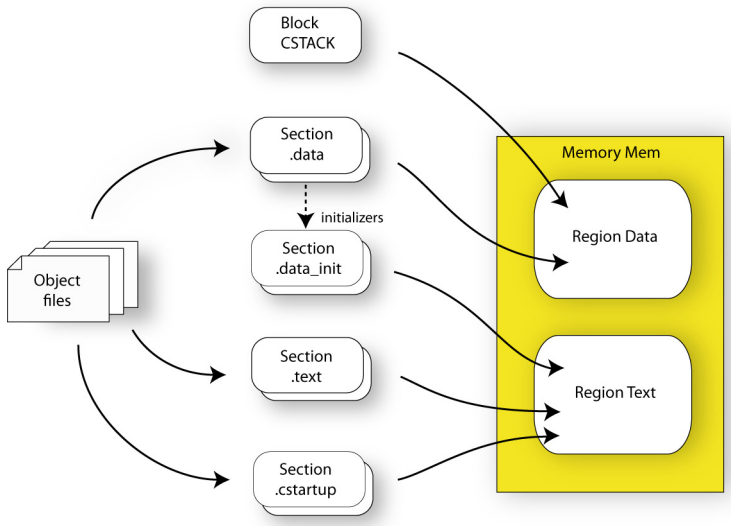
The file then creates an empty block called `STACK` with a size of 4 Kbytes in which the application stack will reside. To create a *block* is the basic method which you can use to get detailed control of placement, size, etc. It can be used for grouping sections, but also as in this example, to specify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

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

The last part of the configuration file handles the actual placement of all the sections into the available regions. First, the startup code—defined to reside in the read-only (`readonly`) object module `cstartup.o`—is placed at the start of the `ROM` region, that is at address `0x10000`. Note that the part within `{}` is referred to as *section selection* and it selects the sections for which the directive should be applied to. Then the rest of the read-only sections are placed in the `ROM` region. Note that the section selection `{ readonly code object cstartup.o }` 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	`.mattr.bss`	None
Zero-initialized data	`int i = 0;`	Read/write data, zero-init	`.mattr.bss`	None
Initialized data (non-zero)	`int i = 6;`	Read/write data	`.mattr.data`	The initializer
Non-initialized data	`__no_init int i;`	Read/write data, zero-init	`.mattr.noinit`	None
Constants	`const int i = 6;`	Read-only data	`.mattr.rodata`	The constant
Code	`__ramfunc void myfunc() {}`	Read/write code	`.textrw`	The code

Table 5: Sections holding initialized data

* The actual memory attribute—*mattr*—used depends on the memory of the variable. For a more information about possible section names, see *Summary of sections*, page 393.

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 are handled automatically by ILINK; they should only be placed in RAM

- Sections that should be initialized, except for zero-initialized sections, should be listed in an `initialize` directive

  Normally during linking, a section that should be initialized is split 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
- Sections holding `__no_init` declared variables should not be initialized and thus should be listed in a `do not initialize` directive. They should also be placed in RAM.

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

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

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

/* 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 378), 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 95.

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 384.

Stack usage analysis

Under the right circumstances, the linker can accurately calculate the maximum stack usage for each call graph root (each function that is not called from another function).

If you enable stack usage analysis (see *--enable_stack_usage*, page 279), 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.

This 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. You can do this by using pragma directives in the source file, or by using a separate stack usage control file when linking.

If you use a stack usage control file (see *--stack_usage_control*, page 290), 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 (see *check that directive*, page 388) in your linker configuration file to check that the stack usage calculated by the linker does not exceed the stack space you have allocated.

LIMITATIONS

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

- The linker might not always be able to identify all functions in object modules that lack stack usage information. In particular this might be a problem with object modules written in assembler or produced by non-IAR tools.
- 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.
- C++ source code that uses exceptions is not supported.
- If you use other forms of function calls, 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 that 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.

STACK USAGE CONTROL FILES

A stack usage control file contains stack usage control directives.

Using stack usage control files, 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`.
- Exclude certain functions from stack usage analysis, by using the stack usage control directive `exclude`.
- Specify the possible destinations for indirect calls in a function, by using the stack usage control directive `possible calls`.
- Specify that functions are call graph roots, including an optional call graph root category, by using the stack usage control directive `call graph root`.
- Specify a maximum recursion depth for a recursion nest (a set of cycles in the call graph with at least one common node).

- 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.

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 using a simple stack usage control file, which might look something like this:

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

For more information, see *call_graph_root*, page 324 and the chapter *The stack usage control file*, page 403.

To comply with the RX ABI, the compiler generates assembler labels for symbol and function names by prefixing an underscore. You must remember to add this extra underscore when you refer to C symbols in the stack usage control file. For example, `main` must be written as `_main`.

SOURCE ANNOTATION

As an alternative to specifying possible calls in a stack usage control file, you can instead annotate the source code.

In C files, at the point of an indirect call, you can use the `#pragma calls` directive to list the possible destinations for that call.

You can also, at the definition of a function, specify that it is a call graph root by using the `#pragma call_graph_root` directive.

SITUATIONS WHERE WARNINGS ARE ISSUED

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

- There is at least one function without stack usage information.
- There is at least one 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 at least one 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.

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.

In this case, the maximum stack depth for the program entry (`__iar_program_start`) is 168 bytes, and occurs inside the system library `printf` function. Public functions are listed by name, while module-local functions also include the name of the module (like `do_something` above).

CHECKING THAT THE STACK IS LARGE ENOUGH

You can use the `check that` directive in your linker configuration file to check that the stack is large enough.

For example, assuming a stack block named `MY_STACK`:

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

When linking, the linker emits an error if the expression is false (zero). In this example there would be an error if the sum of 168 (the maximum stack usage of the program entry), 136 (the sum of the maximum stack usages in category "interrupt"), and 100 (a safety margin) is greater than the size of the `MY_STACK` block.

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

Linking considerations

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

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

CHOOSING A LINKER CONFIGURATION FILE

The `config` directory contains ready-made linker configuration files for all supported devices. 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.

To edit a linker configuration file, use the editor in the IDE, or any other suitable editor.

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

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

DEFINING YOUR OWN MEMORY AREAS

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

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

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

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

Each region definition must be tailored for the actual hardware.

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

Adding an additional region

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

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

Merging different areas into one region

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

```
/* Define the 2nd ROM region to have two areas. The first with
the start address 0x80000 and 128 Kbytes large, and the 2nd with
the start address 0xC0000 and 32 Kbytes large */
define region ROM2 = Mem:[from 0x80000 size 0x20000]
                   | Mem:[from 0xC0000 size 0x08000];
```

or equivalently

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

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 411.

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 80.

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 48.

APPLICATION STARTUP

By default, the point where the application starts execution is defined by the `__iar_program_start` label. The reset vector points to this start label. 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 279.

SETTING UP STACK MEMORY

The sizes of the stack blocks `USTACK` and `ISTACK` are defined in the linker configuration file. To change the allocated amount of memory, change the block definition like this:

```
define block USTACK with size = 0x2000, alignment = 4{ };
```

Specify an appropriate size for your application.

Note: To make it possible to change the stack sizes from the IDE, use the symbols `_ISTACK_SIZE` and `_USTACK_SIZE` instead of the actual size, like this:

```
define block USTACK with size = _USTACK_SIZE, alignment = 4 { };
```

For more information about stack memory, see *Stack considerations*, page 190.

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 = 4{ };
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: To make it possible to change the heap size from the IDE, use the symbol `_HEAP_SIZE` instead of the actual size, like this:

```
define block HEAP with size = _HEAP_SIZE, alignment = 4 { };
```

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 = lzw { readwrite };
```

For more information about the available packing algorithms, see *initialize directive*, page 378.

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 a number of 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 have been mentioned in a `do not initialize` directive. Usually, only `.noinit` sections are specified in a `do not initialize` directive, but you can add any zero-initialized sections you like, and take direct control over when and how these sections are initialized.

Simple copying example with an implicit 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, a synthetic block is created by the linker for those sections.

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 100.

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 100.

If the functions need to run without accessing the flash/ROM, you can use the `__ramfunc` keyword when compiling. See *Executing functions in RAM*, page 69.

Running all code from RAM

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

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

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

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

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

For more information about the available compression algorithms, see *initialize directive*, page 378.

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 196.

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 196.

STANDARD LIBRARY HANDLING

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

To disable the automatic inclusion of the library, use the option `--no_library_search`. In this case, you must explicitly specify every library file to be included. For information about available library files, see *Using prebuilt libraries*, page 109.

PRODUCING OTHER OUTPUT FORMATS THAN ELF/DWARF

ILINK can only produce an output file in the ELF/DWARF format. To convert that format into a format suitable for programming PROM/flash, see *The IAR ELF Tool—ielftool*, page 414.

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 282
- The `--map` option that makes ILINK produce a memory map file, which contains the result of the linker configuration file, see *--map*, page 283.

RELOCATION ERRORS

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

A relocation error produced by ILINK can look like this:

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

The message entries are described in this table:

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

Table 6: Description of a relocation error

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

Table 6: Description of a relocation error (Continued)

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.

The DLIB runtime environment

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

Introduction to the runtime environment

The runtime environment is the environment in which your application executes. The runtime environment depends on the target hardware, the software environment, and the application code.

RUNTIME ENVIRONMENT FUNCTIONALITY

The *runtime environment* supports Standard C and C++, including the standard template library. The runtime environment consists of the *runtime library*, which contains the functions defined by the C and the C++ standards, and include files that define the library interface (the system header files).

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

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

- Support for hardware features:
  - Direct access to low-level processor operations by means of *intrinsic* functions, such as functions for interrupt mask handling
  - Peripheral unit registers and interrupt definitions in include files
- Runtime environment support, that is, startup and exit code and low-level interface to some library functions.
- A floating-point environment (fenv) that contains floating-point arithmetics support, see *fenv.h*, page 365.
- Special compiler support, for instance functions for switch handling or integer arithmetics.

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

SETTING UP THE RUNTIME ENVIRONMENT

The IAR DLIB runtime environment can be used as is together with the debugger. However, to run the application on hardware, you must adapt the runtime environment. Also, to configure the most code-efficient runtime environment, you must determine your application and hardware requirements. The more functionality you need, the larger your code will become.

This is an overview of the steps involved in configuring the most efficient runtime environment for your target hardware:

- Choose which runtime library object file to use

  It is not necessary to specify a library file explicitly, as ILINK automatically uses the correct library file. See *Using prebuilt libraries*, page 109.

- Choose which predefined runtime library configuration to use—Normal or Full

  You can configure the level of support for certain library functionality, for example, locale, file descriptors, and multibyte characters. If you do not specify anything, a default library configuration file that matches the library object file is automatically used. To specify a library configuration explicitly, use the `--dlib_config` compiler option. See *Library configurations*, page 124.

- Optimize the size of the runtime library

  You can specify the formatters used by the functions `printf`, `scanf`, and their variants, see *Choosing formatters for printf and scanf*, page 112.

  You can also specify stack and heap size and placement, see *Setting up stack memory*, page 99, and *Setting up heap memory*, page 99, respectively.

- Include debug support for runtime and I/O debugging

  The library offers support for mechanisms like redirecting standard input and output to the C-SPY Terminal I/O window and accessing files on the host computer, see *Application debug support*, page 114.

- Adapt the library for target hardware

  The library uses a set of low-level functions for handling accesses to your target system. To make these accesses work, you must implement your own version of these functions. For example, to make `printf` write to an LCD display on your board, you must implement a target-adapted version of the low-level function `__write`, so that it can write characters to the display. To customize such functions, you need a good understanding of the library low-level interface, see *Adapting the library for target hardware*, page 116.

- Override library modules

  If you have customized the library functionality, you need to make sure your versions of the library modules are used instead of the default modules. This can be done without rebuilding the entire library, see *Overriding library modules*, page 117.

- Customize system initialization

  It is likely that you need to customize the source code for system initialization, for example, your application might need to initialize memory-mapped special function registers, or omit the default initialization of data sections. You do this by customizing the routine `__low_level_init`, which is executed before the data sections are initialized. See *System startup and termination*, page 119 and *Customizing system initialization*, page 123.

- Configure your own library configuration files

  In addition to the prebuilt library configurations, you can make your own library configuration, but that requires that you *rebuild* the library. This gives you full control of the runtime environment. See *Building and using a customized library*, page 118.

- Manage a multithreaded environment

  In a multithreaded environment, you must adapt the runtime library to treat all library objects according to whether they are global or local to a thread. See *Managing a multithreaded environment*, page 137.

- Check module consistency

  You can use runtime model attributes to ensure that modules are built using compatible settings, see *Checking module consistency*, page 143.

Using prebuilt libraries

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

- Size of the `double` floating-point type
- Byte order for data access
- Support for position-independent code and data
- FPU instructions
- Library configuration—Normal or Full.

The linker will automatically include the correct library object file and library configuration file. To explicitly specify a library configuration, use the `--dlib_config` option. For more information, see *Runtime environment*, page 57.

Note: There are no prebuilt libraries for applications that use 16-bit `int` or *either* ROPI *or* RWPI separately. If you need support for one of these cases, see *Building and using a customized library*, page 118.

LIBRARY FILENAME SYNTAX

The names of the libraries are constructed from these elements:

{*library*}	is `dl` for the IAR DLIB runtime environment
{*core*}	is `rx` for the RXv1 core or `rx2` for the RXv2 core
{*size_of_double*}	is either `f` for 32 bits or `d` for 64 bits
{*size_of_int*}	is `l` for 32 bits
{*byte_order*}	is either `l` for little-endian accesses or `b` for big-endian accesses
{*p-i*}	is `r` for libraries with position-independent code and data (ROPI *and* RWPI)
{*fpu*}	is `s` for libraries without FPU instructions
{*debug_interface*}	is either `n` for No debug I/O output or `d` for Debug I/O output.
{*lib_config*}	is either `n` for the Normal library configuration or `f` for the Full library configuration.

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

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

GROUPS OF LIBRARY FILES

The libraries are delivered in groups of library functions:

Library files for C/C++ standard library functions

These are the functions defined by Standard C and C++, for example functions like `printf` and `scanf`.

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

`dlrx{`*size_of_double*`}l{`*byte_order*`}{`*p-i*`}{`*fpu*`}{`*lib_config*`}.a`

which more specifically means

`dlrx{f|d}l{l|b}{ |r}{ |s}{n|f}.a`

Library files with runtime support

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

`rt{`*`core`*`}{`*`size_of_double`*`}{`*`byte_order`*`}{`*`p-i`*`}.a`

which more specifically means

`rt{rx|rx2}{f|d}{l|b}{ |r}.a`

Library files for C-SPY debugging support

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

`dbgrxl{`*`byte_order`*`}{`*`p-i`*`}{`*`debug_interface`*`}.a`

which more specifically means

`dbgrxl{l|b}{ |r}{n|d}.a`

CUSTOMIZING A PREBUILT LIBRARY WITHOUT REBUILDING

The prebuilt libraries delivered with the compiler can be used as is. However, you can customize parts of a library without rebuilding it.

These items can be customized:

Items that can be customized	Described in
Formatters for printf and scanf	*Choosing formatters for printf and scanf*, page 112
Startup and termination code	*System startup and termination*, page 119
Low-level input and output	*Standard streams for input and output*, page 125
File input and output	*File input and output*, page 129
Low-level environment functions	*Environment interaction*, page 132
Low-level signal functions	*Signal and raise*, page 133
Low-level time functions	*Time*, page 133
Some library math functions	*Math functions*, page 134
Size of heaps, stacks, and sections	*Stack considerations*, page 190 *Heap considerations*, page 190 *Placing code and data—the linker configuration file*, page 83

Table 7: Customizable items

For information about how to override library modules, see *Overriding library modules*, page 117.

Choosing formatters for printf and scanf

The linker automatically chooses an appropriate formatter for `printf`- and `scanf`-related function based on information from the compiler. If that information is missing or insufficient, for example if `printf` is used through a function pointer, if the object file is old, etc, then the automatic choice is the Full formatter. In this case you might want to choose a formatter manually.

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

Note: If you rebuild the library, you can optimize these functions even further, see *Configuration symbols for printf and scanf*, page 127.

CHOOSING A PRINTF FORMATTER

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 in the Standard C/EC++ library.

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

Table 8: Formatters for printf

For information about how to fine-tune the formatting capabilities even further, see *Configuration symbols for printf and scanf*, page 127.

Manually specifying the print formatter in the IDE

To specify a formatter manually, choose **Project>Options** and select the **General Options** category. Select the appropriate option on the **Library options** page.

Manually specifying the printf formatter from the command line

To specify a formatter manually, 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
```

CHOOSING A SCANF FORMATTER

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 in the Standard C/C++ library.

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

Table 9: Formatters for scanf

For information about how to fine-tune the formatting capabilities even further, see *Configuration symbols for printf and scanf*, page 127.

Manually specifying the scanf formatter in the IDE

To specify a formatter manually, choose **Project>Options** and select the **General Options** category. Select the appropriate option on the **Library options** page.

Manually specifying the scanf formatter from the command line

To specify a formatter manually, 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
```

Application debug support

In addition to the tools that generate debug information, there is a debug version of the library low-level interface (typically, I/O handling and basic runtime support). Using the debug library, your application can perform things like opening a file on the host computer and redirecting `stdout` to the debugger Terminal I/O window.

INCLUDING C-SPY DEBUGGING SUPPORT

You can make the library provide debugging support for:

- Handling program abort, exit, and assertions
- I/O handling, which means that `stdin` and `stdout` are redirected to the C-SPY Terminal I/O window, and that it is possible to access files on the host computer during debugging.

In the IDE, choose **Project>Options>Linker**. On the **Library** page, select the **Include C-SPY debugging support** option.

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

Note: If you enable debug information during compilation, this information will be included also in the linker output, unless you use the linker option `--strip`.

THE DEBUG LIBRARY FUNCTIONALITY

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

These capabilities can be valuable during the early development of an application, for example in an application 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. Another use is producing debug printouts.

The mechanism used for implementing this feature works as follows:

The debugger will detect the presence of the function `__DebugBreak`, which will be part of the application if you linked it with the ILINK option for C-SPY debugging support. In this case, the debugger will automatically set a breakpoint at the `__DebugBreak` function. When the application calls, for example, `open`, the `__DebugBreak` function is called, which will cause the application to break and perform the necessary services. The execution will then resume.

THE C-SPY TERMINAL I/O WINDOW

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

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

For more information about the Terminal I/O window, see the *C-SPY® Debugging Guide for RX*.

Speeding up terminal output

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

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

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

```
--redirect ___write=___write_buffered
```

LOW-LEVEL FUNCTIONS IN THE DEBUG LIBRARY

The debug library contains implementations of the following low-level functions:

Function in DLIB low-level interface	Response by C-SPY
`abort`	Notifies that the application has called `abort`
`clock`	Returns the clock on the host computer
`__close`	Closes the associated host file on the host computer
`__exit`	Notifies that the end of the application was reached
`__lseek`	Searches in the associated host file on the host computer
`__open`	Opens a file on the host computer
`__read`	Directs `stdin`, `stdout`, and `stderr` to the Terminal I/O window. All other files will read the associated host file
`remove`	Writes a message to the Debug Log window and returns `-1`
`rename`	Writes a message to the Debug Log window and returns `-1`
`_ReportAssert`	Handles failed asserts
`system`	Writes a message to the Debug Log window and returns `-1`
`time`	Returns the time on the host computer
`__write`	Directs `stdin`, `stdout`, and `stderr` to the Terminal I/O window. All other files will write to the associated host file

Table 10: Functions with special meanings when linked with debug library

Note: You should not use the low-level interface functions prefixed with _ or__ directly in your application. Instead you should use the high-level functions that use these functions. For more information, see *Library low-level interface*, page 117.

Adapting the library for target hardware

The library uses a set of low-level functions for handling accesses to your target system. To make these accesses work, you must implement your own version of these functions. These low-level functions are referred to as the *library low-level interface*.

When you have implemented your low-level interface, you must add your version of these functions to your project. For information about this, see *Overriding library modules*, page 117.

LIBRARY LOW-LEVEL INTERFACE

The library uses a set of low-level functions to communicate with the target system. For example, `printf` and all other standard output functions use the low-level function `__write` to send the actual characters to an output device. Most of the low-level functions, like `__write`, have no implementation. Instead, you must implement them yourself to match your hardware.

However, the library contains a debug version of the library low-level interface, where the low-level functions are implemented so that they interact with the host computer via the debugger, instead of with the target hardware. If you use the debug library, your application can perform tasks like writing to the Terminal I/O window, accessing files on the host computer, getting the time from the host computer, etc. For more information, see *The debug library functionality*, page 114.

Note that your application should not use the low-level functions directly. Instead you should use the corresponding standard library function. For example, to write to `stdout`, you should use standard library functions like `printf` or `puts`, instead of `__write`.

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

The low-level interface is further described in these sections:

- *Standard streams for input and output*, page 125
- *File input and output*, page 129
- *Signal and raise*, page 133
- *Time*, page 133
- *Assert*, page 136.

Overriding library modules

To use a library low-level interface that you have implemented, add it to your application. See *Adapting the library for target hardware*, page 116. Or, you might want to override a default library routine with your customized version. In both cases, follow this procedure:

1. Use a template source file—a library source file or another template—and copy it to your project directory.
2. Modify the file.
3. Add the customized file to your project, like any other source file.

Note: If you have implemented a library low-level interface and added it to a project that you have built with debug support, your low-level functions will be used and not the C-SPY debug support modules. For example, if you replace the debug support module `__write` with your own version, the C-SPY Terminal I/O window will not be supported.

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

Building and using a customized library

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

- There is no prebuilt library for the required combination of compiler options or hardware support, for example, locked registers or 16-bit `int`
- You want to build a library for applications that use *either* ROPI *or* RWPI separately, but not both.
- You want to define your own library configuration with support for locale, file descriptors, multibyte characters, etc.

In those cases, you must:

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

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

For information about the build process and the IAR Command Line Build Utility, see the *IDE Project Management and Building Guide*.

SETTING UP A LIBRARY PROJECT

The IDE provides a library project template which can be used for customizing the runtime environment configuration. This library template uses the Full library configuration, see Table 11, *Library configurations*.

In the IDE, modify the generic options in the created library project to suit your application, see *Basic project configuration*, page 55.

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

MODIFYING THE LIBRARY FUNCTIONALITY

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

The library functionality is determined by a set of *configuration symbols*. The default values of these symbols are defined in the file `DLib_Defaults.h`. This read-only file describes the configuration possibilities. Your library also has its own library configuration file *`libraryname`*`.h`, which sets up that specific library with the required library configuration. For more information, see *Customizing a prebuilt library without rebuilding*, page 111.

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

Modifying the library configuration file

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

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

USING A CUSTOMIZED LIBRARY

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

In the IDE you must do these steps:

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

System startup and termination

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

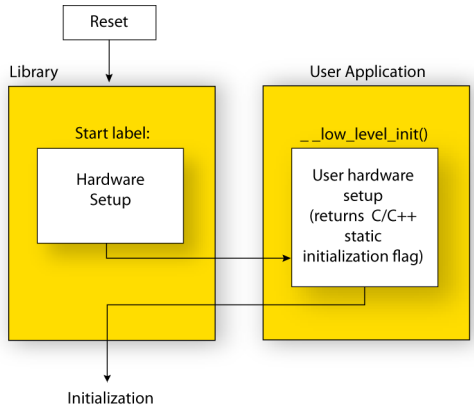
The code for handling startup and termination is located in the source files `cstartup.s`, `cexit.s`, and `low_level_init.c` located in the `rx\src\lib` directory.

For information about how to customize the system startup code, see *Customizing system initialization*, page 123.

SYSTEM STARTUP

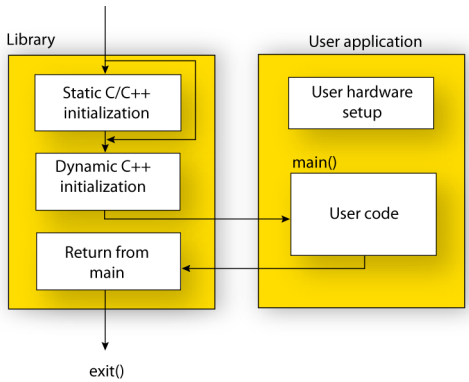
During system startup, an initialization sequence is executed before the `main` function is entered. This sequence performs initializations required for the target hardware and the C/C++ environment.

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.
- The stack pointers, `ISP`, `USP`, and the interrupt vector base, `INTB`, are initialized
- 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:

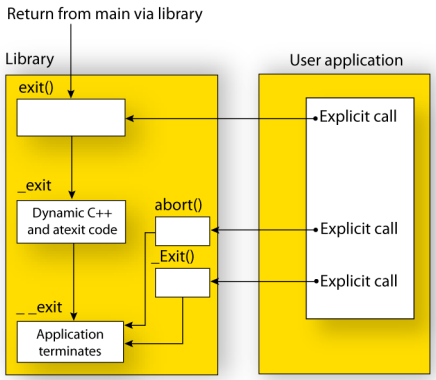


- Static and global variables are initialized. That is, zero-initialized variables are cleared and the values of other initialized variables are copied from ROM to RAM memory. This step is skipped if `__low_level_init` returns zero. For more information, see *Initialization at system startup*, page 86
- 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 50.

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 perform these operations:

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

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

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

C-SPY interface to system termination

If your project is linked with the C-SPY debug library, the normal `__exit` and `abort` functions are replaced with special ones. C-SPY will then recognize when those functions are called and can take appropriate actions to simulate program termination. For more information, see *Application debug support*, page 114.

Customizing system initialization

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

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

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

Note: Normally, you do not need to customize `cexit.s`.

If you intend to rebuild the library, the source files are available in the template library project, see *Building and using a customized library*, page 118.

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

__LOW_LEVEL_INIT

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.

The code calling `__low_level_init` at startup is only included if a module containing a `__low_level_init` definition is included when linking.

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

MODIFYING THE FILE CSTARTUP.S

As noted earlier, you should not modify the file `cstartup.s` if a customized version of `__low_level_init` is enough for your needs. However, if you do need to modify the file `cstartup.s`, we recommend that you follow the general procedure for creating a modified copy of the file and adding it to your project, see *Overriding library modules*, page 117.

Note that you must make sure that the linker uses the start label used in your version of `cstartup.s`. For information about how to change the start label used by the linker, see *--entry*, page 279.

Library configurations

It is possible to configure the level of support for, for example, locale, file descriptors, multibyte characters.

The runtime library configuration is defined in the *library configuration file*. It contains information about what functionality is part of the runtime environment. The configuration file is used for tailoring a build of a runtime library, and tailoring the system header files used when compiling your application. The less functionality you need in the runtime environment, the smaller it becomes.

The library functionality is determined by a set of *configuration symbols*. The default values of these symbols are defined in the file `DLib_Defaults.h`. This read-only file describes the configuration possibilities.

These predefined library configurations are available:

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

Table 11: Library configurations

CHOOSING A RUNTIME CONFIGURATION

To choose a runtime configuration, use one of these methods:

- Default prebuilt configuration—if you do not specify a library configuration explicitly you will get the default configuration. A configuration file that matches the runtime library object file will automatically be used.
- Prebuilt configuration of your choice—to specify a runtime configuration explicitly, use the `--dlib_config` compiler option. See *--dlib_config*, page 247.
- Your own configuration—you can define your own configurations, which means that you must modify the configuration file. Note that the library configuration file describes how a library was built and thus cannot be changed unless you rebuild the

library. For more information, see *Building and using a customized library*, page 118.

The prebuilt libraries are based on the default configurations, see *Library configurations*, page 124.

Standard streams for input and output

Standard communication channels (streams) are defined in `stdio.h`. If any of these streams are used by your application, for example by the functions `printf` and `scanf`, you must customize the low-level functionality to suit your hardware.

There are low-level I/O functions, which are the fundamental functions through which C and C++ performs all character-based I/O. For any character-based I/O to be available, you must provide definitions for these functions using whatever facilities the hardware environment provides. For more information about implementing low-level functions, see *Adapting the library for target hardware*, page 116.

IMPLEMENTING LOW-LEVEL CHARACTER INPUT AND OUTPUT

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

If you intend to rebuild the library, the source files are available in the template library project, see *Building and using a customized library*, page 118. Note that customizing the low-level routines for input and output does not require you to rebuild the library.

Note: If you write your own variants of `__read` or `__write`, special considerations for the C-SPY runtime interface are needed, see *Application debug support*, page 114.

Example of using __write

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 `0x08`:

```
#include <stddef.h>

__no_init volatile unsigned char lcdIO @ 8;

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

  /* Check for the command to flush all handles */
  if (handle == -1)
  {
    return 0;
  }

  /* Check for stdout and stderr
     (only necessary if FILE descriptors are enabled.) */
  if (handle != 1 && handle != 2)
  {
    return -1;
  }

  for (/* Empty */; bufSize > 0; --bufSize)
  {
    lcdIO = *buf;
    ++buf;
    ++nChars;
  }

  return nChars;
}
```

Note: When DLIB calls `__write`, DLIB assumes the following interface: a call to `__write` where `buf` has the value `NULL` is a command to flush the stream. When the handle is `-1`, all streams should be flushed.

Example of using __read

The code in this example uses memory-mapped I/O to read from a keyboard, whose port is assumed to be located at `0x08`:

```
#include <stddef.h>

__no_init volatile unsigned char kbIO @ 8;

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

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

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

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

  return nChars;
}
```

For information about the `@` operator, see *Controlling data and function placement in memory*, page 207.

Configuration symbols for printf and scanf

When you set up your application project, you typically need to consider what `printf` and `scanf` formatting capabilities your application requires, see *Choosing formatters for printf and scanf*, page 112.

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

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

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

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

Table 12: Descriptions of printf configuration symbols

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

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

Table 13: Descriptions of scanf configuration symbols

CUSTOMIZING FORMATTING CAPABILITIES

To customize the formatting capabilities, you must;

1. Set up a library project, see *Building and using a customized library*, page 118.
2. Define the configuration symbols according to your application requirements.

File input and output

The library contains a large number of powerful functions for file I/O operations, 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. Before your application can use the library functions for file I/O operations, you must implement the corresponding low-level function to suit your target hardware. For more information, see *Adapting the library for target hardware*, page 116.

Note that file I/O capability in the library is only supported by libraries with the full library configuration, see *Library configurations*, page 124. In other words, file I/O is supported when the configuration symbol `__DLIB_FILE_DESCRIPTOR` is enabled. If not enabled, functions taking a `FILE *` argument cannot be used.

Template code for these I/O files is included in the product:

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

Table 14: Low-level I/O files

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

Note: If you link your application with I/O debug support, C-SPY variants of the low-level I/O functions are linked for interaction with C-SPY. For more information, see *Application debug support*, page 114.

Locale

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

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

The DLIB library can be used in two main modes:

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

LOCALE SUPPORT IN PREBUILT LIBRARIES

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

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

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

CUSTOMIZING THE LOCALE SUPPORT

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

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

Locale configuration symbols

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

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

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

If you want to customize the locale support, you simply define the locale configuration symbols required by your application. For more information, see *Building and using a customized library*, page 118.

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

Building a library without support for locale interface

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

Building a library with support for locale interface

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

CHANGING LOCALES AT RUNTIME

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

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

`lang_REGION`

or

`lang_REGION.encoding`

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

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

Example

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

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

Environment interaction

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

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

THE GETENV FUNCTION

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

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

```
key=value\0
```

End the string with an extra null character (if you use a C string, this is added automatically). Assign the created sequence of strings to the `__environ` variable.

For example:

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

If you need a more sophisticated environment variable handling, you should implement your own `getenv`, and possibly `putenv` function. This does not require that you rebuild the library. You can find source templates in the files `getenv.c` and `environ.c` in the `rx\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 117.

THE SYSTEM FUNCTION

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

If you decide to rebuild the library, you can find source templates in the library project template. For more information, see *Building and using a customized library*, page 118.

Note: If you link your application with support for I/O debugging, the functions `getenv` and `system` are replaced by C-SPY variants. For more information, see *Application debug support*, page 114.

Signal and raise

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

This does not require that you rebuild the library. You can find source templates in the files `signal.c` and `raise.c` in the `rx\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 117.

If you decide to rebuild the library, you can find source templates in the library project template. For more information, see *Building and using a customized library*, page 118.

Time

To make the `__time32`, `__time64`, 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 366.

To implement these functions does not require that you rebuild the library. You can find source templates in the files `clock.c`, `time.c`, `time64.c`, and `getzone.c` in the `rx\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 117.

If you decide to rebuild the library, you can find source templates in the library project template. For more information, see *Building and using a customized library*, page 118.

The default implementation of `__getzone` specifies UTC (Coordinated Universal Time) as the time zone.

Note: If you link your application with support for I/O debugging, the functions `clock` and `time` are replaced by C-SPY variants that return the host clock and time respectively. For more information, see *Application debug support*, page 114.

Strtod

The function `strtod` does not accept hexadecimal floating-point strings in libraries with the normal library configuration. To make `strtod` accept hexadecimal floating-point strings, you must:

1. Enable the configuration symbol `_DLIB_STRTOD_HEX_FLOAT` in the library configuration file.
2. Rebuild the library, see *Building and using a customized library*, page 118.

Math functions

Some library math functions are also available size-optimized versions, and in more accurate versions.

SMALLER VERSIONS

The functions `cos`, `exp`, `log`, `log10`, `pow`, `sin`, `tan`, and `__iar_Sin` (a help function for `sin` and `cos`) 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.

To use these functions, the default function names must be redirected to these names when linking, using the following options:

```
--redirect _sin=___iar_sin_small
--redirect _cos=___iar_cos_small
--redirect _tan=___iar_tan_small
--redirect _log=___iar_log_small
--redirect _log2=___iar_log2_small
--redirect _log10=___iar_log10_small
--redirect _exp=___iar_exp_small
--redirect _pow=___iar_pow_small
--redirect ___iar_Sin=___iar_Sin_small
--redirect ___iar_Log=___iar_Log_small
```

```
--redirect _sinf=___iar_sin_smallf
--redirect _cosf=___iar_cos_smallf
--redirect _tanf=___iar_tan_smallf
--redirect _logf=___iar_log_smallf
--redirect _log2f=___iar_log2_smallf
--redirect _log10f=___iar_log10_smallf
--redirect _expf=___iar_exp_smallf
--redirect _powf=___iar_pow_smallf
--redirect ___iar_FSin=___iar_Sin_smallf
--redirect ___iar_FLog=___iar_Log_smallf

--redirect _sinl=___iar_sin_smalll
--redirect _cosl=___iar_cos_smalll
--redirect _tanl=___iar_tan_smalll
--redirect _logl=___iar_log_smalll
--redirect _log2l=___iar_log2_smalll
--redirect _log10l=___iar_log10_smalll
--redirect _expl=___iar_exp_smalll
--redirect _powl=___iar_pow_smalll
--redirect ___iar_LSin=___iar_Sin_smalll
--redirect ___iar_LLog=___iar_Log_smalll
```

Note that if you want to redirect any of the functions `sin`, `cos`, or `___iar_Sin`, you must redirect all three functions.

Note that if you want to redirect any of the functions `log`, `log2`, `log10`, or `___iar_Log`, you must redirect all three functions.

MORE ACCURATE VERSIONS

The functions `cos`, `pow`, `sin`, and `tan`, and the help functions `__iar_Sin` and `__iar_Pow` 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.

To use these functions, the default function names must be redirected to these names when linking, using the following options:

```
--redirect _sin=___iar_sin_accurate
--redirect _cos=___iar_cos_accurate
--redirect _tan=___iar_tan_accurate
--redirect _pow=___iar_pow_accurate
--redirect ___iar_Sin=___iar_Sin_accurate
--redirect ___iar_Pow=___iar_Pow_accurate

--redirect _sinf=___iar_sin_accuratef
--redirect _cosf=___iar_cos_accuratef
--redirect _tanf=___iar_tan_accuratef
--redirect _powf=___iar_pow_accuratef
--redirect ___iar_FSin=___iar_Sin_accuratef
--redirect ___iar_FPow=___iar_Pow_accuratef

--redirect _sinl=___iar_sin_accuratel
--redirect _cosl=___iar_cos_accuratel
--redirect _tanl=___iar_tan_accuratel
--redirect _powl=___iar_pow_accuratel
--redirect ___iar_LSin=___iar_Sin_accuratel
--redirect ___iar_LPow=___iar_Pow_accuratel
```

Note that if you want to redirect any of the functions `sin`, `cos`, or `___iar_Sin`, you must redirected all three functions.

Note that if you want to redirect any of the functions `pow` or `___iar_Pow`, you must redirected both functions.

Assert

If you linked your application with the option **Include C-SPY debugging support**, C-SPY will be notified about failed asserts. If this is not the behavior you require, you can add the source file `xreportassert.c` to your application project. The `__ReportAssert` function generates the assert notification. You can find template code in the `rx\src\lib` directory. For more information, see *Overriding library modules*, page 117. To turn off assertions, you must define the symbol `NDEBUG`.

In the IDE, this symbol `NDEBUG` is by default defined in a Release project and *not* defined in a Debug project. If you build from the command line, you must explicitly define the symbol according to your needs. See *NDEBUG*, page 356.

Atexit

The linker allocates a static memory area for `atexit` function calls. By default, the number of calls to the `atexit` function are limited to 32. To change this limit, see *Setting up the atexit limit*, page 100.

Hardware support

Many RX microcontroller devices are equipped with a single-precision hardware floating-point unit (FPU), that supports addition, subtraction, comparison, multiplication, division, and other instructions. For these devices, the compiler will generate code that takes advantage of the FPU.

Managing a multithreaded environment

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).

Prebuilt libraries with multithread support are provided in the product installation. To configure a customized library with multithread support, add the line `#define _DLIB_THREAD_SUPPORT 3` in the library configuration file and rebuild your library

The low-level implementations of locks and TLS are system-specific, and is not included in the DLIB library. 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 LIBRARY

The DLIB library 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 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.
- Dynamically initialized function local objects with static storage duration.

These library objects use TLS:

Library objects using TLS	When these functions are used
Error functions	`errno`, `strerror`
Locale functions	`localeconv`, `setlocale`
Time functions	`asctime`, `localtime`, `gmtime`, `mktime`
Multibyte functions	`mbrlen`, `mbrtowc`, `mbsrtowc`, `mbtowc`, `wcrtomb`, `wcsrtomb`, `wctomb`
Rand functions	`rand`, `srand`
Miscellaneous functions	`atexit`, `strtok`

Table 15: 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 one of the C++ variants is used together with a DLIB library 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 the runtime environment on the command line, for use with threaded applications, use the linker option `--threaded_lib`.

To configure the runtime environment in the IDE for use with threaded applications, choose **Project>Options>General Options>Library Configuration>Enable thread support in the library**. The linker option `--threaded_lib` and the matching prebuilt library with thread support will automatically be used. If one of the C++ variants is used, the IDE will automatically use the compiler option `--guard_calls`.

To complement the built-in multithreaded support in the 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 or redirect the interface to the system locks interface (using the linker option `--redirect`)
- Implement code that handles thread creation, thread destruction, and TLS access methods for the library
- Modify the linker configuration file accordingly.

You can find the required declaration of functions and definitions of macros in the `DLib_Threads.h` file, which is included by `yvals.h`.

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

System locks interface

This interface must be fully implemented for system locks to work:

```
typedef void *__iar_Rmtx;                /* Lock info object */

void __iar_system_Mtxinit(__iar_Rmtx *); /* Initialize a system
                                            lock */
void __iar_system_Mtxdst(__iar_Rmtx *);/*Destroy a system lock */
void __iar_system_Mtxlock(__iar_Rmtx *); /* Lock a system lock */
void __iar_system_Mtxunlock(__iar_Rmtx *); /* Unlock a system
                                              lock */
```

The lock and unlock implementation must survive nested calls.

File streams locks interface

This interface is only needed for the Full library configuration. If file streams are used, they can either be fully implemented or they can be redirected to the system locks interface. This interface must be implemented for file streams locks to work:

```
typedef void *__iar_Rmtx;               /* Lock info object */

void __iar_file_Mtxinit(__iar_Rmtx *);/*Initialize a file lock */
void __iar_file_Mtxdst(__iar_Rmtx *);   /* Destroy a file lock */
void __iar_file_Mtxlock(__iar_Rmtx *);  /* Lock a file lock */
void __iar_file_Mtxunlock(__iar_Rmtx *); /* Unlock a file lock */
```

The lock and unlock implementation must survive nested calls.

DLIB lock usage

The number of locks that the DLIB library assumes exist are:

- `_FOPEN_MAX`—the maximum number of file stream locks. These locks are only used in the Full library configuration, in other words only if both the macro symbols `_DLIB_FILE_DESCRIPTOR` and `_FILE_OP_LOCKS` are true.
- `_MAX_LOCK`—the maximum number of system locks.

Note that even if the application uses fewer locks, the DLIB library will initialize and destroy all of the locks above.

For information about the initialization and destruction code, see `xsyslock.c`.

TLS handling

The DLIB library supports TLS memory areas for two types of threads: the *main thread* (the `main` function including the system startup and exit code) and *secondary threads*.

The main thread's TLS memory area:

- Is automatically created and initialized by your application's startup sequence
- Is automatically destructed by the application's destruct sequence
- Is located in the section `__DLIB_PERTHREAD`
- Exists also for non-threaded applications.

Each secondary thread's TLS memory area:

- Must be manually created and initialized
- Must be manually destructed
- Is located in a manually allocated memory area.

If you need the runtime library to support secondary threads, you must override the function:

```
void *__iar_dlib_perthread_access(void *symbp);
```

The parameter is the address to the TLS variable to be accessed—in the main thread's TLS area—and it should return the address to the symbol in the current TLS area.

Two interfaces can be used for creating and destroying secondary threads. You can use the following interface that allocates a memory area on the heap and initializes it. At deallocation, it destroys the objects in the area and then frees the memory.

```
void *__iar_dlib_perthread_allocate(void);
void  __iar_dlib_perthread_deallocate(void *);
```

Alternatively, if the application handles the TLS allocation, you can use this interface for initializing and destroying the objects in the memory area:

```
void __iar_dlib_perthread_initialize(void *);
void __iar_dlib_perthread_destroy(void *);
```

These macros can be helpful when you implement an interface for creating and destroying secondary threads:

Macro	Description
`__IAR_DLIB_PERTHREAD_SIZE`	The size needed for the TLS memory area.

Table 16: Macros for implementing TLS allocation

Macro	Description
`__IAR_DLIB_PERTHREAD_INIT_SIZE`	The initializer size for the TLS memory area. You should initialize the rest of the TLS memory area, up to `__IAR_DLIB_PERTHREAD_SIZE` to zero.
`__IAR_DLIB_PERTHREAD_SYMBOL_OFFSET(`*`symbolptr`*`)`	The offset to the symbol in the TLS memory area.

Table 16: Macros for implementing TLS allocation (Continued)

Note that the size needed for TLS variables depends on which DLIB resources your application uses.

This is an example of how you can handle threads:

```
#include <yvals.h>

/* A thread's TLS pointer */
void _DLIB_TLS_MEMORY *TLSp;

/* Are we in a secondary thread? */
int InSecondaryThread = 0;

/* Allocate a thread-local TLS memory
   area and store a pointer to it in TLSp. */
void AllocateTLS()
{
  TLSp = __iar_dlib_perthread_allocate();
}

/* Deallocate the thread-local TLS memory area. */
void DeallocateTLS()
{
  __iar_dlib_perthread_deallocate(TLSp);
}

/* Access an object in the
   thread-local TLS memory area. */
void _DLIB_TLS_MEMORY *__iar_dlib_perthread_access(
  void _DLIB_TLS_MEMORY *symbp)
{
  char _DLIB_TLS_MEMORY *p = 0;
  if (InSecondaryThread)
    p = (char _DLIB_TLS_MEMORY *) TLSp;
  else
    p = (char _DLIB_TLS_MEMORY *)
      __segment_begin("__DLIB_PERTHREAD");

  p += __IAR_DLIB_PERTHREAD_SYMBOL_OFFSET(symbp);
  return (void _DLIB_TLS_MEMORY *) p;
}
```

The `TLSp` variable is unique for each thread, and must be exchanged by the RTOS or manually whenever a thread switch occurs.

TLS IN THE LINKER CONFIGURATION FILE

Normally, the linker automatically chooses how to initialize static data. If threads are used, the main thread's TLS memory area must be initialized by plain copying because

the initializers are used for each secondary thread's TLS memory area as well. This is controlled by the following statement in your linker configuration file:

```
initialize by copy with packing = none {section __DLIB_PERTHREAD };
```

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.

For example, in the compiler, it is possible to specify the size of the `double` floating-point type. If you write a routine that only works for 64-bit doubles, it is possible to check that the routine is not used in an application built using 32-bit doubles.

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 17: 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 that key names that start with two underscores are reserved by the compiler. For more information about the syntax, see *rtmodel*, page 336 and the *IAR Assembler User Guide for RX*, respectively.

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

Assembler language interface

- Mixing C and assembler
- Calling assembler routines from C
- Calling assembler routines from C++
- Calling convention
- Assembler instructions used for calling functions
- Memory access methods
- Call frame information

Mixing C and assembler

The IAR C/C++ Compiler for RX provides several ways to access low-level resources:

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

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

INTRINSIC FUNCTIONS

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

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

The advantage of an intrinsic function compared to using inline assembler is that the compiler has all necessary information to interface the sequence properly with register

allocation and variables. The compiler also knows how to optimize functions with such sequences; something the compiler is unable to do with inline assembler sequences. The result is that you get the desired sequence properly integrated in your code, and that the compiler can optimize the result.

For 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 152. The following two are covered in the section *Calling convention*, page 156.

For information about how data in memory is accessed, see *Memory access methods*, page 162.

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 164.

The recommended method for mixing C or C++ and assembler modules is described in *Calling assembler routines from C*, page 152, and *Calling assembler routines from C++*, page 155, respectively.

To comply with the RX ABI, the compiler generates assembler labels for symbol and function names by prefixing an underscore. You must remember to add this extra underscore when you access C symbols from assembler. For example, `main` must be written as `_main`.

Similarly, when referencing an external assembly module from C, an underscore will be added to the symbol used in the C module, so the name of the assembly module must start with an added underscore.

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:

- In big-endian mode, `DC8`, `DC16`, and `DC32` cannot be used because inline assembler will always be in little-endian byte order.
- 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.

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.
- 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 demonstrates the risks of using the `asm` keyword without operands and clobbers:

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

  asm("ADD r1,r0,r0");
  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.

Inline assembler without using operands or clobbered resources is therefore often best avoided.

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])
```

`string` can contain one or more valid assembler instructions or data definition assembler directives, separated by `\n`.

For example:

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

Note that you can define and use local labels in inline assembler instructions.

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 %2,%1,%0 \n"
      : "=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:

```
char matrix[M][N];


void MatrixSetBit(int row)
{
  asm volatile ("bset #1,%0" : : "r" (&matrix[row][0]));
}
```

Operand constraints

Constraint	Description
r	Uses a general purpose register for the expression R1–R15.

Table 18: Inline assembler operand constraints

Constraint	Description
i	An immediate integer operand (one with constant value) is allowed. This includes symbolic constants whose values will be known only at assembly time or later.
Int08	A constant in the range -256 to 255.
Sint08	A constant in the range -128 to 127.
Sint16	A constant in the range -32768 to 32767.
Sint24	A constant in the range -8388608 to 8388607.
Uint04	A constant in the range 0 to 15.

Table 18: Inline assembler operand constraints (Continued)

Constraint modifiers

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

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

Table 19: 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 %[Rm],%[Rn],%[Rd] \n"
      : [Rd]"=r"(sum)
      : [Rn]"r"(term1), [Rm]"r"(term2));
  return sum;
}
```

Operand modifiers

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

Input operands

Input operands cannot have any 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 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("add %2,%1,%0 \n"
      : "=r"(sum)
      : "r"(term1), "r"(term2)
      : "cc");
  return sum;
}
```

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

This table lists valid clobbered resources:

Clobber	Description
`R0-R15`	General purpose registers
`cc`	The condition flags (`C`, `Z`, `S`, and `O`)
`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 20: List of valid clobbers

AN EXAMPLE OF HOW TO USE CLOBBERED MEMORY

```
void MemSet(char *memory, char value, long memorySize )
{
  asm("mov.l %0,R1\n"\
      "mov.l %1,R2\n"\
      "mov.l %2,R3\n"\
      "sstr.b"
      :
      : "r"(memory), "r"(value), "r"(memorySize)
      : "memory", "R1", "R2", "R3");
}
```

Calling assembler routines from C

An assembler routine that will be called from C must:

- Conform to the calling convention
- Have a `PUBLIC` entry-point label

- Be declared as external before any call, to allow type checking and optional promotion of parameters, as in these examples:

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

  or

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

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

CREATING SKELETON CODE

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

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

```
extern int gInt;
extern char gChar;

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

int main()
{
  int locInt = gInt;
  gInt = Func(locInt, gChar);
  return 0;
}
```

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

COMPILING THE 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:

```
iccrx 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, choose **Project>Options>C/C++ Compiler>List** and deselect the suboption **Include call frame information**.

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

The output file

The output file contains the following important information:

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

The `CFI` directives describe the call frame information needed by the Call Stack window in the debugger. For more information, see *Call frame information*, page 164.

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);
}
```

In C++, data structures that only use C features are known as PODs ("plain old data structures"), they use the same memory layout as in C. However, we do not recommend that you access non-PODs from assembler routines.

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.

Note: The calling convention complies with the RX ABI standard.

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 RX CPU registers are divided into three separate sets, which are described in this section.

Scratch registers

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

Any of the registers `R1–R5` or `R14–R15` can be used as a scratch register by the function.

Preserved registers

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

The registers `R6–R13` are preserved registers.

Special registers

The stack pointer register (`R0`) must at all times point to or below the last element on the stack. In the eventuality of an interrupt, everything below the point the stack pointer points to, will be destroyed.

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. 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. The parameters are also passed on the stack in these cases:

- Aggregate types (structures, unions and arrays) larger than 16 bytes, or with a lower alignment than 4
- Unnamed parameters to variable length (variadic) functions; in other words, functions declared as `foo(param1, ...)`, for example `printf`.

Note: Interrupt functions cannot take any parameters.

Hidden parameters

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

If the function returns a structure that does not fit into a register, the memory location where the structure will be stored is passed as the last function parameter.

Hidden parameters are passed in register `R15`.

Register parameters

The registers available for passing parameters are `R1–R4`:

Parameters	Passed in registers
8- to 32-bit values	`R1–R4`
64-bit values	`R2R1`, `R3R2`, `R4R3`
Aggregate values	`R1–R4`

Table 21: Registers used for passing parameters

Small aggregate types are only passed in registers `R1–R4` if they:

- are 16 bytes or smaller
- have an alignment of 4 or more.

Aggregate types that do not fit these two requirements will use a hidden parameter.

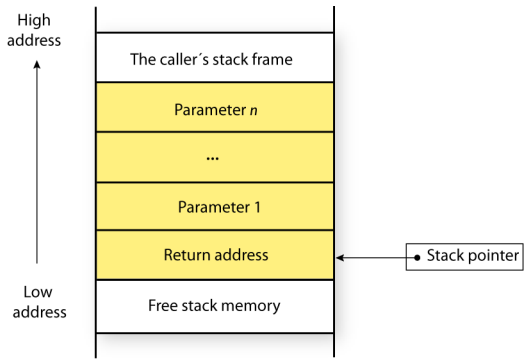
The assignment of registers to parameters is a straightforward process. Traversing the parameters in strict order from left to right, the first parameter is assigned to the

available register or registers. Should there be no suitable register available, the parameter is passed on the stack. This process continues until there are no more parameter registers available or until all parameters have been passed.

Stack parameters and layout

Stack parameters are stored in the main memory, starting at the location pointed to by the stack pointer. Below the stack pointer (toward 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.

This figure illustrates how parameters are stored on the stack:



Objects on the stack should be aligned to 4 bytes at function entry, regardless of their size.

When passed in registers, aggregate types follow the setting of the byte order option `--endian`, but scalar types are always little-endian. On the stack, all parameters are stored according to the byte order setting.

FUNCTION EXIT

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

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

Registers used for returning values

The registers available for returning values are:

Return values	Passed in registers
8- and 16-bit scalars	R1
32-bit values	R1
64-bit values	R2R1
Aggregate values	R1–R4

Table 22: Registers used for returning values

Stack layout at function exit

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

Return address handling

A function written in assembler language should, when finished, return to the caller. At a function call, the return address is stored on the stack.

Typically, a function returns by using the RTS or RTSD instruction.

RESTRICTIONS FOR SPECIAL FUNCTION TYPES

Interrupt functions save all used registers. Task functions save no registers at all, and monitor functions save the interrupt status.

An interrupt function returns by using the RTE instruction. Task functions and monitor functions return by using the RTS or RTSD instruction, depending on whether they need to deallocate a stack frame or not.

EXAMPLES

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

Example 1

Assume this function declaration:

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

This function takes one parameter in the register R1, and the return value is passed back to its caller in the register R1.

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
code
add     #1,R1
rts
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);
```

The calling function must reserve 20 bytes on the top of the stack and copy the contents of the `struct` to that location. The integer parameter `y` is passed in the register `R1`. The return value is passed back to its caller in the register `R1`.

Example 3

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

```
struct MyStruct
{
  int mA;
  int mB;
};

struct MyStruct MyFunction(int x);
```

In this case, the `struct` is small enough to fit in registers, so it is returned in `R2R1`.

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. The parameter `x` is passed in `R1`, and the return value is returned in `R1`.

Assembler instructions used for calling functions

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

Functions can be called in different ways—directly or via a function pointer.

The normal function calling instruction is the `BSR` instruction:

```
bsr label
```

The location that the called function should return to (that is, the location immediately after this instruction) is stored on top of the stack. The destination label cannot be further away than 8 Mbytes. If the destination is further away, the linker will instead call a relay function (veneer) that jumps to the destination. The linker generates veneers automatically if they are needed.

Memory access methods

This section describes the different memory types presented in the chapter *Data storage*. In addition to presenting the assembler code used for accessing data, this section will explain the reason behind the different memory types.

You should be familiar with the RX instruction set, in particular the different addressing modes used by the instructions that can access memory.

For each of the access methods described in the following sections, there are three examples:

- Accessing a global variable
- Accessing a global array using an unknown index
- Accessing a structure using a pointer.

These three examples can be illustrated by this C program:

```
char myVar;
char MyArr[10];

struct MyStruct
{
  long mA;
  char mB;
};

char Foo(int i, struct MyStruct *p)
{
  return myVar + MyArr[i] + p->mB;
}
```

THE DATA16 MEMORY ACCESS METHOD

The data16 memory consists of the highest and the lowest 32 Kbytes of data memory. The code generated by the assembler to access data16 memory becomes slightly smaller. This means a smaller footprint for the application, and faster execution at runtime.

Examples

This example accesses data16 memory:

```
mov.l     #_myVar:16,r3     ; load address of myVar
mov.l     #_MyArr:16,r4     ; load address of MyArr
movu.b    [r1,r4],r1        ; indexed load from MyArr
add       [r3].ub,r1        ; load and add myVar
add       0x4[r2].ub,r1     ; offset p to second
                            ; struct member
```

THE DATA24 MEMORY ACCESS METHOD

The highest and the lowest 8 Mbytes of data memory can be accessed using the data24 memory access method.

Examples

This example accesses data24 memory:

```
mov.l     #_myVar:24,r3     ; load address of myVar
mov.l     #_MyArr:24,r4     ; load address of MyArr
movu.b    [r1,r4],r1        ; indexed load from MyArr
add       [r3].ub,r1        ; load and add myVar
add       0x4[r2].ub,r1     ; offset p to second
                            ; struct member
```

THE DATA32 MEMORY ACCESS METHOD

The data32 memory access method can access the entire data memory range. The data32 memory type uses 4-byte addresses, which can make the code slightly larger.

Examples

This example accesses data32 memory:

```
mov.l     #_myVar:32,r3     ; load address of myVar
mov.l     #_MyArr:32,r4     ; load address of MyArr
movu.b    [r1,r4],r1        ; indexed load from MyArr
add       [r3].ub,r1        ; load and add myVar
add       0x4[r2].ub,r1     ; offset p to second
                            ; struct member
```

THE SBREL MEMORY ACCESS METHOD

The sbrel memory access method is only available when RWPI is enabled, and addressing is relative to the `SB` base register. The actual register that corresponds to the `SB` register depends on the compiler configuration. `SB` is allocated after the ROPI base register and after registers have been locked, and will be the highest numbered available register in the range `R6–R13`. This means that if RWPI is enabled but not ROPI and no registers are locked (using the `--lock` option), the `SB` register is `R13`, and that if both ROPI and RWPI is enabled, `R12` will be used.

The sbrel memory type uses 4-byte addresses, which can make the code slightly larger.

Examples

Assuming that `R12` is the RWPI base register, this example accesses sbrel memory:

```
add       #(_MyArr - __SD_BASE):32,r12,r3  ; compute base
                                ; address of MyArr
movu.b    [r1,r3],r1            ; indexed load from MyArr
add       (_myVar - __SD_BASE):16[r12].ub,r1  ; myVar can
                                ; be accessed directly from
                                ; sbrel
add       0x4[r2].ub,r1         ; offset p to second
                                ; struct member
```

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 RX*.

CFI DIRECTIVES

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

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

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

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

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

Resource	Description
CFA_SP	The call frame of the stack
R1–R15	Normal registers
SP	The stack pointer
?RET32	The return address

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

CREATING ASSEMBLER SOURCE WITH CFI SUPPORT

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

1. Start with suitable C source code, for example:

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

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

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

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

For the source code in this example, the list file looks like this, after it has been cleaned up for increased readability:

```
NAME cfiExample

EXTERN _F

PUBLIC _cfiExample

CFI Names cfiNames0
CFI StackFrame CFA SP DATA
CFI VirtualResource ?RET:32
CFI Resource R1:32, R2:32, R3:32, R4:32,
             R5:32, R6:32, R7:32, R8:32
CFI Resource R9:32, R10:32, R11:32, R12:32,
             R13:32, R14:32, R15:32
CFI Resource SP:32
CFI EndNames cfiNames0
```

```
    CFI Common cfiCommon0 Using cfiNames0
    CFI CodeAlign 1
    CFI DataAlign 1
    CFI ReturnAddress ?RET CODE
    CFI CFA SP+4
    CFI ?RET Frame(CFA, -4)
    CFI R1 Undefined
    CFI R2 Undefined
    CFI R3 Undefined
    CFI R4 Undefined
    CFI R5 Undefined
    CFI R6 SameValue
    CFI R7 SameValue
    CFI R8 SameValue
    CFI R9 SameValue
    CFI R10 SameValue
    CFI R11 SameValue
    CFI R12 SameValue
    CFI R13 SameValue
    CFI R14 Undefined
    CFI R15 Undefined
    CFI EndCommon cfiCommon0

    SECTION .text:CODE:NOROOT(0)
    CFI Block cfiBlock0 Using cfiCommon0
    CFI Function _cfiExample
    CODE
_cfiExample:
    PUSH.L    R6
    CFI R6 Frame(CFA, -8)
    CFI CFA SP+8
    MOV.L     R1,R6
    BSR.A     _F
    ADD       R1,R6
    MOV.L     R6,R1
    RTSD      #0x4,R6,R6
    CFI EndBlock cfiBlock0

    END
```

Note: The header file `cfi.m54` contains the macros `XCFI_NAMES` and `XCFI_COMMON`, 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 RX supports the ISO/IEC 9899:1999 standard (including up to technical corrigendum No.3), also known as C99. In this guide, this standard is referred to as *Standard C* and is the default standard used in the compiler. This standard is stricter than C89.

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.

The C99 standard is derived from C89, but adds features like these:

- The `inline` keyword advises the compiler that the function defined immediately after the keyword should be inlined
- Declarations and statements can be mixed within the same scope
- A declaration in the initialization expression of a `for` loop
- The `bool` data type
- The `long long` data type
- The `complex` floating-point type
- C++ style comments
- Compound literals
- Incomplete arrays at the end of structs
- Hexadecimal floating-point constants
- Designated initializers in structures and arrays
- The preprocessor operator `_Pragma()`
- Variadic macros, which are the preprocessor macro equivalents of `printf` style functions
- VLA (variable length arrays) must be explicitly enabled with the compiler option `--vla`

- Inline assembler using the `asm` or the `__asm` keyword, see *Inline assembler*, page 147.

Note: Even though it is a C99 feature, the IAR C/C++ Compiler for RX does not support UCNs (universal character names).

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 171. 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 very useful in, for example, time-critical routines. The intrinsic functions compile into inline code, either as a single instruction or as a short sequence of instructions. For more information about using intrinsic functions, see *Mixing C and assembler*, page 145. For information about available functions, see the chapter *Intrinsic functions*.

- Library functions

  The IAR DLIB Library provides the C and C++ library definitions that apply to embedded systems. For more information, see *IAR DLIB Library*, page 361.

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 *extensions to Standard C* are enabled, but no *extensions for embedded systems programming*. For information about extensions, see *IAR C language extensions*, page 171.
`-e`	**Standard with IAR extensions**	All *IAR C language extensions* are enabled.

Table 24: Language extensions

* In the IDE, choose **Project>Options>C/C++ Compiler>Language>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 microcontroller 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 174.

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:

- Memory attributes, type attributes, and object attributes

  For information about the related concepts, the general syntax rules, and for reference information, see the chapter *Extended keywords*.

- Placement at an absolute address or in a named 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 207, and *location*, page 331.

- Alignment control

  Each data type has its own alignment; for more information, see *Alignment*, page 293. If you want to change the alignment, the `__packed` data type attribute, the `#pragma pack`, and the `#pragma data_alignment` directives are available. If you want to check the alignment of an object, use the `__ALIGNOF__()` operator.

  The `__ALIGNOF__` operator is used for accessing the alignment of an object. It takes one of two forms:

  - `__ALIGNOF__ (`*`type`*`)`
  - `__ALIGNOF__ (`*`expression`*`)`

  In the second form, the expression is not evaluated.

- Anonymous structs and unions

  C++ includes a feature called anonymous unions. The compiler allows a similar feature for both structs and unions in the C programming language. For more information, see *Anonymous structs and unions*, page 205.

- 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 296.

- `static_assert()`

  The construction `static_assert(`*`const-expression`*`,"`*`message`*`");` can be used in C/C++. The construction will be evaluated at compile time and if *`const-expression`* is false, a message will be issued including the *`message`* string.

- Parameters in variadic macros

  Variadic macros are the preprocessor macro equivalents of `printf` style functions. The preprocessor accepts variadic macros with no arguments, which means if no parameter matches the `...` parameter, the comma is then deleted in the `", ##__VA_ARGS__"` macro definition. According to Standard C, the `...` parameter must be matched with at least one argument.

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. If the section was declared with a memory attribute *`memattr`*, the type of the `__section_begin` operator is a pointer to *`memattr`* `void`. Otherwise, the type is a default 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 25: Section operators and their symbols

Note that 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 __data16 *`.

```
#pragma section="MYSECTION" __data16
...
section_start_address = __section_begin("MYSECTION");
```

See also *section*, page 337, and *location*, page 331.

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 301.

- 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`.

- Repeated `typedef` declarations

  Redeclarations of `typedef` that occur in the same scope are allowed, but a warning is issued.

- Mixing pointer types

  Assignment and pointer difference is allowed between pointers to types that are interchangeable but not identical; for example, `unsigned char *` and `char *`. This includes pointers to integral types of the same size. A warning is issued.

  Assignment of a string constant to a pointer to any kind of character is allowed, and no warning is issued.

- Non-top level `const`

  Assignment of pointers is allowed in cases where the destination type has added type qualifiers that are not at the top level (for example, `int **` to `int const **`). Comparing and taking the difference of such pointers is also allowed.

- Non-`lvalue` arrays

  A non-`lvalue` array expression is converted to a pointer to the first element of the array when it is used.

- Comments at the end of preprocessor directives

  This extension, which makes it legal to place text after preprocessor directives, is enabled unless 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;
}
```

- Expanding function names into strings with the function as context

  Use any of the symbols `__func__` or `__FUNCTION__` inside a function body to make the symbol expand into a string that contains the name of the current function. Use the symbol `__PRETTY_FUNCTION__` to also include the parameter types and return type. The result might, for example, look like this if you use the `__PRETTY_FUNCTION__` symbol:

```
"void func(char)"
```

  These symbols are useful for assertions and other trace utilities and they require that language extensions are enabled, see *-e*, page 249.

- 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. Thus, `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.)

Using C++

- Overview—EC++ and EEC++
- Enabling support for C++
- EC++ feature descriptions
- EEC++ feature description
- C++ language extensions

Overview—EC++ and EEC++

IAR Systems supports the C++ language. You can choose between the industry-standard Embedded C++ and Extended Embedded C++. *Using C++* describes what you need to consider when using the C++ language.

Embedded C++ is a proper subset of the C++ programming language which is intended for embedded systems programming. It was defined by an industry consortium, the Embedded C++ Technical Committee. Performance and portability are particularly important in embedded systems development, which was considered when defining the language. EC++ offers the same object-oriented benefits as C++, but without some features that can increase code size and execution time in ways that are hard to predict.

EMBEDDED C++

These C++ features are supported:

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

C++ features that are excluded are those that introduce overhead in execution time or code size that are beyond the control of the programmer. Also excluded are features added very late before Standard C++ was defined. Embedded C++ thus offers a subset of C++ which is efficient and fully supported by existing development tools.

Embedded C++ lacks these features of C++:

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

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

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

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

EXTENDED EMBEDDED C++

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

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

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

To support Extended EC++, this product includes a version of the standard template library (STL), in other words, the C++ standard chapters utilities, containers, iterators, algorithms, and some numerics. This STL is tailored for use with the Extended EC++

language, which means no exceptions and no support for runtime type information (`rtti`). Moreover, the library is not in the `std` namespace.

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

Enabling support for C++

In the compiler, the default language is C.

To compile files written in Embedded C++, use the `--ec++` compiler option. See *--ec++*, page 249.

To take advantage of *Extended* Embedded C++ features in your source code, use the `--eec++` compiler option. See *--eec++*, page 249.

To enable EC++ or EEC++ in the IDE, choose **Project>Options>C/C++ Compiler>Language** and select the appropriate standard.

EC++ feature descriptions

When you write C++ source code for the IAR C/C++ Compiler for RX, 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, memory, and object attribute.

Member functions are in general treated the same way free functions are, and can have any applicable IAR type, memory, 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.

Example of using attributes with classes

```
class MyClass
{
public:
  // Locate a static variable in __data16 memory at address 60
  static __data16 __no_init int mI @ 60;

  // A static task function
  static __task void F();

  // A task function
  __task void G();

  // A VIRTUALtask function
  virtual __task void H();

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

FUNCTION TYPES

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

Example

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

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

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

void MyG()
{
  MyF(F1);                      // Always works
  MyF(F2);                      // FpCpp is compatible with FpC
}
```

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 119.

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 Embedded C++

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.

TEMPLATES

Extended EC++ supports templates according to the C++ standard, but not the `export` keyword. The implementation uses a two-phase lookup which means that the keyword `typename` must be inserted wherever needed. Furthermore, at each use of a template, the definitions of all possible templates must be visible. This means that the definitions of all templates must be in include files or in the actual source file.

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.

For more information about this, see the *C-SPY® Debugging Guide for RX*.

EEC++ feature description

This section describes features that distinguish Extended EC++ from EC++.

TEMPLATES

The compiler supports templates with the syntax and semantics as defined by Standard C++. However, note that the STL (standard template library) delivered with the product is tailored for Extended EC++, see *Extended Embedded C++*, page 180.

VARIANTS OF CAST OPERATORS

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

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

MUTABLE

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

NAMESPACE

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

THE STD NAMESPACE

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

```
#define std
```

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

C++ language extensions

When you use the compiler in any 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
};
```

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

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

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

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

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

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

Application-related considerations

- Output format considerations
- Stack considerations
- Heap considerations
- Position-independent code and data
- Changing ID code protection and option-setting memory
- Interaction between the tools and your application
- Checksum calculation
- Linker optimizations

Output format considerations

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

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

`ielftool` can produce these output formats:

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

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

The source code for `ielftool` is provided in the `rx/src` directory. For more information about `ielftool`, see *The IAR ELF Tool—ielftool*, page 414.

Stack considerations

To make your application use stack memory efficiently, there are some considerations to be made.

THE USER MODE AND SUPERVISOR MODE STACKS

There are two stacks, the user mode stack and the supervisor mode stack. They are two continuous blocks of memory pointed to by the stack pointer registers `USP` and `ISP`.

The data block used for holding the user mode stack is called `USTACK` and the data block for the supervisor mode stack is called `ISTACK`. The system startup code initializes the stack pointers to the end of the stack blocks.

The processor will be in supervisor mode on power on reset and when processing an interrupt. To enter user mode, special instruction sequences must be executed, as described in the chip manufacturer's documentation.

The startup sequence in `cstartup.s` will remain in supervisor mode when calling the `main` function, so only the `ISTACK` block will be used until the application enters user mode by its own means.

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 99, and *Saving stack space and RAM memory*, page 217.

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:

- Linker sections used for the heap

- Allocating the heap size, see *Setting up heap memory*, page 99.

HEAP SEGMENTS IN DLIB

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

HEAP SIZE AND STANDARD I/O

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

Position-independent code and data

Most applications are designed to be placed at a fixed position in memory. The exact placement of each function and variable is decided at link time. However, sometimes it is useful to instead decide at runtime where to place the application, for example in certain systems where applications are loaded dynamically.

You can configure the compiler to generate read-only position-independent code and data.

ROPI

The term *ROPI* (Read-Only Position-Independent) is a synonym for the Renesas term "PIC/PID", and refers to RX ABI compliant position independence, where the `PID` base register is the static base pointer for accessing constant data as described by the RX ABI. This means that, even though the linker places the code and data at fixed locations, the application can still be executed correctly when the linked image is placed at a different address than where it was linked.

In a system with ROPI applications, there might be a small amount of non-ROPI code that handles startup, dynamic loading of applications, shared firmware functions, etc. Such code must be compiled and linked separately from the ROPI applications.

Note: Only functions and read-only data are affected by ROPI—variables in RAM are only position-independent when RWPI is enabled, see *RWPI*, page 194.

Drawbacks and limitations

There are some drawbacks and limitations to bear in mind when using ROPI:

- The code generated for function calls and accesses to read-only data will be somewhat larger
- Data initialization at startup might be somewhat slower, and the initialization data might be somewhat larger
- Interrupt handlers that use the `#pragma vector` directive are not handled automatically
- The object attribute `__ramfunc` is not supported
- Data pointer constants cannot be initialized with the address of constant data, or a string literal. Writable pointers will be initialized automatically.
- Some C library functions will work differently, mainly because they use RAM instead of ROM for storage (for example the functions for locale support). The C99 functions `erf` and `gamma` are not supported.

Note: In some cases, there is an alternative to ROPI that avoids these limitations and drawbacks: If there is only a small number of possible placements for the application, you can compile the application without ROPI and link it multiple times, once for each possible placement. Then you can choose the correct image at load time. When applicable, this approach makes better use of the available hardware resources than using ROPI.

Creating a static startup module

To execute a ROPI application there must also be a static part, a startup program, because the reset vector must always be static. This program should contain:

- an initialization of the `SB` base register (`R13` by default)
- a jump to the ROPI application start address (relative to the `SB` base register)
- the `NMI` vectors with handlers
- the `__DebugBreak` function, if you want to debug the application using C-SPY
- any functions that should not be part of the ROPI application.

This basic sample program, using the normal C startup code, can be used as a starting point:

```
        section .text:CODE:NOROOT(2)
        public  _main
        extern  _printf
        require _printf
        code
_main:
        mov.l   #my_address,R13
        jmp     R13     ; start the ropi application
        end
```

This will include `__DebugBreak` and initialization code for any library parts that are located in the static part of the application. Add a `require` clause to the program for every additional C library function that should be static, for example `_printf`.

You can compile and link this startup program like a normal application. If you include runtime attributes, you can control which library that is used by the linker. If parts of the C runtime library are included, remember to include the data initialization routines.

If a static function should be visible to the ROPI application (such as the `__DebugBreak` function), you must export it from the linked application using the tool `isymexport`. Do not export symbols that exist in the ROPI module as well, such as `__iar_program_start` or `_main`. Which symbols that are made visible is determined by `show` clauses in the edit file (`show.icf`), for example `show _printf`. The syntax for the export is:

```
isymexport --edit show.icf static.out static.tab
```

If you need to call a static function indirectly from a ROPI module, you must declare it `extern __absolute` for the pointer to be initialized correctly.

Creating the ROPI module

To compile an application with position-independent code and read-only data, use the compiler command line option `--ropi`. The source code cannot contain any initializations that violate the ROPI model, that is, it cannot contain contain any constant data pointers to constant data, as in this example:

```
const char * const msg = "error string"; // cannot be initialized
                                         // in ROPI mode
```

To specify ROPI in the IDE, choose **Project>Options>C/C++ Compiler>Target> Code and read-only data**.

When you link the application, all code and constant data must be placed in a common block, tagged `movable`. For example:

```
define movable block BLK with static base SB, alignment = 4
  { first block BLK16 with maximum size = 64k, fixed order { ro
    code object cstartup.o*, ro section .data16* },
    ro };
```

This example places constant data with the memory attribute `__data16` in the lowest 32 Kbytes of data memory, to allow efficient access to these objects. At the very beginning of the block, the C initialization code is placed. This is only to simplify calling the ROPI module, and is not required.

Interrupt handling

The interrupt vector table and the `INTB` register could be the responsibility of either module, but if the interrupt table is placed in the ROPI module, it must be placed in RAM and initialized at runtime.

Building and debugging the application

Include the symbol table generated by `isymexport` when you link the ROPI module, so that it can call functions in the static module. If the static module uses its own RAM objects, the two applications must be linked with disjoint RAM spaces.

To run the application in C-SPY, use the static program as the main project, and add a C-SPY macro file to this project, that loads the ROPI application image:

```
execUserSetup()
{
    __loadImage("ropi_module_path", offset, 0);
}
```

The *`offset`* parameter to `__loadImage` is the difference between the linked base address of the ROPI module and its final address (*`my_address`* in the static assembler program, see *Creating a static startup module*, page 192). For more information about the `__loadImage` macro, see the *C-SPY® Debugging Guide for RXC-SPY® Debugging Guide for RX*.

You can also load the ROPI application image using the options on the **Project>Options>Debugger>Images** page in the IDE.

RWPI

If more than one application runs concurrently through an operating system, they must share the RAM memory. To make this possible, you can build your applications for position-independent data, RWPI (Read-Write Position Independent). RWPI applies only to data in RAM memory.

An application built for RWPI reserves one machine register for use as a base register for all RAM data accesses. This register is locked and cannot be used for anything else. The memory attribute for this memory area with its static base register is `__sbrel`, and becomes the default memory. To declare variables that are not position-independent, for example shared data and semaphores, use the memory attributes `__data16`, `__data24`, and `__data32`.

The static base register `SB` must be initialized before the RWPI program module is called. By default, the static base register is `R12` if you are also using ROPI and `R13` otherwise.

Limitations

There are some limitations to bear in mind when using RWPI:

- Constant pointers to `__sbrel` objects cannot be used
- `__sbrel` objects cannot be declared `const`.

Note: There are only prebuilt runtime libraries for environments using both ROPI and RWPI. If you use either ROPI or RWPI separately, you must build the library yourself.

Changing ID code protection and option-setting memory

The RX microcontrollers use *ID codes* for boot mode ID code protection and for code protection in the OCD emulator, and *option-setting memory*, a set of processor registers—`OFS0`, `OFS1`, and `MDES`—for selecting the state of the microcontroller after a reset.

For the RXv1 architecture, these symbols are provided as a means to change the default values of the ID codes and processor registers:

```
__ID_BYTES_1_4
__ID_BYTES_5_8
__ID_BYTES_9_12
__ID_BYTES_13_16
__MDES
__OFS0
__OFS1
```

For the RXv2 architecture, these symbols are provided:

```
__OSIS_1
__OSIS_2
__OSIS_3
__OSIS_4
__MDE
__OFS0
__OFS1
__SPCC
```

OVERRIDING THE DEFAULT VALUES

To override the default values for these symbols, use the `#pragma public_equ` directive, for example like this:

```
#pragma public_equ="__ID_BYTES_1_4",0x12345678
```

To see how to do this in assembler language, using the `EQU` directive, study the file `defaults.s` in the `rx\src\lib\rx\` directory.

The easiest way to override the default values is to add a copy of the file `defaults.s` to your project in the IDE, and modify it.

For more information about ID code protection and option-setting memory, see the Renesas *Hardware User's Manual* for your microprocessor. See also *public_equ*, page 335.

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 = 4 {};

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

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

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

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

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

CALCULATING A CHECKSUM

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

Creating a place for the calculated checksum

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

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

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

Note: 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, you can use the linker option `--keep=_checksum` or the linker directive `keep` to force the section to be included.

To place 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 memory Mem with size = 4G;

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

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

define block HEAP      with alignment = 4, size = 16M {};
define block CSTACK    with alignment = 4, size = 16K {};

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

Running ielftool

To calculate the checksum, run `ielftool`:

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

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

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

ADDING A CHECKSUM FUNCTION TO YOUR SOURCE CODE

To check the value of the `ielftool` generated checksum, it must be compared with a checksum that your application calculated. This means that you must add a function for

checksum calculation (that uses the same algorithm as `ielftool`) to your application source code. Your application must also include a call to this function.

A function for checksum calculation

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

```
unsigned short SlowCrc16(unsigned short sum,
                         unsigned char *p,
                         unsigned int len)
{
  while (len--)
  {
    int i;
    unsigned char byte = *(p++);

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

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

Example of checksum calculation

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

```
/* The checksum calculated
 * (note that it is located on address 0x8000)
 */
extern unsigned short const _checksum;


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

  /* Run the checksum algorithm */
  calc = SlowCrc16(0,
                   (unsigned char *) checksumStart,
                   (checksumEnd - checksumStart+1));

  /* Rotate out the answer */
  calc = SlowCrc16(calc, zeros, 2);

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

THINGS TO REMEMBER

When calculating a checksum, you must remember that:

- The checksum must be calculated from the lowest to the highest address for every memory range
- Each memory range must be verified in the same order as defined (ABC is not the same as ACB)
- It is OK to have several ranges for one checksum
- If several checksums are used, you should place them in sections with unique names and use unique symbol names
- If the a slow function variant is used, you must make a final call to the checksum calculation with as many bytes (with the value `0x00`) as there are bytes in the checksum.
- Never calculate a checksum on a location that contains a checksum.

For more information, see also *The IAR ELF Tool—ielftool*, page 414.

C-SPY CONSIDERATIONS

By default, a symbol that you have allocated in memory by using the linker option `--place_holder` is considered by C-SPY to be of the type `int`. If the size of the checksum is 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.

Linker optimizations

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.

Currently, tools from IAR Systems provide the information needed for Virtual Function Elimination in a way that the linker can use.

Note that 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 291 and *--no_vfe*, page 286.

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 auto variables. Stack accesses are cheaper than global accesses, and many auto variables will end up in registers, making execution very fast.
- Use unsigned integer types where possible, unless your application really requires signed values. Many loop optimizations will work much better with unsigned loop variables.
- Try to avoid 64-bit data types, such as 64-bit `double` and `long long`.
- Bitfields with sizes other than 1 bit should be avoided because they will result in inefficient code compared to bit operations.
- Use signed or unsigned `int` for array indexing.
- Using floating-point types without using the built-in floating-point unit is very inefficient, both in terms of code size and execution speed.
- Declaring a pointer to `const` data tells the calling function that the data pointed to will not change, which opens for better optimizations.

For 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 very inefficient, both in terms of code size and execution speed. Thus, you should consider replacing code that uses floating-point operations with code that uses integers, because these are more efficient.

The compiler supports two floating-point formats—32 and 64 bits. The 32-bit floating-point type `float` is more efficient in terms of code size and execution speed. However, the 64-bit format `double` supports higher precision and larger numbers.

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

Unless the application requires the extra precision that 64-bit floating-point numbers give, we recommend using 32-bit floating-point numbers instead.

By default, a *floating-point constant* in the source code is treated as being of the type `double`. This can cause innocent-looking expressions to be evaluated in double precision. In the example below `a` is converted from a `float` to a `double`, the `double` constant `1.0` is added and the result is converted back to a `float`:

```
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 298.

CASTING A FLOATING-POINT VALUE TO AN INTEGER

If you want the result of casting a `float` to an `int` to be a rounded value instead of a truncated value, use the intrinsic function `__ROUND` to insert a `ROUND` instruction directly into the code. See *__ROUND*, page 347.

ALIGNMENT OF ELEMENTS IN A STRUCTURE

The RX microcontroller requires 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 293.

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 334.

ANONYMOUS STRUCTS AND UNIONS

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

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

In the IDE, language extensions are enabled by default.

Use the `-e` compiler option to enable language extensions. See *-e*, page 249, for additional information.

Example

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

```
struct S
{
  char mTag;
  union
  {
    long mL;
    float mF;
  };
} St;

void F(void)
{
  St.mL = 5;
}
```

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

```
__no_init volatile
union
{
  unsigned char IOPORT;
  struct
  {
    unsigned char Way: 1;
    unsigned char Out: 1;
  };
} @ 8;



/* The variables are used here. */
void Test(void)
{
  IOPORT = 0;
  Way = 1;
  Out = 1;
}
```

This declares an I/O register byte `IOPORT` at address 8. The I/O register has 2 bits declared, `Way` and `Out`. Note that both the inner structure and the outer union are anonymous.

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

Controlling data and function placement in memory

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

- Data models

  By selecting a data model, you can control the default memory placement of variables and constants. For more information, see *Data models*, page 64.

- Data memory attributes

  Using IAR-specific keywords or pragma directives, you can override the default placement of variables and constants. For more information, see *Using data memory attributes*, page 61.

- 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. For more information, see *Data placement at an absolute location*, page 207.

- 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 209

DATA PLACEMENT AT AN ABSOLUTE LOCATION

The `@` operator, alternatively the `#pragma location` directive, can be used for placing global and static variables at absolute addresses. The variables must be declared using one of these combinations of keywords:

- `__no_init`
- `__no_init` and `const` (without initializers)

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

Note: All declarations of 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.

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 @ 0x2000;/* OK */
```

The next example contains a `const` declared object which is not initialized. The object is placed in ROM. This is useful for configuration parameters, which are accessible from an external interface.

```
#pragma location=0x2002
__no_init const int beta;                /* OK */

const int gamma @ 0x2004 = 3;            /* OK */
```

The actual value must be set by other means. The typical use is for configurations where the values are loaded to ROM separately, or for special function registers that are read-only.

This shows incorrect usage:

```
int delta @ 0x2006;                /* Error, not __no_init */
__no_init int epsilon @ 0x2007;    /* 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.

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. If no memory attribute is specified, the variable will, like any other variable, be treated as if it is located in the default memory. 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 in the zero and initialized cases arrange for the correct type of initialization for the variable. When placing a `__no_init` variable in a user-defined section, you must add a pattern that matches that section to your `do not initialize`

directive in the linker configuration file. For initialized variables, you can disable the automatic initialization by using the `initialize manually` directive.

As usual, you can use memory attributes to select a memory for the variable. Note that you must as always ensure that the section is placed in the appropriate memory area when linking.

```
__data16 __no_init int alpha @ "MY_DATA16_NOINIT";/* Placed in
                                                     data16*/
```

Examples of placing functions in named sections

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

void g(void) @ "MY_FUNCTIONS"
{
}

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

Controlling compiler optimizations

The compiler performs many transformations on your application to generate the best possible code. Examples of such transformations are storing values in registers instead of memory, removing superfluous code, reordering computations in a more efficient order, and replacing arithmetic operations by cheaper operations.

The linker should also be considered an integral part of the compilation system, because some optimizations are performed by the linker. For instance, all unused functions and variables are removed and not included in the final output.

SCOPE FOR PERFORMED OPTIMIZATIONS

You can decide whether optimizations should be performed on your whole application or on individual files. By default, the same types of optimizations are used for an entire project, but you should consider using different optimization settings for individual files. For example, put code that must execute very quickly into a separate file and compile it for minimal execution time, and the rest of the code for minimal code size. This will give a small program, which is still fast enough where it matters.

You can also exclude individual functions from the performed optimizations. The `#pragma optimize` directive allows you to either lower the optimization level, or specify another type of optimization to be performed. See *optimize*, page 333, 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, cross call, 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 255.

Note: Only one object file is generated, and thus all symbols will be part of that object file.

If the whole application is compiled as one compilation unit, it is very useful to make the compiler also discard unused public functions and variables before the interprocedural optimizations are performed. Doing this limits the scope of the optimizations to functions and variables that are actually used. For more information, see *--discard_unused_publics*, page 247.

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 Code hoisting Register content analysis and optimization Common subexpression elimination Static clustering
High (Balanced)	Same as above, and: Peephole optimization Cross jumping Instruction scheduling (when optimizing for speed) Cross call (when optimizing for size) Loop unrolling Function inlining Code motion Type-based alias analysis

Table 26: Compiler optimization levels

Note: Some of the performed optimizations can be individually enabled or disabled. For more information about these, see *Fine-tuning enabled transformations*, page 213.

A high level of optimization might result in increased compile time, and will most likely also make debugging more difficult, because it is less clear how the generated code relates to the source code. For example, at the low, medium, and high optimization levels, variables do not live through their entire scope, which means processor registers used for storing variables can be reused immediately after they were last used. Due to this, the C-SPY Watch window might not be able to display the value of the variable throughout its scope. At any time, if you experience difficulties when debugging your code, try lowering the optimization level.

SPEED VERSUS SIZE

At the high optimization level, the compiler balances between size and speed optimizations. However, it is possible to fine-tune the optimizations explicitly for either size or speed. They only differ in what thresholds that are used; speed will trade size for speed, whereas size will trade speed for size. Note that one optimization sometimes enables other optimizations to be performed, and an application might in some cases become smaller even when optimizing for speed rather than size.

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.

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
- Cross call
- Instruction scheduling.

Common subexpression elimination

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

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

For more information about the command line option, see *--no_cse*, page 256.

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 260.

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 76.

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 256.

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 259.

Example

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

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

Static clustering

When static clustering is enabled, static and global variables that are defined within the same module are arranged so that variables that are accessed in the same function are stored close to each other. This makes it possible for the compiler to use the same base pointer for several accesses.

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

For more information about the command line option, see *--no_clustering*, page 255.

Cross call

Common code sequences are extracted to local subroutines. This optimization, which is performed at optimization level High, can reduce code size, sometimes dramatically, on behalf of execution time and stack size. The resulting code might however be difficult to debug. This optimization cannot be disabled using the `#pragma optimize` directive.

Note: This option has no effect at optimization levels None, Low, and Medium, unless the option `--do_cross_call` is used.

For more information about related command line options, see *--no_cross_call*, page 256.

Instruction scheduling

The compiler features an instruction scheduler to increase the performance of the generated code. To achieve that goal, the scheduler rearranges the instructions to minimize the number of pipeline stalls emanating from resource conflicts within the microprocessor. Note that not all cores benefit from scheduling. The resulting code might be difficult to debug.

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

For more information about the command line option, see *--no_scheduling*, page 258.

Facilitating good code generation

This section contains hints on how to help the compiler generate good code, for example:

- Using efficient addressing modes
- Helping the compiler optimize
- Generating more useful error message.

WRITING OPTIMIZATION-FRIENDLY SOURCE CODE

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

- Local variables—auto variables and parameters—are preferred over static or global variables. The reason is that the optimizer must assume, for example, that called functions can modify non-local variables. When the life spans for local variables end, the previously occupied memory can then be reused. Globally declared variables will occupy data memory during the whole program execution.
- Avoid taking the address of local variables using the `&` operator. This is inefficient for two main reasons. First, the variable must be placed in memory, and thus cannot be placed in a processor register. This results in larger and slower code. Second, the optimizer can no longer assume that the local variable is unaffected over function calls.
- Module-local variables—variables that are declared static—are preferred over global variables (non-static). Also avoid taking the address of frequently accessed static variables.
- The compiler is capable of inlining functions, see *Function inlining*, page 214. 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 211.

- 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 145.

SAVING STACK SPACE AND RAM MEMORY

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

- If stack space is limited, avoid long call chains and recursive functions.
- Avoid using large non-scalar types, such as structures, as parameters or return type. To save stack space, you should instead pass them as pointers or, in C++, as references.

ALIGNING THE FUNCTION ENTRY POINT

The runtime performance of a function depends on the entry address assigned by the linker. To make the function execution time less dependent on the entry address, the alignment of the function entry point can be specified explicitly using a compiler option, see *--align_func*, page 241. A higher alignment does not necessarily make the function faster, but the execution time will be more predictable.

REGISTER LOCKING

Register locking means that the compiler can be instructed never to touch some processor registers. This can be useful in several situations. For example:

- Some parts of a system could be written in assembler language to improve execution speed. These parts could be given dedicated processor registers.
- The register could be used by an operating system, or by other third-party software.

Registers are locked using the `--lock` compiler option. See *--lock*, page 254.

In general, if two modules are used together in the same application, they should have the same registers locked. The reason is that registers that can be locked could also be used as parameter registers when calling functions. In other words, the calling convention will depend on which registers that are locked.

To ensure that you only link modules with the same registers locked, you can use the `__lockR`*n* runtime model attribute; see *Checking module consistency*, page 143.

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, so it cannot be larger than 255. It also cannot be negative, which means that the integral promoted value can never have the topmost 8 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 sequences of accesses to variables that you do not want to be interrupted, use the `__monitor` keyword. This must be done for both write *and* read sequences, otherwise you might end up reading a partially updated variable. Accessing a small-sized `volatile` variable can be an atomic operation, but you should not rely on it unless you continuously study the compiler output. It is safer to use the `__monitor` keyword to ensure that the sequence is an atomic operation. For more information, see *__monitor*, page 313.

For more information about the `volatile` type qualifier and the rules for accessing volatile objects, see *Declaring objects volatile*, page 303.

ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for several RX 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 `ior5f56108.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;
} @ 8;


/* 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 RX devices. For information about the @ operator, see *Controlling data and function placement in memory*, page 207.

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_INTB(void)
{
  unsigned long value;
  asm("mvfc INTB,%0 ;hej" : "=r"(value));
  return value;
}

static void set_INTB(unsigned long value)
{
  asm("mvtc %0,INTB" : : "r"(value));
}
```

The general purpose register is used for getting and setting the value of the special purpose register `INTB`. The same method can be used also for accessing other special purpose registers and specific instructions.

To read more about inline assembler, see *Inline assembler*, page 147.

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 segment, according to the specified memory keyword.

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.

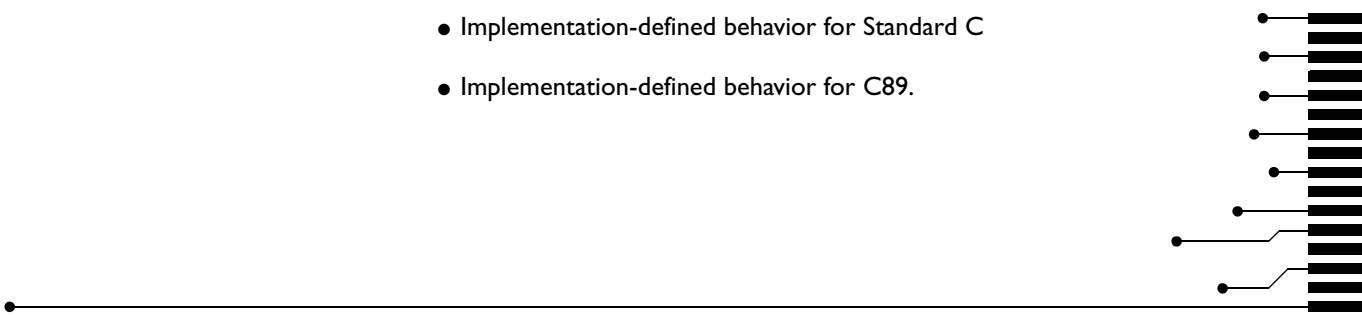
Variables declared using the `__no_init` keyword could, for example, be large input buffers or mapped to special RAM that keeps its content even when the application is turned off.

For more information, see *__no_init*, page 314. Note that to use this keyword, language extensions must be enabled; see *-e*, page 249. For more information, see also *object_attribute*, page 332.

Part 2. Reference information

This part of the *IAR C/C++ Development Guide for RX* contains these chapters:

- External interface details
- Compiler options
- Linker options
- Data representation
- Extended keywords
- Pragma directives
- Intrinsic functions
- The preprocessor
- 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 C89.



External interface details

- Invocation syntax
- Include file search procedure
- Compiler output
- ILINK output
- 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 information about using the build tools from the IDE.

COMPILER INVOCATION SYNTAX

The invocation syntax for the compiler is:

```
iccrx [options] [sourcefile] [options]
```

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

```
iccrx 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:

```
ilinkrx [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:

```
ilinkrx 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 `iccrx` or `ilinkrx` commands; see *Invocation syntax*, page 225.
- Via environment variables

  The compiler and linker automatically append the value of the environment variables to every command line; see *Environment variables*, page 227.
- Via a text file, using the `-f` option; see *-f*, page 251.

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:\program files\iar systems\embedded workbench 6.*n*\rx\inc;c:\headers
QCCRX	Specifies command line options; for example: QCCRX=-lA asm.lst

Table 27: Compiler environment variables

This environment variable can be used with ILINK:

Environment variable	Description
ILINKRX_CMD_LINE	Specifies command line options; for example: ILINKRX_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:

- 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 252.

  2 The directories specified using the `C_INCLUDE` environment variable, if any; see *Environment variables*, page 227.

  3 The automatically set up library system include directories. See *--dlib_config*, page 247.

- 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:

```
iccrx ..\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 information about the syntax for including header files, see *Overview of the preprocessor*, page 351.

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 253. By default, these files will have the filename extension `lst`.
- Optional preprocessor output files

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

- Diagnostic messages

  Diagnostic messages are directed to the standard error stream and displayed on the screen, and printed in an optional list file. For more information about diagnostic messages, see *Diagnostics*, page 230.

- 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 229.

- 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 283. 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 230.

- Error return codes

  ILINK returns status information to the operating system which can be tested in a batch file, see *Error return codes*, page 229.

- Size information about used memory and amount of time

  Information about the generated amount of bytes for functions and data for each memory is directed to `stdout` and displayed on the screen.

Diagnostics

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

MESSAGE FORMAT FOR THE COMPILER

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

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

with these elements:

filename	The name of the source file in which the issue was encountered
linenumber	The line number at which the compiler detected the issue
level	The level of seriousness of the issue
tag	A unique tag that identifies the diagnostic message
message	An explanation, possibly several lines long

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

Use the option `--diagnostics_tables` to list all possible compiler diagnostic messages.

MESSAGE FORMAT FOR THE LINKER

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

level[*tag*]: *message*

with these elements:

level	The level of seriousness of the issue
tag	A unique tag that identifies the diagnostic message
message	An explanation, possibly several lines long

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

Use the option `--diagnostics_tables` to list all possible linker diagnostic messages.

SEVERITY LEVELS

The diagnostic messages are divided into different levels of severity:

Remark

A diagnostic message that is produced when the compiler or linker finds a construct that can possibly lead to erroneous behavior in the generated code. Remarks are by default not issued, but can be enabled, see *--remarks*, page 265.

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 260.

Error

A diagnostic message that is produced when the compiler or linker finds a serious error. An error will produce a non-zero exit code.

Fatal error

A diagnostic message that is produced when the compiler finds a condition that not only prevents code generation, but which makes further processing pointless. After the message is issued, compilation terminates. A fatal error will produce a non-zero exit code.

SETTING THE SEVERITY LEVEL

The diagnostic messages can be suppressed or the severity level can be changed for all diagnostics messages, except for fatal errors and some of the regular errors.

See *Summary of compiler options*, page 237, for information about the compiler options that are available for setting severity levels.

For the compiler see also the chapter *Pragma directives*, for information about the pragma directives that are available for setting severity levels.

INTERNAL ERROR

An internal error is a diagnostic message that signals that there was a serious and unexpected failure due to a fault in the compiler or linker. It is produced using this form:

```
Internal error: message
```

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

- The product name
- The version number of the compiler or of ILINK, which can be seen in the header of the list or map files generated by the compiler or by ILINK, respectively
- Your license number
- The exact internal error message text
- The files involved of the application that generated the internal error

- A list of the options that were used when the internal error occurred.

Compiler options

- 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 226.

RULES FOR SPECIFYING PARAMETERS

There are some general syntax rules for specifying option parameters. First, the rules depending on whether the parameter is *optional* or *mandatory*, and whether the option has a short or a long name, are described. Then, the rules for specifying filenames and directories are listed. Finally, the remaining rules are listed.

Rules for optional parameters

For options with a short name and an optional parameter, any parameter should be specified without a preceding space, for example:

`-O` or `-Oh`

For options with a long name and an optional parameter, any parameter should be specified with a preceding equal sign (=), for example:

```
--misrac2004=n
```

Rules for mandatory parameters

For options with a short name and a mandatory parameter, the parameter can be specified either with or without a preceding space, for example:

`-I..\src` or `-I ..\src\`

For options with a long name and a mandatory parameter, the parameter can be specified either with a preceding equal sign (=) or with a preceding space, for example:

```
--diagnostics_tables=MyDiagnostics.lst
```

or

```
--diagnostics_tables MyDiagnostics.lst
```

Rules for options with both optional and mandatory parameters

For options taking both optional and mandatory parameters, the rules for specifying the parameters are:

- For short options, optional parameters are specified without a preceding space
- For long options, optional parameters are specified with a preceding equal sign (=)
- For short and long options, mandatory parameters are specified with a preceding space.

For example, a short option with an optional parameter followed by a mandatory parameter:

```
-lA MyList.lst
```

For example, a long option with an optional parameter followed by a mandatory parameter:

```
--preprocess=n PreprocOutput.lst
```

Rules for specifying a filename or directory as parameters

These rules apply for options taking a filename or directory as parameters:

- Options that take a filename as a parameter can optionally 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\`:

  ```
  iccrx 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:

  ```
  iccrx 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:

  ```
  iccrx 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:

  ```
  iccrx 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`:

  ```
  iccrx 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
`--align_func`	Specifies the alignment of function entry points
`--c89`	Specifies the C89 dialect
`--char_is_signed`	Treats `char` as signed
`--char_is_unsigned`	Treats `char` as unsigned
`--core`	Specifies a CPU core

Table 30: Compiler options summary

Command line option	Description
-D	Defines preprocessor symbols
--data_model	Specifies the data model
--debug	Generates debug information
--dependencies	Lists file dependencies
--diag_error	Treats these as errors
--diag_remark	Treats these as remarks
--diag_suppress	Suppresses these diagnostics
--diag_warning	Treats these as warnings
--diagnostics_tables	Lists all diagnostic messages
--discard_unused_publics	Discards unused public symbols
--dlib_config	Uses the system include files for the DLIB library and determines which configuration of the library to use
--double	Forces the compiler to use 32-bit or 64-bit doubles
-e	Enables language extensions
--ec++	Specifies Embedded C++
--eec++	Specifies Extended Embedded C++
--enable_multibytes	Enables support for multibyte characters in source files
--enable_restrict	Enables the Standard C keyword restrict
--endian	Specifies the byte order of the generated code and data
--error_limit	Specifies the allowed number of errors before compilation stops
-f	Extends the command line
--guard_calls	Enables guards for function static variable initialization
--header_context	Lists all referred source files and header files
-I	Specifies include file path
--int	Specifies the size of the data type int
-l	Creates a list file
--lock	Locks registers

Table 30: Compiler options summary (Continued)

Command line option	Description
`--macro_positions_in _diagnostics`	Obtains positions inside macros in diagnostic messages
`--mfc`	Enables multi-file compilation
`--misrac`	Enables error messages specific to MISRA-C:1998. This option is a synonym of `--misrac1998` and is only available for backwards compatibility.
`--misrac1998`	Enables error messages specific to MISRA-C:1998. See the *IAR Embedded Workbench® MISRA C:1998 Reference Guide*.
`--misrac2004`	Enables error messages specific to MISRA-C:2004. See the *IAR Embedded Workbench® MISRA C:2004 Reference Guide*.
`--misrac_verbose`	*IAR Embedded Workbench® MISRA C:1998 Reference Guide* or the *IAR Embedded Workbench® MISRA C:2004 Reference Guide*.
`--no_clustering`	Disables static clustering optimizations
`--no_code_motion`	Disables code motion optimization
`--no_cross_call`	Disables cross-call optimization
`--no_cse`	Disables common subexpression elimination
`--no_fpu`	Uses call library routines instead of FPU instructions to perform floating-point arithmetic.
`--no_fragments`	Disables section fragment handling
`--no_inline`	Disables function inlining
`--no_path_in_file_macros`	Removes the path from the return value of the symbols `__FILE__` and `__BASE_FILE__`
`--no_scheduling`	Disables the instruction scheduler
`--no_shattering`	Disables variable shattering.
`--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

Table 30: Compiler options summary (Continued)

Command line option	Description
`--no_unroll`	Disables loop unrolling
`--no_warnings`	Disables all warnings
`--no_wrap_diagnostics`	Disables wrapping of diagnostic messages
`-O`	Sets the optimization level
`-o`	Sets the object filename. Alias for `--output`.
`--only_stdout`	Uses standard output only
`--output`	Sets the object filename
`--patch`	Generates code that does not trigger some known hardware-related problems for a specific device group.
`--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 position-independent references to access code and read-only data.
`--rwpi`	Generates code that uses an offset from the static base register to address-writable data.
`--save_acc`	Saves and restores the DSP accumulator when the interrupt context switches.
`--silent`	Sets silent operation
`--sqrt_must_set_errno`	Disables replacing calls to the library function `sqrtf` with the RXv2 core instruction `FSQRT`.
`--strict`	Checks for strict compliance with Standard C/C++
`--suppress_core_attribute`	Disables generation of the runtime attribute `__core`
`--system_include_dir`	Specifies the path for system include files
`--use_c++_inline`	Uses C++ inline semantics in C99

Table 30: Compiler options summary (Continued)

Command line option	Description
`--use_unix_directory_separators`	Uses / as directory separator in paths
`--vla`	Enables C99 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.

Note that if you use the options page **Extra Options** to specify specific command line options, the IDE does not perform an instant check for consistency problems like conflicting options, duplication of options, or use of irrelevant options.

--align_func

Syntax

```
--align_func={1|2|4|8}
```

Parameters

`1` (default)	Sets the alignment of function entry points to 1 byte
`2`	Sets the alignment of function entry points to 2 bytes
`4`	Sets the alignment of function entry points to 4 bytes
`8`	Sets the alignment of function entry points to 8 bytes

Description

Use this option to specify the alignment of the function entry points.

See also

Aligning the function entry point, page 217.

Project>Options>C/C++ Compiler>Align functions

--c89

Syntax `--c89`

Description Use this option to enable the C89 C dialect instead of Standard C.

Note: This option is mandatory when the MISRA C checking is enabled.

See also *C language overview*, page 169.

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

--core

Syntax `--core={rxv1|rxv2}`

Parameters

`rxv1` (default)	Generates code for the RXv1 architecture. This includes the RX100, RX200, and RX600 1st generation families.
`rxv2`	Generates code for the RXv2 architecture. This includes the RX600 2nd generation and future families.

Description

Use this option to select which RX architecture to generate code for.

Project>Options>General Options>Target>Device

-D

Syntax

`-D symbol[=value]`

Parameters

`symbol`	The name of the preprocessor symbol
`value`	The value of the preprocessor symbol

Description

Use this option to define a preprocessor symbol. If no value is specified, `1` is used. This option can be used one or more times on the command line.

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

`-Dsymbol`

is equivalent to:

`#define symbol 1`

To get the equivalence of:

```
#define FOO
```

specify the = sign but nothing after, for example:

```
-DFOO=
```

Project>Options>C/C++ Compiler>Preprocessor>Defined symbols

--data_model

Syntax

`--data_model={near|far|huge}`

Parameters

`near`	Places variables and constant data in the lowest or highest 32 Kbytes of memory.
`far` (default)	Places variables and constant data in the lowest or highest 8 Mbytes of memory.
`huge`	Places variables and constant data anywhere in memory.

Description

Use this option to select the data model, which means a default placement of data objects. If you do not select a data model, the compiler uses the default data model. Note that all modules of your application must use the same data model.

See also

Data models, page 64.

Project>Options>General Options>Target>Data model

--debug, -r

Syntax

```
--debug
-r
```

Description

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

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

Project>Options>C/C++ Compiler>Output>Generate debug information

--dependencies

Syntax

```
--dependencies[=[i|m]] {filename|directory}
```

Parameters

`i` (default)	Lists only the names of files
`m`	Lists in makefile style

See also *Rules for specifying a filename or directory as parameters*, page 236.

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.

--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 236.

Description

Use this option to list all possible diagnostic messages in a named file. This can be convenient, for example, if you have used a pragma directive to suppress or change the severity level of any diagnostic messages, but forgot to document why.

This option cannot be given together with other options.

This option is not available in the IDE.

--discard_unused_publics

Syntax

```
--discard_unused_publics
```

Description

Use this option to discard unused public functions and variables 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 255 and *Multi-file compilation units*, page 211.

Project>Options>C/C++ Compiler>Discard unused publics

--dlib_config

Syntax

```
--dlib_config filename.h|config
```

Parameters

filename	A DLIB configuration header file, see *Rules for specifying a filename or directory as parameters*, page 236.
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.

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.

All prebuilt runtime libraries are delivered with corresponding configuration files. You can find the library object files and the library configuration files in the directory `rx\lib`. For examples and information about prebuilt runtime libraries, see *Using prebuilt libraries*, page 109.

If you build your own customized runtime library, you should also create a corresponding customized library configuration file, which must be specified to the compiler. For more information, see *Building and using a customized library*, page 118.

To set related options, choose:

Project>Options>General Options>Library Configuration

--double

Syntax

```
--double={32|64}
```

Parameters

`32` (default)	32-bit doubles are used
`64`	64-bit doubles are used

Description

Use this option to select the precision used by the compiler for representing the floating-point types `double` and `long double`. The compiler can use either 32-bit or 64-bit precision. By default, the compiler uses 32-bit precision.

See also

Basic data types—floating-point types, page 298.

Project>Options>General Options>Target>Size of type 'double'

-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 171.

Project>Options>C/C++ Compiler>Language 1>Standard with IAR extensions

Note: By default, this option is selected in the IDE.

--ec++

Syntax

`--ec++`

Description

In the compiler, the default language is C. If you use Embedded C++, you must use this option to set the language the compiler uses to Embedded C++.

Project>Options>C/C++ Compiler>Language 1>C++

and

Project>Options>C/C++ Compiler>Language 1>C++ dialect>Embedded C++

--eec++

Syntax

`--eec++`

Description

In the compiler, the default language is C. If you take advantage of Extended Embedded C++ features like namespaces or the standard template library in your source code, you must use this option to set the language the compiler uses to Extended Embedded C++.

See also

Extended Embedded C++, page 180.

Project>Options>C/C++ Compiler>Language 1>C++

and

Project>Options>C/C++ Compiler>Language 1>C++ dialect>Extended Embedded C++

--enable_multibytes

Syntax

```
--enable_multibytes
```

Description

By default, multibyte characters cannot be used in C or C++ source code. Use this option to make multibyte characters in the source code be interpreted according to the host computer's default setting for multibyte support.

Multibyte characters are allowed in C and C++ style comments, in string literals, and in character constants. They are transferred untouched to the generated code.

Project>Options>C/C++ Compiler>Language 2>Enable multibyte support

--enable_restrict

Syntax

```
--enable_restrict
```

Description

Enables the Standard C keyword `restrict`. 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 for data
`little, l` (default)	Specifies little-endian as the default byte order for data

Description

Use this option to specify the byte order of the generated data. By default, the compiler generates data in little-endian byte order. (Code is always little-endian.).

See also

Byte order, page 294.

Project>Options>General Options>Target>Byte order

--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 236.

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.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--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 137.

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 227.

Project>Options>C/C++ Compiler>Preprocessor>Additional include directories

--int

Syntax `--int={16|32}`

Parameters

`16`	The size of the data type `int` is 16 bits. This might be useful if you are migrating code written for another microcontroller than RX.
`32` (default)	The size of the data type `int` is 32 bits. This is the native `int` size for the RX microcontroller.

Description

Use this option to select whether the compiler uses 16 or 32 bits to represent the `int` data type. By default, 32 bits are used. Selecting 16 bits results in larger code size.

Note: There are no prebuilt libraries for 16-bit `int`.

See also

Basic data types—integer types, page 294.

Project>Options>General Options>Target>Size of type 'int'

-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 236.

Description

Use this option to generate an assembler or C/C++ listing to a file. Note that this option can be used one or more times on the command line.

To set related options, choose:

Project>Options>C/C++ Compiler>List

--lock

Syntax

```
--lock={Ri|Rj,Rk|Rm-Rp}
```

Parameters

`Ri|Rj,Rk|Rm-Rp`	The register(s) to lock

Description

Use this option to lock one or several of the registers `R8`–`R13` so that they cannot be used by the compiler but can be used for global register variables. To maintain module consistency, make sure you lock the same registers in all modules. By default, no registers are locked.

Example

```
--lock=R10

--lock=R8,R12,R13

--lock=R10-R13
--lock=R8,R11-R13
```

See also

Register locking, page 217.

Project>Options>C/C++ Compiler>Code>Lock registers

--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**.

--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

```
iccrx myfile1.c myfile2.c myfile3.c --mfc
```

See also

--discard_unused_publics, page 247, *--output, -o*, page 262, and *Multi-file compilation units*, page 211.

Project>Options>C/C++ Compiler>Multi-file compilation

--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 215.

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 214.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Code motion

--no_cross_call

Syntax `--no_cross_call`

Description Use this option to disable the cross-call optimization.

Note: This option has no effect at optimization levels below High, or when optimizing Balanced or for Speed, because cross-call optimization is not enabled then.

See also *Cross call*, page 215.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Cross call

--no_cse

Syntax `--no_cse`

Description Use this option to disable common subexpression elimination.

Note: This option has no effect at optimization levels below Medium.

See also *Common subexpression elimination*, page 213.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Common subexpression elimination

--no_fpu

Syntax `--no_fpu`

Description

Use this option to tell the compiler not to use FPU instructions when performing floating-point arithmetic, but instead call library routines.

This option is set automatically when you choose:

Project>Options>General Options>Target>Device

--no_fragments

Syntax

```
--no_fragments
```

Description

Use this option to disable section fragment handling. Normally, the toolset uses IAR proprietary information for transferring section fragment information to the linker. The linker uses this information to remove unused code and data, and thus further minimize the size of the executable image. When you use this option, this information is not output in the object files.

See also

Keeping symbols and sections, page 98.

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 76.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Function inlining

--no_path_in_file_macros

Syntax

```
--no_path_in_file_macros
```

Description

Use this option to exclude the path from the return value of the predefined preprocessor symbols `__FILE__` and `__BASE_FILE__`.

See also

Description of predefined preprocessor symbols, page 352.

This option is not available in the IDE.

--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 216.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Instruction scheduling

--no_shattering

Syntax

```
--no_shattering
```

Description

Use this option to disable variable shattering. The compiler uses this feature to break up auto variables, which increases the performance of the generated code.

Note: This option has no effect at optimization levels below Medium.

This option is not available in the IDE.

--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 213.

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 100.

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 247, and *--system_include_dir*, page 268.

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 214.

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_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 213.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Loop unrolling

--no_warnings

Syntax

```
--no_warnings
```

Description

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

This option is not available in the IDE.

--no_wrap_diagnostics

Syntax

```
--no_wrap_diagnostics
```

Description

By default, long lines in diagnostic messages are broken into several lines to make the message easier to read. Use this option to disable line wrapping of diagnostic messages.

This option is not available in the IDE.

-O

Syntax

```
-O[n|l|m|h|hs|hz]
```

Parameters

`n`	**None* (Best debug support)**
`l` (default)	Low*
`m`	Medium
`h`	High, balanced
`hs`	High, favoring speed
`hz`	High, favoring size

*The most important difference between None and Low is that at None, all non-static variables will live during their entire scope.

Description

Use this option to set the optimization level to be used by the compiler when optimizing the code. If no optimization option is specified, the optimization level Low is used by default. If only `-O` is used without any parameter, the optimization level High balanced is used.

A low level of optimization makes it relatively easy to follow the program flow in the debugger, and, conversely, a high level of optimization makes it relatively hard.

See also

Controlling compiler optimizations, page 210.

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`) also for messages that are normally directed to the error output stream (`stderr`).

This option is not available in the IDE.

--output, -o

Syntax

```
--output {filename|directory}
-o {filename|directory}
```

Parameters

See *Rules for specifying a filename or directory as parameters*, page 236.

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.

--patch

Syntax

```
--patch=rx610
```

Description

Use this option to avoid a problem with a specific CPU type. Specifying `--patch=rx610` stops the compiler from using the `MVTIPL` instruction (which causes a problem in the RX610 group) in the generated code.

This option is not available in the IDE.

--predef_macros

Syntax

```
--predef_macros {filename|directory}
```

Parameters

See *Rules for specifying a filename or directory as parameters*, page 236.

Description

Use this option to list the predefined symbols. When using this option, make sure to also use the same options as for the rest of your project.

If a filename is specified, the compiler stores the output in that file. If a directory is specified, the compiler stores the output in that directory, in a file with the `predef` filename extension.

Note that this option requires that you specify a source file on the command line.

This option is not available in the IDE.

--preinclude

Syntax

```
--preinclude includefile
```

Parameters

See *Rules for specifying a filename or directory as parameters*, page 236.

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][l]] {filename|directory}
```

Parameters

`c`	Preserve comments
`n`	Preprocess only
`l`	Generate `#line` directives

See also *Rules for specifying a filename or directory as parameters*, page 236.

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.

Using this option also means that the compiler will not differentiate between a cast from `float` to `signed long` and a cast from `float` to `unsigned long`. Some range is lost, but the compiler can use the hardware FPU instead of having to call a library function.

Note that 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 231.

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

--rwpi

Syntax

```
--rwpi
```

Description

Use this option to enable position-independent writable data. A base register will be locked to hold the base address of the position-independent block.

When this option is used, these limitations apply:

- Constant pointers to `__sbrel` objects cannot be used
- `__sbrel` objects cannot be declared `const`.

See also

RWPI, page 194 and *Description of predefined preprocessor symbols*, page 352.

Project>Options>General Options>Target>Read/write data

--ropi

Syntax

```
--ropi
```

Description

Use this option to make the compiler generate code that uses position-independent references to access code and read-only data.

See also

Position-independent code and data, page 191.

Project>Options>General Options>Target>Code and read-only data

--save_acc

Syntax

```
--save_acc
```

Description

Use this option to save and restore the DSP accumulator when the interrupt context switches. If the application uses the DSP, you should consider saving the accumulator when context switching, because the accumulator is destroyed if the interrupt service routine uses a `MUL` instruction or similar.

This option is not available in the IDE.

--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.

--sqrt_must_set_errno

Syntax

```
--sqrt_must_set_errno
```

Description

Use this option to disable replacing calls to the library function `sqrtf()` with the RXv2 core instruction `FSQRT`. This is needed to make the code comply with Standard C, because the `FSQRT` instruction does not set the C library symbol `errno` if the argument is out of range.

This option is set automatically when you choose an RXv2 core device using **Project>Options>General Options>Target>Device** and select **Project>Options>C/C++ Compiler>Floating-point semantics>Strict conformance**.

--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 171.

Project>Options>C/C++ Compiler>Language 1>Language conformance>Strict

--suppress_core_attribute

Syntax

```
--suppress_core_attribute
```

Description

Use this option to disable the generation of the runtime attribute `__core` for library object files. This means that the same library file can be linked both with applications compiled for the RXv1 architecture and with applications compiled for the RXv2 architecture.

Project>Options>General Options>Library Configuration>Suppress the core runtime model attribute

--system_include_dir

Syntax

```
--system_include_dir path
```

Parameters

path	The path to the system include files, see *Rules for specifying a filename or directory as parameters*, page 236.

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 247, and *--no_system_include*, page 259.

This option is not available in the IDE.

--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 76

Project>Options>C/C++ Compiler>Language 1>C dialect>C99>C++ inline semantics

--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**.

--vla

Syntax

```
--vla
```

Description

Use this option to enable support for C99 variable length arrays. 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 169.

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 246.

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 235.

Summary of linker options

This table summarizes the linker options:

Command line option	Description
`--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
`--cpp_init_routine`	Specifies a user-defined C++ dynamic initialization routine
`--debug_lib`	Uses the C-SPY debug library
`--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
`--enable_stack_usage`	Enables stack usage analysis
`--entry`	Treats the symbol as a root symbol and as the start of the application
`--error_limit`	Specifies the allowed number of errors before linking stops
`--export_builtin_config`	Produces an `icf` file for the default configuration
`-f`	Extends the command line

Table 31: Linker options summary

Command line option	Description
`--force_output`	Produces an output file even if errors occurred
`--image_input`	Puts an image file in a section
`--inline`	Inlines small routines
`--keep`	Forces a symbol to be included in the application
`--log`	Enables log output for selected topics
`--log_file`	Directs the log to a file
`--mangled_names_in_messages`	Adds mangled names in messages
`--map`	Produces a map file
`--merge_duplicate_sections`	Merges equivalent read-only sections
`--misrac`	Enables error messages specific to MISRA-C:1998. This option is a synonym to `--misrac1998` and is only available for backwards compatibility.
`--misrac1998`	Enables error messages specific to MISRA-C:1998. See the *IAR Embedded Workbench® MISRA C:1998 Reference Guide*.
`--misrac2004`	Enables error messages specific to MISRA-C:2004. See the *IAR Embedded Workbench® MISRA C:2004 Reference Guide*.
`--misrac_verbose`	Enables verbose logging of MISRA C checking. See the *IAR Embedded Workbench® MISRA C:1998 Reference Guide* and the *IAR Embedded Workbench® MISRA C:2004 Reference Guid*.
`--no_fragments`	Disables section fragment handling
`--no_library_search`	Disables automatic runtime library search
`--no_locals`	Removes local symbols from the ELF executable image.
`--no_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`.
`--only_stdout`	Uses standard output only
`--output`	Sets the object filename

Table 31: Linker options summary (Continued)

Command line option	Description
`--place_holder`	Reserve a place in ROM to be filled by some other tool, for example a checksum calculated by `ielftool`.
`--redirect`	Redirects a reference to a symbol to another symbol
`--remarks`	Enables remarks
`--search`	Specifies more directories to search for object and library files
`--silent`	Sets silent operation
`--stack_usage_control`	Specifies a stack usage control file
`--strip`	Removes debug information from the executable image
`--threaded_lib`	Configures the runtime library for use with threads
`--vfe`	Controls Virtual Function Elimination
`--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.

To comply with the RX ABI, the compiler generates assembler labels for symbol and function names by prefixing an underscore. You must remember to add this extra underscore when you refer to C symbols in any of the linker options, such as `--define_symbol` and `--redirect`, or in directives in the linker configuration file, such as `define symbol`. For example, `main` must be written as `_main`.

Note that if you use the options page **Extra Options** to specify specific command line options, the IDE does not perform an instant check for consistency problems like conflicting options, duplication of options, or use of irrelevant options.

--call_graph

Syntax

```
--call_graph {filename|directory}
```

Parameters

See *Rules for specifying a filename or directory as parameters*, page 236.

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 88

Project>Options>Linker>Advanced>Call graph output (XML)

--config

Syntax

```
--config filename
```

Parameters

See *Rules for specifying a filename or directory as parameters*, page 236.

Description

Use this option to specify the configuration file to be used by the linker (the default filename extension is `icf`). If no configuration file is specified, a default configuration is used. This option can only be used once on the command line.

See also

The chapter *The linker configuration file*.

Project>Options>Linker>Config>Linker configuration file

--config_def

Syntax

```
--config_def symbol[=constant_value]
```

Parameters

symbol	The name of the symbol to be used in the configuration file. By default, the value 0 (zero) is used.
constant_value	The constant value of the configuration symbol.

Description

Use this option to define a constant configuration symbol to be used in the configuration file. This option has the same effect as the `define symbol` directive in the linker configuration file. This option can be used more that once on the command line.

See also

--define_symbol, page 276 and *Interaction between ILINK and the application*, page 104.

Project>Options>Linker>Config>Defined symbols for configuration file

--cpp_init_routine

Syntax

```
--cpp_init_routine routine
```

Parameters

routine	A user-defined C++ dynamic initialization routine.

Description

When using the IAR C/C++ compiler and the standard library, C++ dynamic initialization is handled automatically. In other cases you might need to use this option.

If any sections with the section type `INIT_ARRAY` or `PREINIT_ARRAY` are included in your application, the C++ dynamic initialization routine is considered to be needed. By default, this routine is named `__iar_cstart_call_ctors` and is called by the startup code in the standard library. Use this option if you are not using the standard library and require another routine to handle these section types.

To set this option, use **Project>Options>Linker>Extra Options**.

--debug_lib

Syntax

```
--debug_lib
```

Description

Use this option to include the C-SPY debug library.

See also

Application debug support, page 114.

Project>Options>Linker>Library>Include C-SPY debugging support

--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 that his option is different from the `define symbol` directive.

See also

--config_def, page 274 and *Interaction between ILINK and the application*, page 104.

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 236.

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 `Pe177`

Description

Use this option to reclassify certain diagnostic messages as remarks. A remark is the least severe type of diagnostic message and indicates a construction that may cause strange behavior in the executable image. This option may be used more than once on the command line.

Note: By default, remarks are not displayed; use the `--remarks` option to display them.

Project>Options>Linker>Diagnostics>Treat these as remarks

--diag_suppress

Syntax

```
--diag_suppress=tag[,tag,...]
```

Parameters

tag	The number of a diagnostic message, for example the message number `Pe117`

Description

Use this option to suppress certain diagnostic messages. These messages will not be displayed. This option may be used more than once on the command line.

Project>Options>Linker>Diagnostics>Suppress these diagnostics

--diag_warning

Syntax

```
--diag_warning=tag[,tag,...]
```

Parameters

tag	The number of a diagnostic message, for example the message number `Pe826`

Description

Use this option to reclassify certain diagnostic messages as warnings. A warning indicates an error or omission that is of concern, but which will not cause the linker to stop before linking is completed. This option may be used more than once on the command line.

Project>Options>Linker>Diagnostics>Treat these as warnings

--diagnostics_tables

Syntax

```
--diagnostics_tables {filename|directory}
```

Parameters

See *Rules for specifying a filename or directory as parameters*, page 236.

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.

--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 88

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`).

Project>Options>Linker>Library>Override default program entry

--error_limit

Syntax

```
--error_limit=n
```

Parameters

n	The number of errors before the linker stops linking. *n* must be a positive integer; `0` indicates no limit.

Description

Use the `--error_limit` option to specify the number of errors allowed before the linker stops the linking. By default, 100 errors are allowed.

This option is not available in the IDE.

--export_builtin_config

Syntax

`--export_builtin_config` *filename*

Parameters

See *Rules for specifying a filename or directory as parameters*, page 236.

Description

Exports the configuration used by default to a file.

This option is not available in the IDE.

-f

Syntax

`-f` *filename*

Parameters

See *Rules for specifying a filename or directory as parameters*, page 236.

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.

To set this option, use **Project>Options>Linker>Extra Options**.

--force_output

Syntax

`--force_output`

Description

Use this option to produce an output executable image regardless of any linking errors.

To set this option, use **Project>Options>Linker>Extra Options**

--image_input

Syntax

```
--image_input filename [,symbol,[section[,alignment]]]
```

Parameters

filename	The pure binary file containing the raw image you want to link
symbol	The symbol which the binary data can be referenced with.
section	The section where the binary data will be placed; default is `.text`.
alignment	The alignment of the section; default is `1`.

Description

Use this option to link pure binary files in addition to the ordinary input files. The file's entire contents are placed in the section, which means it can only contain pure binary data.

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 by use of 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 282.

Project>Options>Linker>Input>Raw binary image

--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.

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.
`initialization`	Lists copy batches and the compression selected for each batch.
`libraries`	Lists all decisions taken by the automatic library selector. This might include extra symbols needed (`--keep`), redirections (`--redirect`), as well as which runtime libraries that were selected.
`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.

`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 283.

Project>Options>Linker>List>Generate log

--log_file

Syntax — `--log_file` *`filename`*

Parameters — See *Rules for specifying a filename or directory as parameters*, page 236.

Description — Use this option to direct the log output to the specified file.

See also — *--log*, page 282.

Project>Options>Linker>List>Generate log

--mangled_names_in_messages

Syntax — `--mangled_names_in_messages`

Description — Use this option to produce both mangled and unmangled names for C/C++ symbols in messages. Mangling is a technique used for mapping a complex C name or a C++ name (for example, for overloading) into a simple name. For example, `void h(int, char)` becomes `_Z1hic`.

This option is not available in the IDE.

--map

Syntax — `--map {`*`filename`*`|`*`directory`*`}`

Description

Use this option to produce a linker memory map file. The map file has the default filename extension `map`. The map file contains:

- Linking summary in the map file header which lists the version of the linker, the current date and time, and the command line that was used.
- Runtime attribute summary which lists runtime attributes.
- Placement summary which lists each section/block in address order, sorted by placement directives.
- Initialization table layout which lists the data ranges, packing methods, and compression ratios.
- Module summary which lists contributions from each module to the image, sorted by directory and library.
- Entry list which lists all public and some local symbols in alphabetical order, indicating which module they came from.
- Some of the bytes might be reported as *shared*.

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

This option can only be used once on the command line.

Project>Options>Linker>List>Generate linker map file

--merge_duplicate_sections

Syntax

```
--merge_duplicate_sections
```

Description

Use this option to keep only one copy of equivalent read-only sections. Note that 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.

Project>Options>Linker>Optimizations>Merge duplicate sections

--no_fragments

Syntax

```
--no_fragments
```

Description

Use this option to disable section fragment handling. Normally, the toolset uses IAR proprietary information for transferring section fragment information to the linker. The linker uses this information to remove unused code and data, and thus further minimize the size of the executable image. 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 98.

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 that 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_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 98.

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 291 and *Virtual function elimination*, page 202.

To set related options, choose:

Project>Options>Linker>Optimizations>PerformC++ 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`) also for messages that are normally directed to the error output stream (`stderr`).

This option is not available in the IDE.

--output, -o

Syntax

```
--output {filename|directory}
-o {filename|directory}
```

Parameters

See *Rules for specifying a filename or directory as parameters*, page 236.

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

--place_holder

Syntax

```
--place_holder symbol[,size[,section[,alignment]]]
```

Parameters

symbol	The name of the symbol to create
size	Size in ROM; by default 4 bytes
section	Section name to use; by default `.text`
alignment	Alignment of section; by default `1`

Description

Use this option to reserve a place in ROM to be filled by some other tool, for example a checksum calculated by `ielftool`. Each use of this linker option results in a section with the specified name, size, and alignment. The symbol can be used by your application to refer to the section.

Note: Like any other section, sections created by the `--place_holder` option will only be included in your application if the section appears to be needed. The `--keep` linker option, or the `keep` linker directive can be used for forcing such section to be included.

See also

IAR utilities, page 411.

To set this option, use **Project>Options>Linker>Extra Options**

--redirect

Syntax

```
--redirect from_symbol=to_symbol
```

Parameters

from_symbol	The name of the source symbol
to_symbol	The name of the destination symbol

Description

Use this option to change a reference from one symbol to another symbol.

To set this option, use **Project>Options>Linker>Extra Options**

--remarks

Syntax

```
--remarks
```

Description

The least severe diagnostic messages are called remarks. A remark indicates a source code construct that may cause strange behavior in the generated code. By default, the linker does not generate remarks. Use this option to make the linker generate remarks.

See also

Severity levels, page 231.

Project>Options>Linker>Diagnostics>Enable remarks

--search

Syntax

```
--search 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, page 48.

This option is not available in the IDE.

--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 236.

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 88

Project>Options>Linker>Advanced>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

--threaded_lib

Syntax

```
--threaded_lib
```

Description

Use this option to automatically configure the runtime library for use with threads.

Project>Options>General Options>Library Configuration>Enable thread support in library

--vfe

Syntax

```
--vfe=[forced]
```

Parameters

`forced`	Performs Virtual Function Elimination even if one or more modules lack the needed virtual function elimination information.

Description

Use this option to perform Virtual Function Elimination even if one or more modules lack the needed virtual function elimination information. Without the parameter `forced`, this option has no effect.

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 286 and *Virtual function elimination*, page 202.

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 246 and *--diag_warning*, page 278.

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 236.

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 98

To set this option, use **Project>Options>Linker>Extra Options**

Data representation

- Alignment
- 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 most strict alignment. To decrease the alignment requirements on the structure and its members, use `#pragma pack` or the `__packed` data type attribute.

All data types must have a size that is a multiple of their alignment. Otherwise, only the first element of an array would be guaranteed to be placed in accordance with the alignment requirements. This means that the compiler might add pad bytes at the end of the structure. For more information about pad bytes, see *Packed structure types*, page

302.

Note that with the `#pragma data_alignment` directive you can increase the alignment demands on specific variables.

ALIGNMENT ON THE RX MICROCONTROLLER

The RX microcontroller can access memory using 8- to 32-bit operations. However, when an unaligned access is performed, more bus cycles are required. The compiler avoids this by assigning an alignment to every data type, ensuring that the RX microcontroller can read the data efficiently.

Byte order

For data access, the RX architecture allows a choice between the big- and little-endian byte order. All user and library modules in your application must use the same byte order.

Note: See the *IAR Assembler User Guide for RX* for more information about the assembler directives that toggle between code and data sections in linker segments.

Basic data types—integer types

The compiler supports both all Standard C basic data types and some additional types.

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`	32 bits	-2^{31} to $2^{31}-1$	4
`unsigned long`	32 bits	0 to $2^{32}-1$	4

Table 32: Integer types

Data type	Size	Range	Alignment
`signed long long`	64 bits	-2^{63} to $2^{63}-1$	4
`unsigned long long`	64 bits	0 to $2^{64}-1$	4

Table 32: Integer types (Continued)

* If you use the --int=16 compiler option, the int type will have the same size, range, and alignment as the short type.

Signed variables are represented using the two's complement form.

BOOL

The `bool` data type is supported by default in the C++ language. If you have enabled language extensions, the `bool` type can also be used in C source code if you include the file `stdbool.h`. This will also enable the boolean values `false` and `true`.

THE LONG LONG TYPE

The `long long` data type is supported with one restriction:

A `long long` variable cannot be used in a switch statement.

THE ENUM TYPE

The compiler will use the smallest type required to hold `enum` constants, preferring `signed` 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};
```

THE CHAR TYPE

The `char` type is by default unsigned in the compiler, but the `--char_is_signed` compiler option allows you to make it signed. Note, however, that the library is compiled with the `char` type as unsigned.

THE WCHAR_T TYPE

The `wchar_t` data type is an integer type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locals.

The `wchar_t` data type is supported by default in the C++ language. To use the `wchar_t` type also in C source code, you must include the file `stddef.h` from the runtime library.

BITFIELDS

In 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 RX, 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 into a container of its base type from the least significant bit to the most significant bit. If the last container is of the same type and has enough bits available, the bitfield is placed into this container, otherwise a new container is allocated. This allocation scheme is referred to as the disjoint type bitfield allocation.

If you use the directive `#pragma bitfield=reversed`, bitfields are placed from the most significant bit to the least significant bit in each container. See *bitfields*, page 323.

Assume this example:

```
struct BitfieldExample
{
  uint32_t a : 12;
  uint16_t b : 3;
  uint16_t c : 7;
  uint8_t  d;
};
```

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.

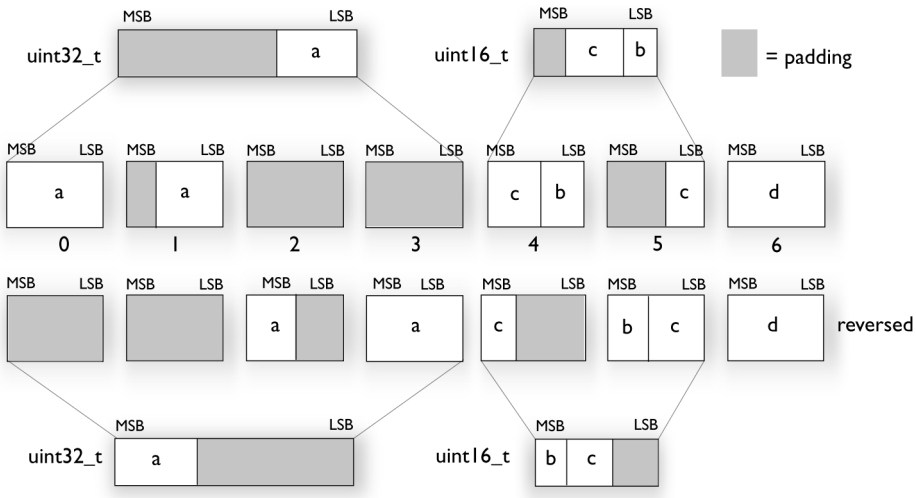
To place the second bitfield, `b`, a new container is allocated at offset `4`, because the type of the bitfield is not the same as that of the previous one. `b` is placed into the least significant three bits of this container.

The third bitfield, `c`, has the same type as `b` and fits into the same container.

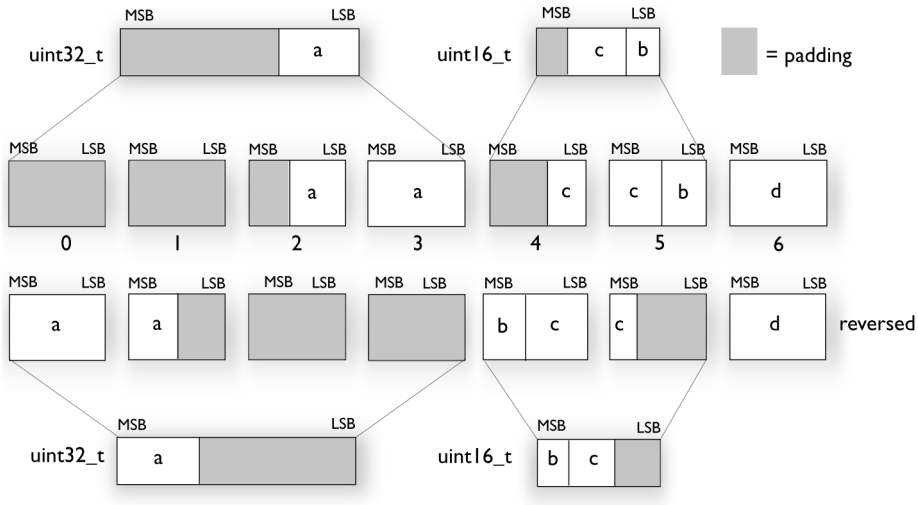
The fourth member, `d`, is allocated into the byte at offset `6`. `d` cannot be placed into the same container as `b` and `c` because it is not a bitfield, it is not of the same type, and it would not fit.

When using reverse order, each bitfield is instead placed starting from the most significant bit of its container.

This is the layout of `bitfield_example` for little-endian data:



This is the layout of `bitfield_example` for big-endian data:



Basic data types—floating-point types

In the IAR C/C++ Compiler for RX, floating-point values are represented in standard IEEE 754 format. The sizes for the different floating-point types are:

Type	Size if double=32	Size if double=64
`float`	32 bits	32 bits
`double`	32 bits (default)	64 bits
`long double`	32 bits	64 bits

Table 33: Floating-point types

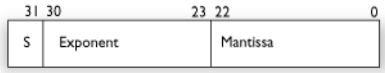
Note: The size of `double` and `long double` depends on the `--double={32|64}` option, see *--double*, page 248. The type `long double` uses the same precision as `double`.

FLOATING-POINT ENVIRONMENT

Exception flags are not supported. The `feraiseexcept` function does not raise any exceptions.

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
```

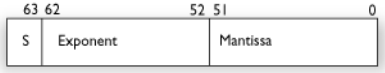
The range of the number is at least:

```
±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.
- For the float type, Not a number (`NaN`) is represented by setting the exponent to the highest positive value and the mantissa to a non-zero value. The value of the sign bit is ignored.

- For the `double` type, Not a number (`NaN`) is represented by setting the exponent to `7FF` and at least one of the highest twenty bits in the mantissa to non-zero. The lower thirty-two bits of the mantissa are ignored. The value of the sign bit is also 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.

By default, subnormal numbers are only supported for 64-bit floating-point numbers. However, the RX600 libraries can use the *unimplemented processing exception* of the CPU to support 32-bit floating-point subnormal numbers.

To enable the subnormal number exception handler, use the *linker* option `--redirect` and use this linker command:

```
--redirect __float_placeholder=__unimpl_processing_handler
```

Supporting subnormal numbers for 32-bit floating-point numbers this way requires a large overhead, both in size and speed, compared to a normal FPU instruction which requires very few CPU cycles. The subnormal number exception handler will use approximately 900 bytes of code space, and about 50–200 cycles per exception, depending on the operation and the operands. For that reason, if execution speed is important, try to use floating-point algorithms that do not require subnormal number capabilities for 32-bit floating-point numbers.

To remove subnormal number handling for 32-bit floating-point numbers, use this linker command:

```
--redirect __float_placeholder=__floating_point_handler
```

Pointer types

The compiler has two basic types of pointers: function pointers and data pointers.

FUNCTION POINTERS

The function pointer of the IAR C/C++ Compiler for RX is a 32-bit pointer that can address the entire memory. The internal representation of the function pointer is the actual address it refers to. The function pointer is a pointer to `__code` memory.

DATA POINTERS

The data pointer of the IAR C/C++ Compiler for RX is a 32-bit `signed int` pointer that can address the entire memory. The data pointer is a pointer to `__data32` memory.

CASTING

Casts between pointers have these characteristics:

- Casting a *value* of an integer type to a pointer of a smaller type is performed by truncation
- Casting a value of an integer type to a pointer of a larger type is performed by zero extension
- Casting a pointer type to a smaller integer type is performed by truncation
- Casting a *pointer type* to a larger integer type is performed by zero extension
- Casting a *data pointer* to a function pointer and vice versa is illegal
- Casting a *function pointer* to an integer type gives an undefined result

size_t

`size_t` is the unsigned integer type of the result of the `sizeof` operator. In the IAR C/C++ Compiler for RX, 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 the IAR C/C++ Compiler for RX, the type used for `ptrdiff_t` is the signed integer variant of the `size_t` type.

intptr_t

`intptr_t` is a signed integer type large enough to contain a `void *`. In the IAR C/C++ Compiler for RX, the type used for `intptr_t` is `unsigned long`.

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

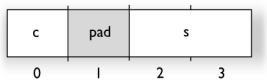
The `struct` and `union` types have the same alignment as the member with the highest alignment requirement. Note that this alignment requirement also applies to a member that is a structure. To allow arrays of aligned structure objects, the size of a `struct` is adjusted to an even multiple of the alignment.

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 that accessing an object that is not correctly aligned requires code that is both larger and slower. If such structure members are accessed many times, it is usually better to construct the correct values in a `struct` that is not packed, and access this `struct` instead.

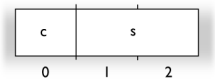
Special care is also needed when creating and using pointers to misaligned members. For direct access to misaligned members in a packed `struct`, the compiler will emit the correct (but slower and larger) code when needed. However, when a misaligned member is accessed through a pointer to the member, the normal (smaller and faster) code is used. In the general case, this will not work, 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()
```

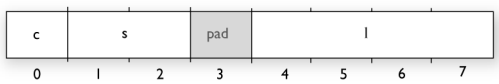
The structure `S` has this memory layout:



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 204.

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—thus 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 RX are described below.

Rules for accesses

In the IAR C/C++ Compiler for RX, 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 all memory types and for all properly aligned basic data types except 64-bit `double` and `long`. For 64-bit `double` and `long`, 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, 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 Embedded C++ features are used for a type, no assumptions can be made concerning the data representation. This means, for example, that it is not supported to write assembler code that accesses class members.

Extended keywords

- General syntax rules for extended keywords
- Summary of extended keywords
- Descriptions of extended keywords

For information about the address ranges of the different memory areas, see the chapter *Section reference*.

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 RX microcontroller. There are two types of attributes—*type attributes* and *object attributes*:

- Type attributes affect the *external functionality* of the data object or function
- Object attributes affect the *internal functionality* of the data object or function.

The syntax for the keywords differs slightly depending on whether it is a type attribute or an object attribute, and whether it is applied to a data object or a function.

For more information about each attribute, see *Descriptions of extended keywords*, page 311. For information about how to use attributes to modify data, see the chapter *Data storage*.

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 249.

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`.

Type attributes can be further divided into *memory type attributes* and *general type attributes*. Memory type attributes are referred to as simply *memory attributes* in the rest of the documentation.

Memory attributes

A memory attribute corresponds to a certain logical or physical memory in the microcontroller.

Available *data memory attributes*:

`__data16`, `__data24`, `__data32`, `__sfr`, `__sbrel`.

Data objects, functions, and destinations of pointers or C++ references always have a memory attribute. If no attribute is explicitly specified in the declaration or by the pragma directive `#pragma type_attribute`, an appropriate default attribute is implicitly used by the compiler. You can specify one memory attribute for each level of pointer indirection.

General type attributes

Available *function type attributes* (affect how the function should be called):

`__fast_interrupt`, `__interrupt`, `__monitor`, `__task`

Available *data type attributes*:

`__packed`

You can specify as many type attributes as required for each level of pointer indirection.

Syntax for type attributes used on data objects

Type attributes use almost the same syntax rules as the type qualifiers `const` and `volatile`. For example:

```
__data16 int i;
int __data16 j;
```

Both `i` and `j` are placed in data16 memory.

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.

```
int __data16 * p;      /* integer in data16 memory */

int * __data16 p;      /* pointer in data16 memory */
```

```
__data16 int * p;      /* pointer in data16 memory */
```

In all cases, if a memory attribute is not specified, an appropriate default memory type is used.

Using a type definition can sometimes make the code clearer:

```
typedef __data16 int d16_int;
d16_int *q1;
```

`d16_int` is a typedef for integers in data16 memory. 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=__data16
int * q2;
```

The variable `q2` is placed in data16 memory.

For more examples of using memory attributes, see *More examples*, page 63.

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 in parentheses, for example:

```
__interrupt void my_handler(void);
```

or

```
void (__interrupt my_handler)(void);
```

This declaration of `my_handler` is equivalent with the previous one:

```
#pragma type_attribute=__interrupt
void my_handler(void);
```

OBJECT ATTRIBUTES

These object attributes are available:

- Object attributes that can be used for variables:

  `__no_init`
- Object attributes that can be used for functions and variables:

  `location`, `@`, `__root`, `__weak`

- Object attributes that can be used for functions:

  `__absolute`, `__intrinsic`, `__nested`, `__noreturn`, `__ramfunc`, `vector`

You can specify as many object attributes as required for a specific function or data object.

For more information about `location` and `@`, see *Controlling data and function placement in memory*, page 207. For more information about `vector`, see *vector*, page 340.

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
`__data16`	Controls the storage of data objects
`__data24`	Controls the storage of data objects
`__data32`	Controls the storage of data objects
`__fast_interrupt`	Supports fast interrupt functions
`__interrupt`	Specifies interrupt functions
`__intrinsic`	Reserved for compiler internal use only
`__monitor`	Specifies atomic execution of a function
`__nested`	Allows an interrupt function to be nested, that is, interruptible by another interrupt
`__no_init`	Places a data object in non-volatile memory
`__noreturn`	Informs the compiler that the function will not return

Table 34: Extended keywords summary

Extended keyword	Description
`__packed`	Decreases data type alignment to 1
`__ramfunc`	Makes a function execute in RAM
`__root`	Ensures that a function or variable is included in the object code even if unused
`__sbrel`	Controls the storage of data objects
`__sfr`	Controls the storage of data objects
`__task`	Relaxes the rules for preserving registers
`__weak`	Declares a symbol to be externally weakly linked

Table 34: Extended keywords summary (Continued)

Descriptions of extended keywords

These sections give detailed information about each extended keyword.

__absolute

Syntax

See *Syntax for object attributes*, page 310.

Description

The `__absolute` keyword makes references to the object use absolute addressing.

The following limitations apply:

- Only available when the `--ropi` compiler option is used
- Can only be used on external declarations.

Example

```
extern __absolute int func1(void);
```

__data16

Syntax

See *Syntax for type attributes used on data objects*, page 308.

Description

The `__data16` memory attribute overrides the default storage of variables given by the selected data model and places individual variables and constants in data16 memory.

Storage information

- Address range: `0-0x7FFF`, `0xFFFF8000-0xFFFFFFFF` (64 Kbytes)
- Maximum object size: 32 Kbytes.
- Pointer size: 4 bytes.

Example
```
__data16 int x;
```

See also
Memory types, page 60.

__data24

Syntax
See *Syntax for type attributes used on data objects*, page 308.

Description
The `__data24` memory attribute overrides the default storage of variables and constants given by the selected data model, and places individual variables and constants in data24 memory.

Storage information
- Address range: `0-0x7FFFFF`, `0xFF800000-0xFFFFFFFF` (16 Mbytes)
- Maximum object size: 8 Mbytes–1
- Pointer size: 4 bytes

Example
```
__data24 int x;
```

See also
Memory types, page 60.

__data32

Syntax
See *Syntax for type attributes used on data objects*, page 308.

Description
The `__data32` memory attribute overrides the default storage of variables and constants given by the selected data model, and places individual variables and constants in data32 memory.

Storage information
- Address range: `0-0xFFFFFFFF` (4 Gbytes)
- Maximum object size: 2 Gbytes–1
- Pointer size: 4 bytes.

Example
```
__data32 int x;
```

See also
Memory types, page 60.

__fast_interrupt

Syntax

See *Syntax for type attributes used on functions*, page 309.

Description

The `__fast_interrupt` keyword specifies a very fast interrupt function of the highest priority, using the `FREIT` return mechanism. A fast interrupt function must have a `void` return type and cannot have any parameters.

Example

```
__fast_interrupt void my_interrupt_handler(void);
```

See also

Fast interrupt functions, page 72, *vector*, page 340, and *.inttable*, page 399.

__interrupt

Syntax

See *Syntax for type attributes used on functions*, page 309.

Description

The `__interrupt` keyword specifies interrupt functions. To specify one or several interrupt vectors, use the `#pragma vector` directive. The range of the interrupt vectors depends on the device used. It is possible to define an interrupt function without a vector, but then the compiler will not generate an entry in the interrupt vector table.

An interrupt function must have a `void` return type and cannot have any parameters.

The header file `io`*`device`*`.h`, where *`device`* corresponds to the selected device, contains predefined names for the existing interrupt vectors.

Example

```
#pragma vector=0x14
__interrupt void my_interrupt_handler(void);
```

See also

Interrupt functions, page 70, *vector*, page 340, and *.inttable*, page 399.

__intrinsic

Description

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

__monitor

Syntax

See *Syntax for type attributes used on functions*, page 309.

Description

The `__monitor` keyword causes interrupts to be disabled during execution of the function. This allows atomic operations to be performed, such as operations on

semaphores that control access to resources by multiple processes. A function declared with the `__monitor` keyword is equivalent to any other function in all other respects.

Example

```
__monitor int get_lock(void);
```

See also

Monitor functions, page 73. For information about related intrinsic functions, see *__disable_interrupt*, page 343, *__enable_interrupt*, page 343, *__get_interrupt_state*, page 344, and *__set_interrupt_state*, page 347, respectively.

__nested

Syntax

See *Syntax for object attributes*, page 310.

Description

The `__nested` keyword enables interrupts, which means new interrupts can be served inside an interrupt or fast interrupt function.

Example

```
__interrupt __nested void interrupt_handler(void);
```

See also

Nested interrupts, page 73.

__no_init

Syntax

See *Syntax for object attributes*, page 310.

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 221 and *do not initialize directive*, page 381.

__noreturn

Syntax

See *Syntax for object attributes*, page 310.

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.

Example

```
__noreturn void terminate(void);
```

__packed

Syntax

See *Syntax for type attributes used on data objects*, page 308. 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. The type of each member also receives the `__packed` type attribute.

  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 *" -> "int __packed *" not allowed: */
  int * p1 = xp->1;
  /* OK: */
  int __packed * p2 = &xp->i;
  /* OK, char not affected */
  char * p3 = &xp->ch;
}
```

See also

pack, page 334.

__ramfunc

Syntax

See *Syntax for object attributes*, page 310.

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`.

Note: This keyword cannot be used together with the compiler option `--endian=b`.

Example

```
__ramfunc int FlashPage(char * data, char * page);
```

See also

To read more about `__ramfunc` declared functions in relation to breakpoints, see the *C-SPY® Debugging Guide for RX*.

__root

Syntax — See *Syntax for object attributes*, page 310.

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 98.

__sbrel

Syntax — See *Syntax for type attributes used on data objects*, page 308.

Description — The `__sbrel` memory attribute places individual variables and constants in sbrel memory. It is only available when RWPI is enabled, and it is then the default memory attribute.

Storage information

- Address range: `0-0xFFFFFFFF` (4 Gbytes), offset relative to the `DB` base register
- Maximum object size: 2 Gbytes–1
- Pointer size: 4 bytes.

Example

```
__sbrel int x;
```

See also — *Memory types*, page 60.

__sfr

Syntax — See *Syntax for type attributes used on data objects*, page 308.

Description — The `__sfr` memory attribute overrides the default storage of variables and constants given by the selected data model, and places individual variables and constants in data32 memory. Using this attribute also stops the compiler from using the instructions `SMOVF` and `SSTR` to access the data.

Use this memory type attribute for all special function registers (SFR) and pointers to SFRs, to avoid unexpected program behavior.

Storage information

- Address range: `0-0xFFFFFFFF` (4 Gbytes)

- Maximum object size: 2 Gbytes–1
- Pointer size: 4 bytes.

Example
```
__sfr int x;
```

See also
Memory types, page 60.

__task

Syntax
See *Syntax for type attributes used on functions*, page 309.

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.

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);
```

__weak

Syntax
See *Syntax for object attributes*, page 310.

Description
Using the `__weak` object attribute on an external declaration of a symbol makes all references to that symbol in the module weak.

Using the `__weak` object attribute on a public definition of a symbol makes that definition a weak definition.

The linker will not include a module from a library solely to satisfy weak references to a symbol, nor will the lack of a definition for a weak reference result in an error. If no definition is included, the address of the object will be zero.

When linking, a symbol can have any number of weak definitions, and at most one non-weak definition. If the symbol is needed, and there is a non-weak definition, this definition will be used. If there is no non-weak definition, one of the weak definitions will be used.

Example

```
extern __weak int foo; /* A weak reference. */

__weak void bar(void)  /* A weak definition. */
{
  /* Increment foo if it was included. */
  if (&foo != 0)
    ++foo;
}
```

Pragma directives

- 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.
`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_variable_attributes`	Sets default type and object attributes for declarations and definitions of variables.
`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.
`error`	Signals an error while parsing.
`include_alias`	Specifies an alias for an include file.
`inline`	Controls inlining of a function.

Table 35: Pragma directives summary

Pragma directive	Description
language	Controls the IAR Systems language extensions.
location	Specifies the absolute address of a variable, or places groups of functions or variables in named sections.
message	Prints a message.
object_attribute	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.
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.
type_attribute	Adds type attributes to a declaration or to definitions.
vector	Specifies the vector of an interrupt or trap function.
weak	Makes a definition a weak definition, or creates a weak alias for a function or a variable.

Table 35: Pragma directives summary (Continued)

Note: For portability reasons, see also *Recognized pragma directives (6.10.6)*, page 450.

Descriptions of pragma directives

This section gives detailed information about each pragma directive.

bitfields

Syntax

```
#pragma bitfields={reversed|default}
```

Parameters

`reversed`	Bitfield members are placed from the most significant bit to the least significant bit.
`default`	Bitfield members are placed from the least significant bit to the most significant bit.

Description

Use this pragma directive to control the order of bitfield members.

Example

```
#pragma bitfields=reversed
/* Structure that uses reversed bitfields. */
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 296.

calls

Syntax

```
#pragma calls=function[, function...]
```

Parameters

`function`	Any declared function

Description

Use this pragma directive to list the functions that can be indirectly called in the following statement. This information can be used for stack usage analysis in the linker.

Note: For an accurate result, you must list all possible called functions.

Example

```
void Fun1(), Fun2();

void Caller(void (*fp)(void))
{
#pragma calls = Fun1, Fun2
  (*fp)();
}
```

See also

Stack usage analysis, page 88

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 88

data_alignment

Syntax

```
#pragma data_alignment=expression
```

Parameters

expression	A constant which must be a power of two (1, 2, 4, etc.).

Description

Use this pragma directive to give a variable a higher (more strict) alignment of the start address than it would otherwise have. This directive can be used on variables with static and automatic storage duration.

When you use this directive on variables with automatic storage duration, there is an upper limit on the allowed alignment for each function, determined by the calling convention used.

Note: Normally, the size of a variable is a multiple of its alignment. The `data_alignment` directive only affects the alignment of the variable's start address, and not its size, and can thus be used for creating situations where the size is not a multiple of the alignment.

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 307.
object_attribute	See *Object attributes*, page 309.
@ section_name	See *Data and function placement in sections*, page 209.

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 331

object_attribute, page 332

type_attribute, page 339

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 307.
object_attributes	See *Object attributes*, page 309.
@ *section_name*	See *Data and function placement in sections*, page 209.

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 with no attributes restores the initial state of no such defaults being applied to variables with static storage duration.

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 331

object_attribute, page 332

type_attribute, page 339

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.

See also

Diagnostics, page 230.

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.

See also

Diagnostics, page 230.

diag_remark

Syntax

```
#pragma diag_remark=tag[,tag,...]
```

Parameters

tag	The number of a diagnostic message, for example the message number `Pe177`.

Description

Use this pragma directive to change the severity level to `remark` for the specified diagnostic messages.

See also

Diagnostics, page 230.

diag_suppress

Syntax
```
#pragma diag_suppress=tag[,tag,...]
```

Parameters

tag	The number of a diagnostic message, for example the message number `Pe117`.

Description
Use this pragma directive to suppress the specified diagnostic messages.

See also
Diagnostics, page 230.

diag_warning

Syntax
```
#pragma diag_warning=tag[,tag,...]
```

Parameters

tag	The number of a diagnostic message, for example the message number `Pe826`.

Description
Use this pragma directive to change the severity level to `warning` for the specified diagnostic messages.

See also
Diagnostics, page 230.

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.

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 227.

inline

Syntax

```
#pragma inline[=forced|=never]
```

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.

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` 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` will enable inlining of the function also on the Medium optimization level.

See also

Inlining functions, page 76.

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 249 and *--strict*, page 267.

location

Syntax

```
#pragma location={address|NAME}
```

Parameters

address	The absolute address of the global or static variable for which you want an absolute location.
NAME	A user-defined section name; cannot be a section name predefined for use by the compiler and linker.

Description

Use this pragma directive to specify the location—the absolute address—of the global or static variable whose declaration follows the pragma directive. The variables must be declared `__no_init`. Alternatively, the directive can take a string specifying a section for placing either a variable or a 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=0x2000
__no_init volatile char PORT1; /* PORT1 is located at address
                                  0x2000 */

#pragma segment="FLASH"
#pragma location="FLASH"
__no_init char PORT2; /* PORT2 is located in section FLASH */

/* A better way is to use a corresponding mechanism */
#define FLASH _Pragma("location=\"FLASH\"")
/* ... */
FLASH __no_init int i; /* i is placed in the FLASH section */
```

See also — *Controlling data and function placement in memory*, page 207 and *Declare and place your own sections*, page 97.

message

Syntax

```
#pragma message(message)
```

Parameters

message	The message that you want to direct to the standard output stream.

Description — Use this pragma directive to make the compiler print a message to the standard output stream when the file is compiled.

Example

```
#ifdef TESTING
#pragma message("Testing")
#endif
```

object_attribute

Syntax

```
#pragma object_attribute=object_attribute[ object_attribute...]
```

Parameters — For information about object attributes that can be used with this pragma directive, see *Object attributes*, page 309.

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 307.

optimize

Syntax

```
#pragma optimize=[goal][level][no_optimization...]
```

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`.
`no_`*optimization*	Disables one or several optimizations; choose between: `no_code_motion`, disables code motion `no_cse`, disables common subexpression elimination `no_inline`, disables function inlining `no_tbaa`, disables type-based alias analysis `no_unroll`, disables loop unrolling `no_scheduling`, disables instruction scheduling.

Description

Use this pragma directive to decrease the optimization level, or to turn off some specific optimizations. This pragma directive only affects the function that follows immediately after the directive.

The parameters `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 213.

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 301 and *__packed*, page 315.

__printf_args

Syntax

```
#pragma __printf_args
```

Description

Use this pragma directive on a function with a printf-style format string. For any call to that function, the compiler verifies that the argument to each conversion specifier (for example `%d`) is syntactically correct.

Example

```
#pragma __printf_args
int printf(char const *,...);

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 264.

required

Syntax

```
#pragma required=symbol
```

Parameters

symbol	Any statically linked function or variable.

Description

Use this pragma directive to ensure that a symbol which is needed by a second symbol is included in the linked output. The directive must be placed immediately before the second symbol.

Use the directive if the requirement for a symbol is not otherwise visible in the application, for example if a variable is only referenced indirectly through the section it resides in.

Example

```
const char copyright[] = "Copyright by me";

#pragma required=copyright
int main()
{
  /* Do something here. */
}
```

Even if the copyright string is not used by the application, it will still be included by the linker and available in the output.

rtmodel

Syntax

```
#pragma rtmodel="key","value"
```

Parameters

"*key*"	A text string that specifies the runtime model attribute.
"*value*"	A text string that specifies the value of the runtime model attribute. Using the special value * is equivalent to not defining the attribute at all.

Description

Use this pragma directive to add a runtime model attribute to a module, which can be used by the linker to check consistency between modules.

This pragma directive is useful for enforcing consistency between modules. All modules that are linked together and define the same runtime attribute key must have the same value for the corresponding key, or the special value `*`. It can, however, be useful to state explicitly that the module can handle any runtime model.

A module can have several runtime model definitions.

Note: The predefined compiler runtime model attributes start with a double underscore. To avoid confusion, this style must not be used in the user-defined attributes.

Example

```
#pragma rtmodel="I2C","ENABLED"
```

The linker will generate an error if a module that contains this definition is linked with a module that does not have the corresponding runtime model attributes defined.

See also

Checking module consistency, page 143.

__scanf_args

Syntax

```
#pragma __scanf_args
```

Description

Use this pragma directive on a function with a scanf-style format string. For any call to that function, the compiler verifies that the argument to each conversion specifier (for example `%d`) is syntactically correct.

Example

```
#pragma __scanf_args
int 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" [__memoryattribute]
```

alias

```
#pragma segment="NAME" [__memoryattribute]
```

Parameters

NAME	The name of the section.
__memoryattribute	An optional memory attribute identifying the memory the section will be placed in; if not specified, default memory is used.

Description

Use this pragma directive to define a section name that can be used by the section operators `__section_begin`, `__section_end`, and `__section_size`. All section declarations for a specific section must have the same memory type attribute and alignment.

The *__memoryattribute* parameteris only relevant when used together with the section operators `__section_begin`, `__section_end`, and `__section_size`.

If an optional memory attribute is used, the return type of the section operators `__section_begin` and `__section_end` is:

`void __memoryattribute *`.

Note: To place variables or functions in a specific section, use the `#pragma location` directive or the `@` operator.

Example

```
#pragma section="MYDATA16" __data16
```

See also

Dedicated section operators, page 173. For more information about sections and segment parts, see the chapter *Linking your application*.

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 that 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. Note that this argument is not supported by the compiler.
`DEFAULT`	Sets the default behavior, that is `ON`.

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
```

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 307.

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, an `int` object with the memory attribute `__data16` is defined:

```
#pragma type_attribute=__data16
int x;
```

This declaration, which uses extended keywords, is equivalent:

```
__data16 int x;
```

See also — The chapter *Extended keywords*.

vector

Syntax

```
#pragma vector=vector1[, vector2, vector3, ...]
```

Parameters

vectorN	The vector number(s) of an interrupt function.

Description — Use this pragma directive to specify the vector(s) of an interrupt or trap function whose declaration follows the pragma directive. Note that several vectors can be defined for each function.

Example

```
#pragma vector=0x14
__interrupt void my_handler(void);
```

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 318.

Intrinsic functions

- Summary of intrinsic functions
- Descriptions of intrinsic functions

Summary of intrinsic functions

To use intrinsic functions in an application, include the header file `intrinsics.h`.

Note that the intrinsic function names start with double underscores, for example:

```
__disable_interrupt
```

This table summarizes the intrinsic functions:

Intrinsic function	Description
`__break`	Inserts a `BRK` instruction
`__c_base`	Returns the value of the base register for static data when RWPI is enabled
`__delay_cycles`	Inserts code to delay execution
`__disable_interrupt`	Disables interrupts
`__enable_interrupt`	Enables interrupts
`__exchange`	Inserts an `XCHG` instruction
`__FSQRT`	Inserts an `FSQRT` instruction
`__get_FINTV_register`	Returns the value of the `FINTV` register
`__get_FPSW_register`	Returns the value of the `FPSW` register
`__get_interrupt_level`	Returns the interrupt level
`__get_interrupt_state`	Returns the interrupt state
`__get_interrupt_table`	Returns the value of the `INTB` register
`__get_ISP_register`	Returns the value of the `ISP` register
`__get_PSW_register`	Returns the value of the `PSW` register
`__get_USP_register`	Returns the value of the `USP` register
`__illegal_opcode`	Inserts an illegal operation code
`__MOVCO`	Inserts a `MOVCO` instruction
`__MOVLI`	Inserts a `MOVLI` instruction

Table 36: Intrinsic functions summary

Intrinsic function	Description
__no_operation	Inserts a NOP instruction
__RMPA_B	Inserts an RMPA.B instruction
__RMPA_L	Inserts an RMPA.L instruction
__RMPA_W	Inserts an RMPA.W instruction
__ROUND	Inserts a ROUND instruction
__s_base	Returns the value of the base register for static code when ROPI is enabled
__set_FINTV_register	Writes a specific value to the FINTV register
__set_FPSW_register	Writes a specific value to the FPSW register
__set_interrupt_level	Sets the interrupt level
__set_interrupt_state	Restores the interrupt state
__set_interrupt_table	Writes a specific value to the INTB register
__set_ISP_register	Writes a specific value to the ISP register
__set_PSW_register	Writes a specific value to the PSW register
__set_USP_register	Writes a specific value to the USP register
__software_interrupt	Inserts an INT instruction
__wait_for_interrupt	Inserts a WAIT instruction

Table 36: Intrinsic functions summary (Continued)

Descriptions of intrinsic functions

This section gives reference information about each intrinsic function.

__break

Syntax

```
void __break(void);
```

Description

Inserts a BRK instruction.

__c_base

Syntax

```
unsigned long __c_base(void);
```

Description

Returns the value of the base register for static data when RWPI is enabled.

__delay_cycles

Syntax

```
void __delay_cycles(unsigned long cycles);
```

Description

Inserts code to delay execution for at least *cycles* number of execution cycles.

__disable_interrupt

Syntax

```
void __disable_interrupt(void);
```

Description

Disables interrupts by inserting the `DI` instruction.

__enable_interrupt

Syntax

```
void __enable_interrupt(void);
```

Description

Enables interrupts by inserting the `EI` instruction.

__exchange

Syntax

```
unsigned long __exchange(unsigned long src,
                         unsigned long * dst);
```

Description

Inserts an `XCHG` *src*,*dst* instruction.

__FSQRT

Syntax

```
float __FSQRT(float);
```

Description

Inserts an `FSQRT` instruction. This instruction is only supported by the RXv2 architecture.

Subnormal operands are not supported by the exception handler. Refer to the hardware manual for other limitations on floating-point representation.

__get_FINTV_register

Syntax

```
__fast_int_f __get_FINTV_register(void);
```

Description Returns the value of the `FINTV` register. The type `__fast_int_f` is declared in the `intrinsics.h` file.

__get_FPSW_register

Syntax
```
unsigned long __get_FPSW_register(void);
```

Description Returns the value of the `FPSW` register.

__get_interrupt_level

Syntax
```
__ilevel_t __get_interrupt_level(void);
```

Description Returns the current interrupt level. The return type `__ilevel_t` has this definition:

```
typedef unsigned char __ilevel_t;
```

The return value of `__get_interrupt_level` can be used as an argument to the `__set_interrupt_level` intrinsic function.

__get_interrupt_state

Syntax
```
__istate_t __get_interrupt_state(void);
```

Description Returns the global interrupt state. The return value can be used as an argument to the `__set_interrupt_state` intrinsic function, which will restore the interrupt state.

Example
```
#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_interrupt_table

Syntax
```
unsigned long __get_interrupt_table(void);
```

Description
Returns the value of the INTB register.

__get_ISP_register

Syntax
```
unsigned long __get_ISP_register(void);
```

Description
Returns the value of the ISP register.

__get_PSW_register

Syntax
```
unsigned long __get_PSW_register(void);
```

Description
Returns the value of the PSW register.

__get_USP_register

Syntax
```
unsigned long __get_USP_register(void);
```

Description
Returns the value of the USP register.

__illegal_opcode

Syntax
```
void __illegal_opcode(void);
```

Description
Inserts an illegal operation code.

__MOVCO

Syntax
```
long __MOVCO(long, long *);
```

Description
Inserts a MOVCO instruction. The MOVCO instruction is used together with the MOVLI instruction to support thread synchronization. These instructions are only available in the RXv2 core.

__MOVLI

Syntax

```
void __MOVLI(long *);
```

Description

Inserts a `MOVLI` instruction. The `MOVLI` instruction is used together with the `MOVCO` instruction to support thread synchronization. These instructions are only available in the RXv2 core.

__no_operation

Syntax

```
void __no_operation(void);
```

Description

Inserts a `NOP` instruction.

__RMPA_B

Syntax

```
void __RMPA_B(signed char * v1, signed char * v2,
              unsigned long n, rmpa_t * acc);
```

Description

Inserts an `RMPA.B` instruction. The `RMPA` instruction sequentially multiplies the two vectors *v1* and *v2* and adds each product to the accumulator *acc*. The length of the vectors is *n*. You can supply an initial value for the accumulator *acc*, either variable or a constant. The type `rmpa_t` is declared in the `intrinsics.h` file.

__RMPA_L

Syntax

```
void __RMPA_L(signed long * v1, signed long * v2,
              unsigned long n, rmpa_t * acc);
```

Description

Inserts an `RMPA.L` instruction. The `RMPA` instruction sequentially multiplies the two vectors *v1* and *v2* and adds each product to the accumulator *acc*. The length of the vectors is *n*. You can supply an initial value for the accumulator *acc*, either variable or a constant. The type `rmpa_t` is declared in the `intrinsics.h` file.

__RMPA_W

Syntax

```
void __RMPA_W(signed short * v1, signed short * v2,
              unsigned long n, rmpa_t * acc);
```

Description

Inserts an `RMPA.W` instruction. The `RMPA` instruction sequentially multiplies the two vectors *v1* and *v2* and adds each product to the accumulator *acc*. The length of the

vectors is *n*. You can supply an initial value for the accumulator *acc*, either variable or a constant. The type `rmpa_t` is declared in the `intrinsics.h` file.

__ROUND

Syntax
```
int __ROUND(float);
```

Description
Inserts a `ROUND` instruction. See *Casting a floating-point value to an integer*, page 204.

__s_base

Syntax
```
unsigned long __s_base(void);
```

Description
Returns the value of the base register for static code when ROPI is enabled.

__set_FINTV_register

Syntax
```
void __set_FINTV_register(__fast_int_f);
```

Description
Writes a specific value to the `FINTV` register. The type `__fast_int_f` is declared in the `intrinsics.h` file.

__set_FPSW_register

Syntax
```
void __set_FPSW_register(unsigned long);
```

Description
Writes a specific value to the `FPSW` register.

__set_interrupt_level

Syntax
```
void __set_interrupt_level(__ilevel_t);
```

Description
Sets the interrupt level. For information about the `__ilevel_t` type, see *__get_interrupt_level*, page 344.

__set_interrupt_state

Syntax
```
void __set_interrupt_state(__istate_t);
```

Description — Restores the interrupt state to a value previously returned by the `__get_interrupt_state` function.

For information about the `__istate_t` type, see *__get_interrupt_state*, page 344.

__set_interrupt_table

Syntax
```
void __set_interrupt_table(unsigned long address);
```

Description — Writes a specific value to the `INTB` register.

__set_ISP_register

Syntax
```
void __set_ISP_register(unsigned long);
```

Description — Writes a specific value to the `ISP` register.

__set_PSW_register

Syntax
```
void __set_PSW_register(unsigned long);
```

Description — Writes a specific value to the `PSW` register.

__set_USP_register

Syntax
```
void __set_USP_register(unsigned long);
```

Description — Writes a specific value to the `USP` register.

__software_interrupt

Syntax
```
void __software_interrupt(unsigned char);
```

Description — Inserts an `INT` instruction.

__wait_for_interrupt

Syntax `void __wait_for_interrupt(void);`

Description Inserts a `WAIT` instruction.

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 RX 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 352.
- 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 243.
- Preprocessor extensions

  There are several preprocessor extensions, for example many pragma directives; for more information, see the chapter *Pragma directives*. For information about the corresponding `_Pragma` operator and the other extensions related to the preprocessor, see *Descriptions of miscellaneous preprocessor extensions*, page 356.
- Preprocessor output

  Use the option `--preprocess` to direct preprocessor output to a named file, see *--preprocess*, page 263.

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 that backslashes can also be used. In this case, 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.

__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 353, and *--no_path_in_file_macros*, page 257.

__BIG_ENDIAN__

Description

An integer that reflects the setting of the `--endian` option and is defined to `1` when the byte order for data is big-endian. The symbol is defined to `0` when the byte order for data is little-endian.

__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 `--core` option and is defined to `1` for the RXv1 architecture or `2` for the RXv2 architecture.

__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 `199711L`. This symbol can be used with `#ifdef` to detect whether the compiler accepts C++ code. It is particularly useful when creating header files that are to be shared by C and C++ code.

This symbol is required by Standard C.

__DATA_MODEL__

Description

An integer that identifies the data model in use. The value reflects the setting of the `--data_model` option and is defined to `__DATA16__`, `__DATA24__`, or `__DATA32__`. These symbolic names can be used when testing the `__DATA_MODEL__` symbol.

__DATE__

Description

A string that identifies the date of compilation, which is returned in the form "`Mmm dd yyyy`", for example "`Oct 30 2014`"

This symbol is required by Standard C.

__embedded_cplusplus

Description

An integer which is defined to `1` when the compiler runs in any of the C++ modes, otherwise the symbol is undefined. This symbol can be used with `#ifdef` to detect whether the compiler accepts C++ code. It is particularly useful when creating header files that are to be shared by C and C++ code.

This symbol is required by Standard C.

__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 352, and *--no_path_in_file_macros*, page 257.

__FPU__

Description

An integer that is set to `1` when the code is compiled with support for a hardware floating point unit, and to `0` otherwise.

__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.

This symbol is required by Standard C.

See also -*e*, page 249 and *__PRETTY_FUNCTION__*, page 355.

__FUNCTION__

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 249 and *__PRETTY_FUNCTION__*, page 355.

__IAR_SYSTEMS_ICC__

Description An integer that identifies the IAR compiler platform. The current value is 8. Note that the number could be higher in a future version of the product. This symbol can be tested with `#ifdef` to detect whether the code was compiled by a compiler from IAR Systems.

__ICCRX__

Description An integer that is set to `1` when the code is compiled with the IAR C/C++ Compiler for RX.

__INTSIZE__

Description An integer that identifies the size of the data type `int`. The value reflects the setting of the `--int` option and is defined to `16` or `32`.

__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 `--endian` option and is defined to `1` when the byte order for data is little-endian. The symbol is defined to `0` when the byte order for data is big-endian.

__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 249 and *__func__*, page 353.

__ROPI__

Description

An integer that is set to `1` when the code is compiled with the compiler option `--ropi`, and to `0` otherwise.

__RWPI__

Description

An integer that is set to `1` when the code is compiled with the compiler option `--rwpi`, and to `0` otherwise.

__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_VERSION__

Description

An integer that identifies the version of the C standard in use. The symbol expands to `199901L`,unless the `--c89` compiler option is used in which case the symbol expands to `199409L`. This symbol does not apply in EC++ mode.

This symbol is required by Standard C.

__SUBVERSION__

Description

An integer that identifies the subversion number of the compiler version number, for example 3 in 1.2.3.4.

__TIME__

Description

A string that identifies the time of compilation in the form `"hh:mm:ss"`.

This symbol is required by Standard C.

__VER__

Description

An integer that identifies the version number of the IAR compiler in use. The value of the number is calculated in this way: `(100 * the major version number + the minor version number)`. For example, for compiler version 3.34, 3 is the major version number and 34 is the minor version number. Hence, the value of `__VER__` is 334.

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 that 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 *Assert*, page 136.

#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.

Library functions

- Library overview
- IAR DLIB Library

For detailed reference information about the library functions, see the online help system.

Library overview

The IAR DLIB Library is a complete library, compliant with Standard C and C++. This library also supports floating-point numbers in IEEE 754 format and it can be configured to include different levels of support for locale, file descriptors, multibyte characters, etc.

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* in this guide.

HEADER FILES

Your application program gains access to library definitions through header files, which it incorporates using the `#include` directive. The definitions are divided into several different header files, each covering a particular functional area, letting you include just those that are required.

It is essential to include the appropriate header file before making any reference to its definitions. Failure to do so can cause the call to fail during execution, or generate error or warning messages at compile time or link time.

LIBRARY OBJECT FILES

Most of the library definitions can be used without modification, that is, directly from the library object files that are supplied with the product. For information about how to set up a runtime library, see *Setting up the runtime environment*, page 108. The linker

will include only those routines that are required—directly or indirectly—by your application.

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 any of these more accurate versions, use the `--redirect` 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 library are reentrant, but the following functions and parts are not reentrant because they need static data:

- Heap functions—`malloc`, `free`, `realloc`, `calloc`, and the C++ operators `new` and `delete`
- Locale functions—`localeconv`, `setlocale`
- Multibyte functions—`mbrlen`, `mbrtowc`, `mbsrtowc`, `mbtowc`, `wcrtomb`, `wcsrtomb`, `wctomb`
- Rand functions—`rand`, `srand`
- Time functions—`asctime`, `localtime`, `gmtime`, `mktime`
- The miscellaneous functions `atexit`, `strerror`, `strtok`
- Functions that use files or the heap in some way. This includes `scanf`, `sscanf`, `getchar`, and `putchar`. In addition, if you are using the options `--enable_multibyte` 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.

IAR DLIB Library

The IAR DLIB Library provides most of the important C and C++ library definitions that apply to embedded systems. These are of the following types:

- Adherence to a free-standing implementation of Standard C. The library supports most of the hosted functionality, but you must implement some of its base functionality. For additional information, see the chapter *Implementation-defined behavior for Standard C* in this guide.
- 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* in this guide.
- Runtime support libraries; for example low-level floating-point routines.
- Intrinsic functions, allowing low-level use of RX features. See the chapter *Intrinsic functions* for more information.

In addition, the IAR DLIB Library includes some added C functionality, see *Added C functionality*, page 365.

C HEADER FILES

This section lists the header files specific to the DLIB library C definitions. Header files may additionally contain target-specific definitions; these are documented in the chapter *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

Table 37: Traditional Standard C header files—DLIB

Header file	Usage
inttypes.h	Defining formatters for all types defined in stdint.h
iso646.h	Using Amendment 1—iso646.h standard header
limits.h	Testing integer type properties
locale.h	Adapting to different cultural conventions
math.h	Computing common mathematical functions
setjmp.h	Executing non-local goto statements
signal.h	Controlling various exceptional conditions
stdarg.h	Accessing a varying number of arguments
stdbool.h	Adds support for the bool data type in C.
stddef.h	Defining several useful types and macros
stdint.h	Providing integer characteristics
stdio.h	Performing input and output
stdlib.h	Performing a variety of operations
string.h	Manipulating several kinds of strings
tgmath.h	Type-generic mathematical functions
time.h	Converting between various time and date formats
uchar.h	Unicode functionality (IAR extension to Standard C)
wchar.h	Support for wide characters
wctype.h	Classifying wide characters

Table 37: 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 Embedded C++ library.
- The C++ standard template library (STL) header files
  The header files that constitute STL for the Extended Embedded 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 Embedded C++:

Header file	Usage
`complex`	Defining a class that supports complex arithmetic
`fstream`	Defining several I/O stream classes that manipulate external files
`iomanip`	Declaring several I/O stream manipulators that take an argument
`ios`	Defining the class that serves as the base for many I/O streams classes
`iosfwd`	Declaring several I/O stream classes before they are necessarily defined
`iostream`	Declaring the I/O stream objects that manipulate the standard streams
`istream`	Defining the class that performs extractions
`new`	Declaring several functions that allocate and free storage
`ostream`	Defining the class that performs insertions
`sstream`	Defining several I/O stream classes that manipulate string containers
`streambuf`	Defining classes that buffer I/O stream operations
`string`	Defining a class that implements a string container
`strstream`	Defining several I/O stream classes that manipulate in-memory character sequences

Table 38: C++ header files

The C++ standard template library (STL) header files

The following table lists the standard template library (STL) header files that can be used in Extended Embedded C++:

Header file	Description
`algorithm`	Defines several common operations on sequences
`deque`	A deque sequence container
`functional`	Defines several function objects
`hash_map`	A map associative container, based on a hash algorithm
`hash_set`	A set associative container, based on a hash algorithm
`iterator`	Defines common iterators, and operations on iterators
`list`	A doubly-linked list sequence container
`map`	A map associative container
`memory`	Defines facilities for managing memory
`numeric`	Performs generalized numeric operations on sequences

Table 39: <Standard template library header files

Header file	Description
queue	A queue sequence container
set	A set associative container
slist	A singly-linked list sequence container
stack	A stack sequence container
utility	Defines several utility components
vector	A vector sequence container

Table 39: <Standard template library 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`.

This table shows the new header files:

Header file	Usage
cassert	Enforcing assertions when functions execute
cctype	Classifying characters
cerrno	Testing error codes reported by library functions
cfloat	Testing floating-point type properties
cinttypes	Defining formatters for all types defined in `stdint.h`
climits	Testing integer type properties
clocale	Adapting to different cultural conventions
cmath	Computing common mathematical functions
csetjmp	Executing non-local goto statements
csignal	Controlling various exceptional conditions
cstdarg	Accessing a varying number of arguments
cstdbool	Adds support for the `bool` data type in C.
cstddef	Defining several useful types and macros
cstdint	Providing integer characteristics
cstdio	Performing input and output
cstdlib	Performing a variety of operations
cstring	Manipulating several kinds of strings
ctime	Converting between various time and date formats

Table 40: New Standard C header files—DLIB

Header file	Usage
cwchar	Support for wide characters
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`.

ADDED C FUNCTIONALITY

The IAR DLIB Library includes some added C functionality.

The following include files provide these features:

- `fenv.h`
- `stdio.h`
- `stdlib.h`
- `string.h`
- `time.h`

fenv.h

In `fenv.h`, trap handling support for floating-point numbers is defined with the functions `fegettrapenable` and `fegettrapdisable`.

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.

In both interfaces, `time_t` starts at the year 1970.

The interfaces are defined in the system header file `time.h`.

An application can use either interface, and even mix them by explicitly using the 32- or 64-bit variants. By default, the library and the header redirect `time_t`, `time` etc. to the 32-bit variants. However, to explicitly redirect them to their 64-bit variants, define `_DLIB_TIME_USES_64` in front of the inclusion of `time.h` or `ctime`.

See also, *Time*, page 133.

`clock_t` is represented by a 32-bit integer type.

SYMBOLS USED INTERNALLY BY THE LIBRARY

The following symbols are used by the library, which means that they are visible in library source files, etc:

`__assignment_by_bitwise_copy_allowed`

This symbol determines properties for class objects.

`__code`

This symbol is used as a memory attribute internally by the compiler, and might have to be used as an argument in certain templates.

`__constrange()`

Determines the allowed range for a parameter to an intrinsic function and that the parameter must be of type `const`.

`__construction_by_bitwise_copy_allowed`

This symbol determines properties for class objects.

`__has_constructor`, `__has_destructor`

These symbols determine properties for class objects and they function like the `sizeof` operator. The symbols are true when a class, base class, or member (recursively) has a user-defined constructor or destructor, respectively.

`__memory_of`

Determines the class memory. A class memory determines which memory a class object can reside in. This symbol can only occur in class definitions as a class memory.

Note: The symbols are reserved and should only be used by the library.

Use the compiler option `--predef_macros` to determine the value for any predefined symbols.

The linker configuration file

- Overview
- 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 80.

Overview

To link and locate an application in memory according to your requirements, ILINK needs information about how to handle sections and how to place them into the available memory regions. In other words, ILINK needs a *configuration*, passed to it by means of the *linker configuration file*.

This file consists of a sequence of directives and typically, provides facilities for:

- Defining available addressable memories

  giving the linker information about the maximum size of possible addresses and defining the available physical memory, as well as dealing with memories that can be addressed in different ways.
- Defining the regions of the available memories that are populated with ROM or RAM

  giving the start and end address for each region.
- Section groups

dealing with how to group sections into blocks and overlays depending on the section requirements.

- Defining how to handle initialization of the application

  giving information about which sections that are to be initialized, and how that initialization should be made.

- Memory allocation

  defining where—in what memory region—each set of sections should be placed.

- Using symbols, expressions, and numbers

  expressing addresses and sizes, etc, in the other configuration directives. The symbols can also be used in the application itself.

- Structural configuration

  meaning that you can include or exclude directives depending on a condition, and to split the configuration file into several different files.

Comments can be written either as C comments (`/*...*/`) or as C++ comments (`//...`).

Defining memories and regions

ILINK needs information about the available memory spaces, or more specifically it needs information about:

- The maximum size of possible addressable memories

  The `define memory` directive defines a *memory space* with a given size, which is the maximum possible amount of addressable memory, not necessarily physically available. See *define memory directive*, page 371.

- Available physical memory

  The `define region` directive defines a region in the available memories in which specific sections of application code and sections of application data can be placed. See *define region directive*, page 371.

  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 373.

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 389.

Parameters

size_expr	Specifies how many *units* the memory space contains; always counted from address zero.
bitsize_expr	Specifies how many bits each unit contains.
bytesize_expr	Specifies how many bytes each unit contains. Each byte contains 8 bits.

Description

The `define memory` directive defines a *memory space* with a given size, which is the maximum possible amount of addressable memory, not necessarily physically available. This sets the limits for the possible addresses to be used in the linker configuration file. For many microcontrollers, one memory space is sufficient. However, some microcontrollers require two or more. For example, a Harvard architecture usually requires two different memory spaces, one for code and one for data. If only one memory is defined, the memory name is optional. If no *unit-size* is given, the unit contains 8 bits.

Example

```
/* Declare the memory space Mem of four Gigabytes */
define memory Mem with size = 4G;
```

define region directive

Syntax

```
define region name = region-expr;
```

where *region-expr* is a region expression, see also *Regions*, page 372.

Parameters

name	The name of the region.

Description

The `define region` directive defines a region in which specific sections of code and sections of data can be placed. A region consists of one or several memory ranges, where each memory range consists of a continuous sequence of bytes in a specific memory.

Several ranges can be combined by using region expressions. Note that those ranges do not need to be consecutive or even in the same memory.

Example

```
/* Define the 0x10000-byte code region ROM located at address
   0x10000 in memory Mem */
define region ROM = Mem:[from 0x10000 size 0x10000];
```

Regions

A *region* is s a set of non-overlapping memory ranges. A *region expression* is built up out of *region literals* and set operations (union, intersection, and difference) on regions.

Region literal

Syntax

```
[ memory-name: ][from expr { to expr | size expr }

  [ repeat expr [ displacement expr ]]]
```

where *expr* is an expression, see *expressions*, page 389.

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 371, and *Region expression*, page 373.

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 371

where *region-literal* is a region literal, see *Region literal*, page 372

and where *empty-region* is an empty region, see *Empty region*, page 374.

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 373.

Section handling

Section handling describes how ILINK should handle the sections of the execution image, which means:

- Placing sections in regions

  The `place at` and `place into` directives place sets of sections with similar attributes into previously defined regions. See *place at directive*, page 382 and *place in directive*, page 383.

- Making sets of sections with special requirements

  The `block` directive makes it possible to create empty sections with specific sizes and alignments, sequentially sorted sections of different types, etc.

  The `overlay` directive makes it possible to create an area of memory that can contain several overlay images. See *define block directive*, page 375, and *define overlay directive*, page 377.

- 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 378 and *do not initialize directive*, page 381.

- 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 381.

define block directive

Syntax

```
define [movable] block name
  [ with param, param... ]
{
  extended-selectors
}
[except
 {
   section_selectors
  }];
```

where *param* can be one of:

```
size = expr
maximum size = expr
alignment = expr
fixed order
static base [basename]
```

and where the rest of the directive selects sections to include in the block, see *Section selection*, page 383.

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.
`maximum size`	Specifies an upper limit for the size of the block. An error is generated if the sections in the block do not fit.
`alignment`	Specifies a minimum alignment for the block. If any section in the block has a higher alignment than the minimum alignment, the block will have that alignment.
`fixed order`	Places sections in fixed order; if not specified, the order of the sections will be arbitrary.
`static base` *[basename]*	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.

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.

Example

```
/* Create a 0x1000-byte block for the heap */
define block HEAP with size = 0x1000, alignment = 4 { };
```

See also

Interaction between the tools and your application, page 196. See *define overlay directive*, page 377 for an Accessing example.

define overlay directive

Syntax

```
define overlay name [ with param, param... ]
{
  extended-selectors;
}
[except
 {
   section_selectors
  }];
```

For information about extended selectors and except clauses, see *Section selection*, page 383.

Parameters

name	The name of the overlay.
`size`	Customizes the size of the overlay. By default, the size of a overlay is the sum of its parts dependent of its contents.
`maximum size`	Specifies an upper limit for the size of the overlay. An error is generated if the sections in the overlay do not fit.
`alignment`	Specifies a minimum alignment for the overlay. If any section in the overlay has a higher alignment than the minimum alignment, the overlay will have that alignment.
`fixed order`	Places sections in fixed order; if not specified, the order of the sections will be arbitrary.

Description

The `overlay` directive defines a named set of sections. In contrast to the `block` directive, the overlay directive can define the same name several times. Each definition will then be grouped in memory at the same place as all other definitions of the same name. This creates an *overlaid* memory area, which can be useful for an application that has several independent sub-applications.

Place each sub-application image in ROM and reserve a RAM overlay area that can hold all sub-applications. To execute a sub-application, first copy it from ROM to the RAM overlay. Note that ILINK does not help you with managing interdependent overlay definitions, apart from generating a diagnostic message for any reference from one overlay to another overlay.

The size of an overlay will be the same size as the largest definition of that overlay name and the alignment requirements will be the same as for the definition with the highest alignment requirements.

Note: Sections that were overlaid must be split into a RAM and a ROM part and you must take care of all the copying needed.

See also

Manual initialization, page 100.

initialize directive

Syntax

```
initialize { by copy | manually }
  [ with packing = algorithm ]
{
  section-selectors
}
[except
 {
   section_selectors
  }];
```

where the rest of the directive selects sections to include in the block. See *Section selection*, page 383.

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 consecutive bytes of the same value.

`lz77` - Compresses with the Lempel-Ziv-77 algorithm. This method handles a larger variety of inputs well, but has a slightly larger decompressor.

`bwt` - Compresses with the Burrows-Wheeler algorithm. This method improves the `packbits` method by transforming blocks of data before they are compressed.

`lzw` - Compresses with the Lempel-Ziv-Welch algorithm. This method uses a dictionary to store byte patterns in the data.

`auto` - Similar to `smallest`, but ILINK chooses between `none`, `packbits`, and `lz77`. This is the default method for `initialize by copy`.

`smallest` - ILINK estimates the resulting size using each packing method (except for `auto`), and then chooses the packing method that produces the smallest estimated size. Note that the size of the decompressor is also included.

Description

The `initialize` directive splits the initialization section into one section holding the initializers and another section holding the initialized data. The section holding the initialized data retains the original section name and the section holding the initializers 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`) or `smallest`, 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. The decompressors for `bwt` and `lzw` use significantly more execution time and RAM

than the decompressors for the other methods. Approximately 9 Kbytes of stack space is needed for `bwt` and 3.5 Kbytes for `lzw`.

When initializers are compressed, the exact size of the compressed initializers is unknown until the exact content of the uncompressed data is known. If this data contains other addresses, and some of these addresses are dependent on the size of the compressed initializers, the linker fails with error Lp017. To avoid this, place compressed initializers last, or in a memory region together with sections whose addresses do not need to be known.

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 377.

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 86, and *do not initialize directive*, page 381.

do not initialize directive

Syntax

```
do not initialize
{
  section-selectors
}
[except
 {
   section-selectors
 }];
```

For information about extended selectors and except clauses, see *Section selection*, page 383.

Description

The `do not initialize` directive specifies the sections that should not be initialized by the system startup code. The directive can only be used on `zeroinit` sections.

The compiler keyword `__no_init` places variables into sections that must be handled by a `do not initialize` directive.

Example

```
/* Do not initialize read-write sections whose name ends with
   _noinit at program start */
do not initialize { rw section .*_noinit };
place in RAM { rw section .*_noinit };
```

See also

Initialization at system startup, page 86, and *initialize directive*, page 378.

keep directive

Syntax

```
keep
{
  section-selectors
}
[except
 {
   section-selectors
 }];
```

For information about extended selectors and except clauses, see *Section selection*, page 383.

Description

The `keep` directive specifies that all selected sections should be kept in the executable image, even if there are no references to the sections.

Example

```
keep { section .keep* } except {section .keep};
```

place at directive

Syntax

```
[ "name": ]
place at { address [ memory: ] expr | start of region_expr |
           end of region_expr }
{
  extended-selectors
}
[except
 {
   section-selectors
 }];
```

For information about extended selectors and except clauses, see *Section selection*, page 383.

Parameters

memory: *expr*	A specific address in a specific memory. The address must be available in the supplied memory defined by the `define memory` directive. The memory specifier is optional if there is only one memory.
`start of` *region_expr*	A region expression 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.

Description

The `place at` directive places sections and blocks either at a specific address or, at the beginning or the end of a region. The same address cannot be used for two different `place at` directives. It is also not possible to use an empty region in a `place at` directive. If placed in a region, the sections and blocks will be placed before any other sections or blocks placed in the same region with a `place in` directive.

The *name*, if specified, is used in the map file and in some log messages.

Example

```
/* Place the read-only section .startup at the beginning of the
   code_region */
"START": place at start of ROM { readonly section .startup };
```

See also

place in directive, page 383.

place in directive

Syntax

```
[ "name": ]
place in region-expr
{
  extended-selectors
}
 [except{
   section-selectors
 }];
```

where *region-expr* is a region expression, see also *Regions*, page 372.

and where the rest of the directive selects sections to include in the block. See *Section selection*, page 383.

Description

The `place in` directive places sections and blocks in a specific region. The sections and blocks will be placed in the region in an arbitrary order.

To specify a specific order, use the `block` directive. The region can have several ranges.

The *name*, if specified, is used in the map file and in some log messages.

Example

```
/* Place the read-only sections in the code_region */
"ROM": place in ROM { readonly };
```

See also

place at directive, page 382.

Section selection

The purpose of *section selection* is to specify—by means of *section selectors* and *except clauses*—the sections that an ILINK directive should be applied to. All sections that match one or more of the section selectors will be selected, and none of the sections selectors in the except clause, if any. Each section selector can match sections on section attributes, section name, and object or library name.

Some directives provide functionality that requires more detailed selection capabilities, for example directives that can be applied on both sections and blocks. In this case, the *extended-selectors* are used.

section-selectors

Syntax

```
{ [ section-selector ][, section-selector...] }
```

where *section-selector* is:

```
 [ section-attribute ][ section-type ][ section sectionname ]
    [object {module | filename } ]
```

where *section-attribute* is:

```
 [ ro [ code | data ] | rw [ code | data ] | zi ]
```

and where `ro`, `rw`, and `zi` also can be `readonly`, `readwrite`, and `zeroinit`, respectively.

And *section-type* is:

```
 [ preinit_array | init_array ]
```

Parameters

`ro` or `readonly`	Read-only sections.
`rw` or `readwrite`	Read/write sections.
`zi` or `zeroinit`	Zero-initialized sections. These sections have no content and should possibly be initialized with zeros during system startup.
`code`	Sections that contain code.
`data`	Sections that contain data.
`preinit_array`	Sections of the ELF section type `SHT_PREINIT_ARRAY`.
`init_array`	Sections of the ELF section type `SHT_INIT_ARRAY`.
sectionname	The section name. Two wildcards are allowed: `?` matches any single character `*` matches zero or more characters.
module	A name in the form *objectname*`(`*libraryname*`)`. Sections from object modules where both the object name and the library name match their respective patterns are selected. An empty library name pattern selects only sections from object files.

filename	The name of an object file, a library, or an object in a library. Two wildcards are allowed: ? matches any single character * matches zero or more characters.

Description

A section selector selects all sections that match the section attribute, section type, section name, and the name of the *object*, where *object* is an object file, a library, or an object in a library. Up to three of the four conditions can be omitted. If the section attribute is omitted, any section will be selected, without restrictions on the section attribute. If the section type is omitted, sections of any type will be selected.

If the section name part or the object name part is omitted, sections will be selected without restrictions on the section name or object name, respectively.

It is also possible to use only `{ }` without any section selectors, which can be useful when defining blocks.

Note that a section selector with narrower scope has higher priority than a more generic section selector.

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:

- It specifies a section type and the other one does not
- It specifies a section name or object name with no wildcards and the other one does not
- There could be sections that match the other selector that also match this one, and the reverse is not true.

Selector 1	Selector 2	More specific
`section "foo*"`	`section "f*"`	Selector 1
`section "*x"`	`section "f*"`	Neither
`ro code section "f*"`	`ro section "f*"`	Selector 1
`init_array`	`ro section "xx"`	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 378, *do not initialize directive*, page 381, and *keep directive*, page 381.

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 375, *define overlay directive*, page 377, and *place at directive*, page 382.

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 388, and *export directive*, page 389.

- Use expressions and numbers

  In the linker configuration file, expressions and numbers are used for specifying addresses, sizes, etc. See *expressions*, page 389.

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 88.

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 389 and *Interaction between ILINK and the application*, page 104.

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 390

where *symbol* is a defined symbol, see *define symbol directive*, page 388 and *--config_def*, page 274

and where *func-operator* is one of these function-like operators:

minimum(*expr*,*expr*)	Returns the smallest of the two parameters.
maximum(*expr*,*expr*)	Returns the largest of the two parameters.
isempty(*r*)	Returns True if the region is empty, otherwise False.
isdefinedsymbol(*expr-symbol*)	Returns True if the expression symbol is defined, otherwise False.
start(*r*)	Returns the lowest address in the region.
end(*r*)	Returns the highest address in the region.
size(*r*)	Returns the size of the complete region.

where *expr* is an expression, and *r* is a region expression, see *Region expression*, page 373.

Description

In the linker configuration file, an expression is a 65-bit value with the range -2^64 to 2^64. The expression syntax closely follows C syntax with some minor exceptions. There are no assignments, casts, pre- or post-operations, and no address operations (*, &, [], ->, and .). Some operations that extract a value from a region expression, etc, use a syntax resembling that of a function call. A boolean expression returns 0 (False) or 1 (True).

numbers

Syntax

nr [*nr-suffix*]

where nr is either a decimal number or a hexadecimal number (0x... or 0X...).

and where *nr-suffix* is one of:

```
K           /* Kilo = (1 << 10) 1024 */
M           /* Mega = (1 << 20) 1048576 */
G           /* Giga = (1 << 30) 1073741824 */
T           /* Tera = (1 << 40) 1099511627776 */
P           /* Peta = (1 << 50) 1125899906842624 */
```

Description

A number can be expressed either by normal C means or by suffixing it with a set of useful suffixes, which provides a compact way of specifying numbers.

Example

`1024` is the same as `0x400`, which is the same as `1K`.

Structural configuration

The structural directives provide means for creating structure within the configuration, such as:

- Conditional inclusion

  An `if` directive includes or excludes other directives depending on a condition, which makes it possible to have directives for several different memory configurations in the same file. See *if directive*, page 391.

- 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 392.

if directive

Syntax

```
if (expr) {
  directives
[ } else if (expr) {
  directives ]
[ } else {
  directives ]
}
```

where *expr* is an expression, see *expressions*, page 389.

Parameters

directives	Any ILINK directive.

Description

An `if` directive includes or excludes other directives depending on a condition, which makes it possible to have directives for several different memory configurations, for example both a banked and non-banked memory configuration, in the same file.

The directives inside an `if` part, `else if` part, or an `else` part are syntax checked and processed regardless of whether the conditional expression was true or false, but only the directives in the part where the conditional expression was true, or the `else` part if none of the conditions were true, will have any effect outside the `if` directive. The `if` directives can be nested.

Example

See *Empty region*, page 374.

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.

Section reference

- Section reference
- Descriptions of sections and blocks

For more information about sections, see the chapter *Modules and sections*, page 80.

Summary of sections

This table lists the ELF sections and blocks that are used by the IAR build tools:

Section	Description
`.data16.bss`	Holds zero-initialized `__data16` static and global variables.
`.data16.data`	Holds `__data16` static and global initialized variables.
`.data16.data_init`	Holds initial values for `.data16.data` sections when the linker directive `initialize` is used.
`.data16.noinit`	Holds `__no_init __data16` static and global variables.
`.data16.rodata`	Holds `__data16` constant data.
`.data24.bss`	Holds zero-initialized `__data24` static and global variables.
`.data24.data`	Holds `__data24` static and global initialized variables.
`.data24.data_init`	Holds initial values for `.data24.data` sections when the linker directive `initialize` is used.
`.data24.noinit`	Holds `__no_init __data24` static and global variables.
`.data24.rodata`	Holds `__data24` constant data.
`.data32.bss`	Holds zero-initialized `__data32` static and global variables.
`.data32.data`	Holds `__data32` static and global initialized variables.
`.data32.data_init`	Holds initial values for `.data32.data` sections when the linker directive `initialize` is used.
`.data32.noinit`	Holds `__no_init __data32` static and global variables.
`.data32.rodata`	Holds `__data32` constant data.
`DIFUNCT`	Holds pointers to code, typically C++ constructors, that should be executed by the system startup code before `main` is called.
`__DLIB_PERTHREAD`	Holds variables that contain static states for DLIB modules.

Table 42: Section summary

Section	Description
EARLYDIFUNCT	Holds pointers to code objects that require early initialization, typically stream I/O, that should be executed by the system startup code before `main` is called.
HEAP	Holds the heap used for dynamically allocated data.
.iar.dynexit	Holds the `atexit` table.
.init_array	Holds a table of dynamic initialization functions.
.inttable	Holds all interrupt vectors except for non-maskable interrupts
ISTACK	Holds the supervisor mode stack.
.nmivec	Holds the reset and non-maskable interrupt vectors.
.sbrel.bss	Holds zero-initialized `__sbrel` static and global variables.
.sbrel.data	Holds `__sbrel` static and global initialized variables.
.sbrel.data_init	Holds initial values for `.sbrel.data` sections when the linker directive `initialize` is used.
.sbrel.noinit	Holds `__no_init __sbrel` static and global variables.
.switch.rodata	Holds tables for switch statements.
.text	Holds the program code.
.preinit_array	Holds a table of dynamic initialization functions.
.textrw	Holds `__ramfunc` declared program code.
.textrw_init	Holds initializers for the `.textrw` declared section.
USTACK	Holds the user mode stack.

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 83.

.data16.bss

Description Holds zero-initialized `__data16` static and global variables.

Memory placement This section must be placed in the lowest 32 Kbytes of RAM memory.

See also *Memory types*, page 60.

.data16.data

Description Holds `__data16` static and global initialized variables. In object files, this includes the initial values. When the linker directive `initialize` is used, a corresponding `.data16.data_init` section is created for each `.data16.data` section, holding the possibly compressed initial values.

Memory placement This section must be placed in the lowest 32 Kbytes of RAM memory.

See also *Memory types*, page 60.

.data16.data_init

Description Holds the possibly compressed initial values for `.data16.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 ROM memory.

See also *Memory types*, page 60.

.data16.noinit

Description	Holds static and global `__no_init` `__data16` variables.
Memory placement	This section must be placed in the lowest 32 Kbytes of RAM memory.
See also	*Memory types*, page 60.

.data16.rodata

Description	Holds `__data16` constant data. This can include constant variables, string and aggregate literals, etc.
Memory placement	This section must be placed in the highest 32 Kbytes of ROM memory.
See also	*Memory types*, page 60.

.data24.bss

Description	Holds zero-initialized `__data24` static and global variables.
Memory placement	This section must be placed in the lowest or highest 8 Mbytes of memory.
See also	*Memory types*, page 60.

.data24.data

Description	Holds `__data24` static and global initialized variables. In object files, this includes the initial values. When the linker directive `initialize` is used, a corresponding `.data24.data_init` section is created for each `.data24.data` section, holding the possibly compressed initial values.
Memory placement	This section must be placed in the lowest or highest 8 Mbytes of memory.
See also	*Memory types*, page 60.

.data24.data_init

Description	Holds the possibly compressed initial values for `.data24.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 ROM memory.
See also	*Memory types*, page 60.

.data24.noinit

Description	Holds static and global `__no_init` `__data24` variables.
Memory placement	This section must be placed in the lowest or highest 8 Mbytes of memory.
See also	*Memory types*, page 60.

.data24.rodata

Description	Holds `__data24` constant data. This can include constant variables, string and aggregate literals, etc.
Memory placement	This section must be placed in the lowest or highest 8 Mbytes of ROM memory.
See also	*Memory types*, page 60.

.data32.bss

Description	Holds zero-initialized `__data32` static and global variables.
Memory placement	This section can be placed anywhere in RAM memory.
See also	*Memory types*, page 60.

.data32.data

Description	Holds `__data32` static and global initialized variables. In object files, this includes the initial values. When the linker directive `initialize` is used, a corresponding `.data32.data_init` section is created for each `.data32.data` section, holding the possibly compressed initial values.
Memory placement	This section can be placed anywhere in RAM memory.
See also	*Memory types*, page 60.

.data32.data_init

Description	Holds the possibly compressed initial values for `.data32.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 ROM memory.
See also	*Memory types*, page 60.

.data32.noinit

Description	Holds static and global `__no_init __data32` variables.
Memory placement	This section can be placed anywhere in RAM memory.
See also	*Memory types*, page 60.

.data32.rodata

Description	Holds `__data32` constant data. This can include constant variables, string and aggregate literals, etc.
Memory placement	This section can be placed anywhere in ROM memory.
See also	*Memory types*, page 60.

DIFUNCT

Description	Holds the dynamic initialization vector used by C++.
Memory placement	This section can be placed anywhere in ROM memory.

__DLIB_PERTHREAD

Description	Holds thread-local static and global initialized variables used by the main thread. This section is placed automatically. If you change the placement, you must not change its initialization. The initialization of this section must be controlled with the `initialize` directive.
Memory placement	This section must be placed in the lowest 32 Kbytes of memory.

See also — *Managing a multithreaded environment*, page 137.

EARLYDIFUNCT

Description — Holds the dynamic initialization vector used by C++ for objects that require early initialization, typically stream I/O.

Memory placement — This section can be placed anywhere in ROM memory.

HEAP

Description — Holds the heap used for dynamically allocated data, in other words data allocated by `malloc` and `free`, and in C++, `new` and `delete`.

Memory placement — This section can be placed anywhere in RAM memory.

See also — *Setting up heap memory*, page 99.

.iar.dynexit

Description — Holds the table of calls to be made at exit.

Memory placement — This section can be placed anywhere in ROM memory.

See also — *Setting up the atexit limit*, page 100.

.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.

.inttable

Description — Holds the interrupt vector table generated by the use of the `__interrupt` extended keyword in combination with the `#pragma vector` directive.

Memory placement — This section can be placed anywhere in ROM memory.

ISTACK

Description	Block that holds the supervisor mode stack.
Memory placement	This block can be placed anywhere in RAM memory.
See also	*Setting up stack memory*, page 99.

.nmivec

Description	Holds the non-maskable interrupt vector table and the reset vector.
Memory placement	This section must be placed in the memory range `0xFFFFFF80–0xFFFFFFFF`.

.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 399.

.sbrel.bss

Description	Holds zero-initialized `__sbrel` static and global variables.
Memory placement	This section can be placed anywhere in RAM memory.
See also	*Memory types*, page 60.

.sbrel.data

Description	Holds `__sbrel` static and global initialized variables. In object files, this includes the initial values. When the linker directive `initialize` is used, a corresponding `.sbrel.data_init` section is created for each `.sbrel.data` section, holding the possibly compressed initial values.
Memory placement	This section can be placed anywhere in RAM memory.

See also — *Memory types*, page 60.

.sbrel.data_init

Description — Holds the possibly compressed initial values for `.sbrel.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 ROM memory.

See also — *Memory types*, page 60.

.sbrel.noinit

Description — Holds static and global `__no_init __sbrel` variables.

Memory placement — This section can be placed anywhere in RAM memory.

See also — *Memory types*, page 60.

.switch.rodata

Description — Holds tables for switch statements. Not all switch statements generate a table, but those who do will place the table in this section.

Memory placement — This section can be placed anywhere in ROM memory.

.text

Description — Holds program code.

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 RAM memory.

See also — *__ramfunc*, page 316.

.textrw_init

Description	Holds initializers for the `.textrw` declared sections.
Memory placement	This section can be placed anywhere in RAM memory.
See also	*__ramfunc*, page 316.

USTACK

Description	Block that holds the user mode stack, referred to by the `USP` stack pointer.
Memory placement	This block can be placed anywhere in RAM memory.
See also	*Setting up stack memory*, page 99.

The stack usage control file

- Overview
- Stack usage control directives
- Syntactic components

Before you read this chapter, see *Stack usage analysis*, page 88.

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)"
"ostream::flush()"                        // EC++
"std::ostream::flush()"                   // C++
"operator <<(ostream &, char const *)"
"void _Sort<#*>(#*, #*, #*)"
```

Stack usage control directives

This section gives detailed reference information about each stack usage control directive.

function directive

Syntax

```
[ override ] function [ category ] function-spec : stack-size
[ , call-info... ];
```

Parameters

category	See *category*, page 407
function-spec	See *function-spec*, page 407
call-info	See *call-info*, page 408
stack-size	See *stack-size*, page 408

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] nmi: 44
```

exclude directive

Syntax

```
exclude function-spec [, function-spec... ];
```

Parameters

function-spec	See *function-spec*, page 407

Description

Excludes the specified functions, and call trees originating with them, from stack usage calculations.

Example

```
exclude fun1, fun2;
```

possible calls directive

Syntax

```
possible calls calling-func : called-func [ , called-func... ];
```

Parameters

calling-func	See *function-spec*, page 407
called-func	See *function-spec*, page 407

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 afun: bfun, cfun;
```

See also

calls, page 323.

call graph root directive

Syntax

```
call graph root [ category ] : function-spec [, function-spec... ];
```

Parameters

category	See *category*, page 407
function-spec	See *function-spec*, page 407

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]: fun1, fun2;
```

See also

call_graph_root, page 324.

max recursion depth directive

Syntax

```
max recursion depth function-spec : size;
```

Parameters

function-spec	See *function-spec*, page 407
size	See *size*, page 409

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 fun1: 10;
```

no calls from directive

Syntax

```
no calls from module-spec to function-spec [, function-spec... ];
```

Parameters

function-spec	See *function-spec*, page 407
module-spec	See *module-spec*, page 407

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 fun1, fun2;
```

Syntactic components

The stack usage control directives use some syntactical components. These are described below.

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]
```

function-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 the function-spec does not match a symbol in the program, a warning is emitted. Prefixing with `?` suppresses this warning.

Example

function-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 that 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 function-spec [ , function-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
```

Description

Specifies the size of a stack frame.

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 Archive Tool—iarchive

The IAR Archive Tool, `iarchive`, can create a library (an archive) file from several ELF object files. You can also use `iarchaive` to manipulate ELF libraries.

A library file contains several relocatable ELF object modules, each of which can be independently used by a linker. In contrast with object modules specified directly to the linker, each module in a library is only included if it is needed.

For information about how to build a library in the IDE, see the *IDE Project Management and Building Guide.*

INVOCATION SYNTAX

The invocation syntax for the archive builder is:

`iarchive` *`parameters`*

Parameters

The parameters are:

Parameter	Description
command	Command line options that define an operation to be performed. Such an option must be specified before the name of the library file.
libraryfile	The library file to be operated on.
objectfile1 ... objectfileN	The object file(s) that the specified command operates on.
options	Command line options that define actions to be performed. These options can be placed anywhere on the command line.

Table 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.
`--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

For more information, see *Descriptions of options*, page 425.

SUMMARY OF IARCHIVE OPTIONS

This table summarizes the `iarchive` options:

Command line option	Description
`-f`	Extends the command line.
`--output, -o`	Specifies the library file.
`--silent`	Sets silent operation.
`--verbose, -V`	Reports all performed operations.

Table 45: iarchive options summary

For more information, see *Descriptions of options*, page 425.

DIAGNOSTIC MESSAGES

This section lists the messages produced by `iarchive`:

La001: could not open file *filename*

`iarchive` failed to open an object file.

La002: illegal path *pathname*

The path *`pathname`* is not a valid path.

La006: too many parameters to *cmd* command

A list of object modules was specified as parameters to a command that only accepts a single library file.

La007: too few parameters to *cmd* command

A command that takes a list of object modules was issued without the expected modules.

La008: *lib* is not a library file

The library file did not pass a basic syntax check. Most likely the file is not the intended library file.

La009: *lib* has no symbol table

The library file does not contain the expected symbol information. The reason might be that the file is not the intended library file, or that it does not contain any ELF object modules.

La010: no library parameter given

The tool could not identify which library file to operate on. The reason might be that a library file has not been specified.

La011: file *file* already exists

The file could not be created because a file with the same name already exists.

La013: file confusions, *lib* given as both library and object

The library file was also mentioned in the list of object modules.

La014: module *module* not present in archive *lib*

The specified object module could not be found in the archive.

La015: internal error

The invocation triggered an unexpected error in `iarchive`.

Ms003: could not open file *filename* for writing

`iarchive` failed to open the archive file for writing. Make sure that it is not write protected.

Ms004: problem writing to file *filename*

An error occurred while writing to file *`filename`*. A possible reason for this is that the volume is full.

Ms005: problem closing file *filename*

An error occurred while closing the file *`filename`*.

The IAR ELF Tool—ielftool

The IAR ELF Tool, `ielftool`, can generate a checksum on specific ranges of memories. This checksum can be compared with a checksum calculated on your application.

The source code for `ielftool` and a Microsoft VisualStudio 2005 template project are available in the `rx\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 415.
outputfile	An absolute ELF executable image.

Table 46: ielftool parameters

See also *Rules for specifying a filename or directory as parameters*, page 236.

Example

This example fills a memory range with `0xFF` and then calculates a checksum on the same range:

```
ielftool my_input.out my_output.out --fill 0xFF;0-0xFF
          --checksum _checksum:4,crc32;0-0xFF
```

SUMMARY OF IELFTOOL OPTIONS

This table summarizes the `ielftool` command line options:

Command line option	Description
`--bin`	Sets the format of the output file to binary.
`--checksum`	Generates a checksum.
`--fill`	Specifies fill requirements.
`--ihex`	Sets the format of the output file to linear Intel hex.
`--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.

Table 47: ielftool options summary

Command line option	Description
`--srec`	Sets the format of the output file to Motorola S-records.
`--srec-len`	Restricts the number of data bytes in each S-record.
`--srec-s3only`	Restricts the S-record output to contain only a subset of records.
`--strip`	Removes debug information.
`--titxt`	Saves as TI-txt format.
`--verbose, -V`	Prints all performed operations.

Table 47: ielftool options summary (Continued)

For more information, see *Descriptions of options*, page 425.

The IAR ELF Dumper—ielfdump

The IAR ELF Dumper for RX, `ielfdumprx`, can be used for creating a text representation of the contents of a relocatable or absolute ELF file.

`ielfdumprx` 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 `ielfdumprx` is:

`ielfdumprx` *`input_file`* [*`output_file`*]

Note: `ielfdumprx` 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.

Table 48: ielfdumprx parameters

Parameter	Description
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: ielfdumprx parameters (Continued)

See also *Rules for specifying a filename or directory as parameters*, page 236.

SUMMARY OF IELFDUMP OPTIONS

This table summarizes the `ielfdumprx` command line options:

Command line option	Description
`--all`	Generates output for all input sections regardless of their names or numbers.
`--code`	Dumps all sections that contain executable code.
`-f`	Extends the command line.
`--output, -o`	Specifies an output file.
`--no_strtab`	Suppresses dumping of string table sections.
`--raw`	Uses the generic hexadecimal/ASCII output format for the contents of any selected section, instead of any dedicated output format for that section.
`--section, -s`	Generates output for selected input sections.

Table 49: ielfdumprx options summary

For more information, see *Descriptions of options*, page 425.

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 236.

Examples

This example renames the section `.example` in `input.o` to `.example2` and stores the result in `output.o`:

```
iobjmanip --rename_section .example=.example2 input.o output.o
```

SUMMARY OF IOBJMANIP OPTIONS

This table summarizes the `iobjmanip` options:

Command line option	Description
`-f`	Extends the command line.
`--remove_file_path`	Removes path information from the file symbol.
`--rename_section`	Renames a section.
`--rename_symbol`	Renames a symbol.
`--strip`	Removes debug information.

Table 51: iobjmanip options summary

For more information, see *Descriptions of options*, page 425.

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 [options]
```

Parameters

The parameters are:

Parameter	Description
inputfile	A ROM image in the form of an executable ELF file (output from linking).
options	Any of the available command line options, see *Summary of isymexport options*, page 421.
outputfile	A relocatable ELF file that can be used as input to linking, and which contains all or a selection of the absolute symbols in the input file. The output file contains only the symbols, not the actual code or data sections. A steering file can be used to control which symbols that are included, and also to rename some of the symbols if that is desired.

Table 52: isymexport parameters

For information about specifying a filename or a directory, see *Rules for specifying a filename or directory as parameters*, page 236.

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 like this:

```
$TOOLKIT_DIR$\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.
`-f`	Extends the command line.
`--ram_reserve_ranges`	Generates symbols to reserve the areas in RAM that the image uses.
`--reserve_ranges`	Generates symbols to reserve the areas in ROM and RAM that the image uses.

Table 53: isymexport options summary

For more information, see *Descriptions of options*, page 425.

STEERING FILES

A steering file can be used for controlling which symbols that are included, and also to rename some of the symbols if that is desired. In the file, you can use `show` and `hide`

directives to select which public symbols from the input file that are to be included in the output file. `rename` directives can be used for changing the names of symbols in the input file.

When you use a steering file, only actively exported symbols will be available in the output file. Thus, 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 */
```

Show directive

Syntax

```
show pattern
```

Parameters

pattern	A pattern to match against a symbol name.

Description

A symbol with a name that matches the pattern will be included in the output file unless this is overridden by a later `hide` directive.

Example

```
/* Include all public symbols ending in _pub. */
show *_pub
```

Hide directive

Syntax

```
hide pattern
```

Parameters

pattern	A pattern to match against a symbol name.

Description

A symbol with a name that matches the pattern will not be included in the output file unless this is overridden by a later `show` directive.

Example

```
/* Do not include public symbols ending in _sys. */
hide *_sys
```

Rename directive

Syntax

```
rename pattern1 pattern2
```

Parameters

pattern1	A pattern used for finding symbols to be renamed. The pattern can contain no more than one `*` or `?` wildcard character.
pattern2	A pattern used for the new name for a symbol. If the pattern contains a wildcard character, it must be of the same kind as in *pattern1*.

Description

Use this directive to rename symbols from the output file to the input file. No exported symbol is allowed to match more than one `rename` pattern.

`rename` directives can be placed anywhere in the steering file, but they are executed before any `show` and `hide` directives. Thus, if a symbol will be renamed, all `show` and `hide` directives in the steering file must refer to the new name.

If the name of a symbol matches a *pattern1* pattern that contains no wildcard characters, the symbol will be renamed *pattern2* in the output file.

If the name of a symbol matches a *pattern1* pattern that contains a wildcard character, the symbol will be renamed *pattern2* in the output file, with part of the name matching the wildcard character preserved.

Example

```
/* xxx_start will be renamed Y_start_X in the output file,
   xxx_stop will be renamed Y_stop_X in the output file. */
rename xxx_* Y_*_X
```

DIAGNOSTIC MESSAGES

This section lists the messages produced by `isymexport`:

Es001: could not open file *filename*

`isymexport` failed to open the specified file.

Es002: illegal path *pathname*

The path *`pathname`* is not a valid path.

Es003: format error: *message*

A problem occurred while reading the input file.

Es004: no input file

No input file was specified.

Es005: no output file

An input file, but no output file was specified.

Es006: too many input files

More than two files were specified.

Es007: input file is not an ELF executable

The input file is not an ELF executable file.

Es008: unknown directive: *directive*

The specified directive in the steering file is not recognized.

Es009: unexpected end of file

The steering file ended when more input was required.

Es010: unexpected end of line

A line in the steering file ended before the directive was complete.

Es011: unexpected text after end of directive

There is more text on the same line after the end of a steering file directive.

Es012: expected *text*

The specified text was not present in the steering file, but must be present for the directive to be correct.

Es013: pattern can contain at most one * or ?

Each pattern in the current directive can contain at most one `*` or one `?` wildcard character.

Es014: rename patterns have different wildcards

Both patterns in the current directive must contain exactly the same kind of wildcard. That is, both must either contain:

- No wildcards
- Exactly one `*`
- Exactly one `?`

This error occurs if the patterns are not the same in this regard.

Es014: ambiguous pattern match: *symbol* matches more than one rename pattern

A symbol in the input file matches more than one `rename` pattern.

Descriptions of options

This section gives detailed reference information about each command line option available for the different utilities.

--all

Syntax `--all`

For use with `ielfdumprx`

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`

For use with `ielftool`

Description Sets the format of the output file to binary.

To set related options, choose:

Project>Options>Output converter

--checksum

Syntax
```
--checksum
{symbol[+offset]|address}:size,algorithm[:[1|2][m][L|W][r][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	An offset to the symbol.
address	The absolute address where the checksum value should be stored.
size	The number of bytes in the checksum: `1`, `2`, or `4`; must not be larger than the size of the checksum symbol.

algorithm	The checksum algorithm used, one of: • `sum`, a byte-wise calculated arithmetic sum. The result is truncated to 8 bits. • `sum8wide`, a byte-wise calculated arithmetic sum. The result is truncated to the size of the symbol. • `sum32`, a word-wise (32 bits) calculated arithmetic sum. • `crc16`, CRC16 (generating polynomial `0x11021`); used by default. • `crc32`, CRC32 (generating polynomial `0x104C11DB7`). • `crc64iso`, CRC64iso (generating polynomial `0x1B`). • `crc64ecma`, CRC64ECMA (generating polynomial `0x42F0E1EBA9EA3693`). • `crc=`*n*, CRC with a generating polynomial of *n*.
`1|2`	If specified, can be one of: • `1` – Specifies one's complement. • `2` – Specifies two's complement.
`m`	Reverses the order of the bits within each byte when calculating the checksum.
`L|W`	Specifies the size of the unit for which a checksum should be calculated. Choose between: `L`, calculates a checksum on 32 bits in every iteration `W`, calculates a checksum on 16 bits in every iteration. If you do not specify a unit size, 8 bits will be used by default. Using these parameters does not add any additional error detection power to the checksum.
`r`	Reverses the byte order of the input data within each word of size *size*.

`i|p`	Use either `i` or `p`, if the *`start`* value is bigger than `0`. If specified, can be one of: • `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`	The address range on which the checksum should be calculated. Hexadecimal and decimal notation is allowed (for example, `0x8002-0x8FFF`).

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.

See also

Checksum calculation, page 198

To set related options, choose:

Project>Options>Linker>Checksum

--code

Syntax

```
--code
```

For use with

```
ielfdump
```

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. See *Rules for specifying a filename or directory as parameters*, page 236.
objectfile1 ... objectfileN	The object file(s) to build the library from.

For use with

```
iarchive
```

Description

Use this command to build a new library from a set of object files (modules). The object files are added to the library in the exact order that they are specified on the command line.

If no command is specified on the command line, `--create` is used by default.

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. See *Rules for specifying a filename or directory as parameters*, page 236.

objectfile1 ... objectfileN The object file(s) that the command operates on.

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.

--edit

Syntax `--edit` *steering_file*

For use with `isymexport`

Description Use this option to specify a steering file to control which symbols that are included in the `isymexport` output file, and also to rename some of the symbols if that is desired.

See also *Steering files*, page 421.

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. See *Rules for specifying a filename or directory as parameters*, page 236.

objectfile1 ... objectfileN The object file(s) that the command operates on.

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 236.

For use with

`iarchive`, `ielfdumprx`, `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.

--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. Hexadecimal and decimal notation is allowed (for example, `0x8002-0x8FFF`). Note that each address must be 4-byte aligned.

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

--ihex

Syntax `--ihex`

For use with `ielftool`

Description Sets the format of the output file to linear Intel hex.

To set related options, choose:

Project>Options>Linker>Output converter

--no_strtab

Syntax `--no_strtab`

For use with `ielfdumprx`

Description Use this option to suppress dumping of string table sections (sections of type `SHT_STRTAB`).

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 236.

For use with

`iarchive` and `ielfdumprx`.

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.

`ielfdumprx`

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. Hexadecimal and decimal notation are allowed (for example, `0x8002-0x8FFF`).

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.

--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 437.

This option is not available in the IDE.

--raw

Syntax

```
--raw
```

For use with

`ielfdumprx`

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.

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 `ielfdumprx`.

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 `ielfdumprx`.
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. See *Rules for specifying a filename or directory as parameters*, page 236.
objectfile1 ... objectfileN	The object file(s) that the command operates on.

For use with `iarchive`

Description Use this command to replace or add object files (modules) to an existing library. The object files specified on the command line either replace existing object files in the library (if they have the same name) or are appended to the library.

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 434.

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

`ielfdumprx`

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.

--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.

--silent

Syntax

```
--silent
-S (iarchive only)
```

For use with

`iarchive` and `ielftool`.

Description

Causes the tool to operate without sending any messages to the standard output stream.

By default, `ielftool` sends various messages via the standard output stream. You can use this option to prevent this. `ielftool` sends error and warning messages to the error output stream, so they are displayed regardless of this setting.

This option is not available in the IDE.

--simple

Syntax

```
--simple
```

For use with

`ielftool`

Description

Sets the format of the output file to Simple code.

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

--srec

Syntax `--srec`

For use with `ielftool`

Description Sets the format of the output file to Motorola S-records.

To set related options, choose:

Project>Options>Output converter

--srec-len

Syntax `--srec-len=`*`length`*

Parameters

`length`	The number of data bytes in each S-record.

For use with `ielftool`

Description Restricts the number of data bytes in each S-record. This option can be used in combination with the `--srec` option.

This option is not available in the IDE.

--srec-s3only

Syntax `--srec-s3only`

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.

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

To set related options, choose:

Project>Options>Linker>Output>Include debug information in output

--symbols

Syntax `--symbols` *`libraryfile`*

Parameters

`libraryfile`	The library file that the command operates on. See *Rules for specifying a filename or directory as parameters*, page 236.

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.

--titxt

Syntax
```
--titxt
```

For use with
```
ielftool
```

Description
Sets the format of the output file to TI–txt.


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 *Rules for specifying a filename or directory as parameters*, page 236.

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.

--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.

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 459. For a short overview of the differences between Standard C and C89, see *C language overview*, page 169.

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, if you use the `--enable_multibytes` compiler option, the host character set is used instead.

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 *Customizing system initialization*, page 123.

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.

Signals, their semantics, and the default handling (7.14)

In the DLIB library, 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 library, there are no implementation-defined values that correspond to a computational exception.

Signals at system startup (7.14.1.1)

In the DLIB library, there are no implementation-defined signals that are executed at system startup.

Environment names (7.20.4.5)

In the DLIB library, there are no implementation-defined environment names that are used by the `getenv` function.

The system function (7.20.4.6)

The `system` function is not supported.

J.3.3 IDENTIFIERS

Multibyte characters in identifiers (6.4.2)

Additional multibyte characters may not appear in identifiers.

Significant characters in identifiers (5.2.4.1, 6.1.2)

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 host character set.

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`.

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. By default, the source character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the source character set will be the host computer's default character set.

The execution character set is the set of legal characters that can appear in the execution environment. By default, the execution character set is the standard ASCII character set.

However, if you use the command line option `--enable_multibytes`, the execution character set will be the host computer's default character set. The IAR DLIB Library needs a multibyte character scanner to support a multibyte execution character set. See *Locale*, page 129.

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)

By default, the C locale is used. If the `--enable_multibytes` compiler option is used, the default host locale is used instead.

Locale used for wide string literals (6.4.5)

By default, the C locale is used. If the `--enable_multibytes` compiler option is used, the default host locale is used instead.

Source characters as executive characters (6.4.5)

All source characters can be represented as executive characters.

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 294.

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.

J.3.6 FLOATING POINT

Accuracy of floating-point operations (5.2.4.2.2)

The accuracy of floating-point operations 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 integral 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 `OFF`.

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 301.

ptrdiff_t (6.5.6)

For information about `ptrdiff_t`, see *ptrdiff_t*, page 301.

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 76.

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 296.

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 249.

Bitfields straddling a storage-unit boundary (6.7.2.1)

A bitfield is always placed in one—and one only—storage unit, which means that the bitfield cannot 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 296.

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 293.

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 303.

J.3.11 PREPROCESSING DIRECTIVES

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 351.

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 242.

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 227.

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 227.

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.

Universal character names (6.10.3.2)

Universal character names (UCN) are not supported.

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.

```
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
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 107.

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)

For information about the floating-point status flags, see *fenv.h*, page 365.

Feraiseexcept raising floating-point exception (7.6.2.3)

For information about the `feraiseexcept` function raising floating-point exceptions, see *Floating-point environment*, page 298.

Strings passed to the setlocale function (7.11.1.1)

For information about strings passed to the `setlocale` function, see *Locale*, page 129.

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 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`.

signal() (7.14.1.1)

The signal part of the library is not supported.

Note: Low-level interface functions exist in the library, but will not perform anything. Use the template source code to implement application-specific signal handling. See *Signal and raise*, page 133.

NULL macro (7.17)

The `NULL` macro is defined to `0`.

Terminating newline character (7.19.2)

`stdout` stream functions recognize either `newline` or `end of file (EOF)` as the terminating character for a line.

Space characters before a newline character (7.19.2)

Space characters written to a stream immediately before a newline character are preserved.

Null characters appended to data written to binary streams (7.19.2)

No null characters are appended to data written to binary streams.

File position in append mode (7.19.3)

The file position is initially placed at the beginning of the file when it is opened in append-mode.

Truncation of files (7.19.3)

Whether a write operation on a text stream causes the associated file to be truncated beyond that point, depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 129.

File buffering (7.19.3)

An open file can be either block-buffered, line-buffered, or unbuffered.

A zero-length file (7.19.3)

Whether a zero-length file exists depends on the application-specific implementation of the low-level file routines.

Legal file names (7.19.3)

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 (7.19.3)

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 (7.19.3)

The encoding of multibyte characters in a file depends on the application-specific implementation of the low-level file routines.

remove() (7.19.4.1)

The effect of a remove operation on an open file depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 129.

rename() (7.19.4.2)

The effect of renaming a file to an already existing filename depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 129.

Removal of open temporary files (7.19.4.3)

Whether an open temporary file is removed depends on the application-specific implementation of the low-level file routines.

Mode changing (7.19.5.4)

`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 (7.19.6.1, 7.24.2.1)

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() (7.19.6.1, 7.24.2.1)

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 (7.19.6.2, 7.24.2.1)

A - (dash) character is always treated as a range symbol.

%p in scanf (7.19.6.2, 7.24.2.2)

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 (7.19.9.1, 7.19.9.3, 7.19.9.4)

On file position errors, the functions `fgetpos`, `ftell`, and `fsetpos` store `EFPOS` in `errno`.

An n-char-sequence after nan (7.20.1.3, 7.24.4.1.1)

An n-char-sequence after a NaN is read and ignored.

errno value at underflow (7.20.1.3, 7.24.4.1.1)

`errno` is set to `ERANGE` if an underflow is encountered.

Zero-sized heap objects (7.20.3)

A request for a zero-sized heap object will return a valid pointer and not a null pointer.

Behavior of abort and exit (7.20.4.1, 7.20.4.4)

A call to `abort()` or `_Exit()` will not flush stream buffers, not close open streams, and not remove temporary files.

Termination status (7.20.4.1, 7.20.4.3, 7.20.4.4)

The termination status will be propagated to `__exit()` as a parameter. `exit()` and `_Exit()` use the input parameter, whereas abort uses `EXIT_FAILURE`.

The system function return value (7.20.4.6)

The `system` function is not supported.

The time zone (7.23.1)

The local time zone and daylight savings time must be defined by the application. For more information, see *Time*, page 133.

Range and precision of time (7.23)

For information about range and precision, see *time.h*, page 366. The application must supply the actual implementation for the functions `time` and `clock`. See *Time*, page 133.

clock() (7.23.2.1)

The application must supply an implementation of the `clock` function. See *Time*, page 133.

%Z replacement string (7.23.3.5, 7.24.5.1)

By default, ":" is used as a replacement for `%Z`. Your application should implement the time zone handling. See *Time*, page 133.

Math functions rounding mode (F.9)

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, 7.18.2, 7.18.3)

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 293.

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).

The number, order, and encoding of bytes (6.2.6.1)

See *Data representation*, page 293.

The value of the result of the sizeof operator (6.5.3.4)

See *Data representation*, page 293.

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. If the compiler option `--enable_multibytes` is used, the host multibyte characters are accepted in comments and string literals as well.

The meaning of the additional character set (5.2.1.2)

Any multibyte characters in the extended source character set is translated verbatim into the extended execution 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)

Using the compiler option `--enable_multibytes` enables the use of the host's default multibyte characters as extended source characters.

Direction of successive printing characters (5.2.2)

The application defines the characteristics of a display device.

The decimal point character (7.1.1)

The default decimal-point character is a '`.`'. You can redefine it by defining the library configuration symbol `_LOCALE_DECIMAL_POINT`.

Printing characters (7.4, 7.25.2)

The set of printing characters is determined by the chosen locale.

Control characters (7.4, 7.25.2)

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, 7.25.2.1.2, 7.25.5.1.3, 7.25.2.1.7, 7.25.2.1.9, 7.25.2.1.10, 7.25.2.1.11)

The sets of characters tested are determined by the chosen locale.

The native environment (7.1.1.1)

The native environment is the same as the "C" locale.

Subject sequences for numeric conversion functions (7.20.1, 7.24.4.1)

There are no additional subject sequences that can be accepted by the numeric conversion functions.

The collation of the execution character set (7.21.4.3, 7.24.4.4.2)

The collation of the execution character set is determined by the chosen locale.

Message returned by strerror (7.21.6.2)

The messages returned by the `strerror` function 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 54: Message returned by strerror()—IAR DLIB library

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 443. For a short overview of the differences between Standard C and C89, see *C language overview*, page 169.

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 (5.1.2.2.1)

The function called at program startup is called `main`. No prototype was declared for `main`, and the only definition supported for `main` is:

```
int main(void)
```

To change this behavior for the IAR DLIB runtime environment, see *Customizing system initialization*, page 123.

Interactive devices (5.1.2.3)

The streams `stdin` and `stdout` are treated as interactive devices.

IDENTIFIERS

Significant characters without external linkage (6.1.2)

The number of significant initial characters in an identifier without external linkage is 200.

Significant characters with external linkage (6.1.2)

The number of significant initial characters in an identifier with external linkage is 200.

Case distinctions are significant (6.1.2)

Identifiers with external linkage are treated as case-sensitive.

CHARACTERS

Source and execution character sets (5.2.1)

The source character set is the set of legal characters that can appear in source files. The default source character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the source character set will be the host computer's default character set.

The execution character set is the set of legal characters that can appear in the execution environment. The default execution character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the execution character set will be the host computer's default character set. The IAR DLIB Library needs a multibyte character scanner to support a multibyte execution character set.

See *Locale*, page 129.

Bits per character in execution character set (5.2.4.2.1)

The number of bits in a character is represented by the manifest constant `CHAR_BIT`. The standard include file `limits.h` defines `CHAR_BIT` as 8.

Mapping of characters (6.1.3.4)

The mapping of members of the source character set (in character and string literals) to members of the execution character set is made in a one-to-one way. In other words, the same representation value is used for each member in the character sets except for the escape sequences listed in the ISO standard.

Unrepresented character constants (6.1.3.4)

The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or in the extended character set for a wide character constant generates a diagnostic message, and will be truncated to fit the execution character set.

Character constant with more than one character (6.1.3.4)

An integer character constant that contains more than one character will be treated as an integer constant. The value will be calculated by treating the leftmost character as the most significant character, and the rightmost character as the least significant character, in an integer constant. A diagnostic message will be issued if the value cannot be represented in an integer constant.

A wide character constant that contains more than one multibyte character generates a diagnostic message.

Converting multibyte characters (6.1.3.4)

The only locale supported—that is, the only locale supplied with the IAR C/C++ Compiler—is the 'C' locale. If you use the command line option `--enable_multibytes`, the IAR DLIB Library will support multibyte characters if you add a locale with multibyte support or a multibyte character scanner to the library.

See *Locale*, page 129.

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 294, for information about the ranges for the different integer types.

Demotion of integers (6.2.1.2)

Converting an integer to a shorter signed integer is made by truncation. If the value cannot be represented when converting an unsigned integer to a signed integer of equal length, the bit-pattern remains the same. In other words, a large enough value will be converted into a negative value.

Signed bitwise operations (6.3)

Bitwise operations on signed integers work the same way as bitwise operations on unsigned integers; in other words, the sign-bit will be treated as any other bit.

Sign of the remainder on integer division (6.3.5)

The sign of the remainder on integer division is the same as the sign of the dividend.

Negative valued signed right shifts (6.3.7)

The result of a right-shift of a negative-valued signed integral type preserves the sign-bit. For example, shifting `0xFF00` down one step yields `0xFF80`.

FLOATING POINT

Representation of floating-point values (6.1.2.5)

The representation and sets of the various floating-point numbers adheres to IEEE 854–1987. A typical floating-point number is built up of a sign-bit (`s`), a biased exponent (`e`), and a mantissa (`m`).

See *Basic data types—floating-point types*, page 298, 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 301, for information about `size_t`.

Conversion from/to pointers (6.3.4)

See *Casting*, page 301, for information about casting of data pointers and function pointers.

ptrdiff_t (6.3.6, 7.1.1)

See *ptrdiff_t*, page 301, 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 294, for information about the alignment requirement for data objects.

Sign of 'plain' bitfields (6.5.2.1)

A 'plain' `int` bitfield is treated as a signed `int` bitfield. All integer types are allowed as bitfields.

Allocation order of bitfields within a unit (6.5.2.1)

Bitfields are allocated within an integer from least-significant to most-significant bit.

Can bitfields straddle a storage-unit boundary (6.5.2.1)

Bitfields cannot straddle a storage-unit boundary for the chosen bitfield integer type.

Integer type chosen to represent enumeration types (6.5.2.2)

The chosen integer type for a specific enumeration type depends on the enumeration constants defined for the enumeration type. The chosen integer type is the smallest possible.

QUALIFIERS

Access to volatile objects (6.5.3)

Any reference to an object with volatile qualified type is an access.

DECLARATORS

Maximum numbers of declarators (6.5.4)

The number of declarators is not limited. The number is limited only by the available memory.

STATEMENTS

Maximum number of case statements (6.6.4.2)

The number of case statements (case values) in a switch statement is not limited. The number is limited only by the available memory.

PREPROCESSING DIRECTIVES

Character constants and conditional inclusion (6.8.1)

The character set used in the preprocessor directives is the same as the execution character set. The preprocessor recognizes negative character values if a 'plain' character is treated as a `signed` character.

Including bracketed filenames (6.8.2)

For file specifications enclosed in angle brackets, the preprocessor does not search directories of the parent files. A parent file is the file that contains the `#include` directive. Instead, it begins by searching for the file in the directories specified on the compiler command line.

Including quoted filenames (6.8.2)

For file specifications enclosed in quotes, the preprocessor directory search begins with the directories of the parent file, then proceeds through the directories of any grandparent files. Thus, searching begins relative to the directory containing the source file currently being processed. If there is no grandparent file and the file 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`

`warnings`

Default __DATE__ and __TIME__ (6.8.8)

The definitions for `__TIME__` and `__DATE__` are always available.

IAR DLIB LIBRARY FUNCTIONS

The information in this section is valid only if the runtime library configuration you have chosen supports file descriptors. See the chapter *The DLIB runtime environment* for more information about runtime library configurations.

NULL macro (7.1.6)

The `NULL` macro is defined to `0`.

Diagnostic printed by the assert function (7.2)

The `assert()` function prints:

`filename``:`*`linenr expression`* `-- assertion failed`

when the parameter evaluates to zero.

Domain errors (7.5.1)

`NaN` (Not a Number) will be returned by the mathematic functions on domain errors.

Underflow of floating-point values sets errno to ERANGE (7.5.1)

The mathematics functions set the integer expression `errno` to `ERANGE` (a macro in `errno.h`) on underflow range errors.

fmod() functionality (7.5.6.4)

If the second argument to `fmod()` is zero, the function returns `NaN`; `errno` is set to `EDOM`.

signal() (7.7.1.1)

The signal part of the library is not supported.

Note: Low-level interface functions exist in the library, but will not perform anything. Use the template source code to implement application-specific signal handling. See *Signal and raise*, page 133.

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 *File input and output*, page 129.

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 *File input and output*, page 129.

rename() (7.9.4.2)

The effect of renaming a file to an already existing filename depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 129.

%p in printf() (7.9.6.1)

The argument to a `%p` conversion specifier, print pointer, to `printf()` is treated as having the type `void *`. The value will be printed as a hexadecimal number, similar to using the `%x` conversion specifier.

%p in scanf() (7.9.6.2)

The `%p` conversion specifier, scan pointer, to `scanf()` reads a hexadecimal number and converts it into a value with the type `void *`.

Reading ranges in scanf() (7.9.6.2)

A - (dash) character is always treated as a range symbol.

File position errors (7.9.9.1, 7.9.9.4)

On file position errors, the functions `fgetpos` and `ftell` store `EFPOS` in `errno`.

Message generated by perror() (7.9.10.4)

The generated message is:

usersuppliedprefix`:`*errormessage*

Allocating zero bytes of memory (7.10.3)

The `calloc()`, `malloc()`, and `realloc()` functions accept zero as an argument. Memory will be allocated, a valid pointer to that memory is returned, and the memory block can be modified later by `realloc`.

Behavior of abort() (7.10.4.1)

The `abort()` function does not flush stream buffers, and it does not handle files, because this is an unsupported feature.

Behavior of exit() (7.10.4.3)

The argument passed to the `exit` function will be the return value returned by the `main` function to `cstartup`.

Environment (7.10.4.4)

The set of available environment names and the method for altering the environment list is described in *Environment interaction*, page 132.

system() (7.10.4.5)

How the command processor works depends on how you have implemented the `system` function. See *Environment interaction*, page 132.

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 55: Message returned by strerror()—IAR DLIB library

The time zone (7.12.1)

The local time zone and daylight savings time implementation is described in *Time*, page 133.

clock() (7.12.2.1)

From where the system clock starts counting depends on how you have implemented the `clock` function. See *Time*, page 133.

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Symbols

Numerics