IAR Embedded Workbench

IAR C/C++ Compiler
User Guide

for Microchip Technology's
AVR Microcontroller Family

◉IAR
SYSTEMS

Brief contents

Contents

Pragma directives 319

Tables

Preface

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

Who should read this guide

Read this guide if you plan to develop an application using the C or C++ language for the AVR microcontroller and need detailed reference information on how to use the compiler.

REQUIRED KNOWLEDGE

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

● The architecture and instruction set of the Microchip AVR microcontroller (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 25.

How to use this guide

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

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

- *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 AVR 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 overview* describes the linking process using the IAR XLINK Linker and the related concepts.
- *Linking your application* lists aspects that you must consider when linking your application, including using XLINK 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.
- *The CLIB runtime environment* gives an overview of the CLIB runtime libraries and how to customize them. The chapter also describes system initialization and introduces the file `cstartup`.
- *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 interacts with its environment—the invocation syntax, methods for passing options to the compiler, environment variables, the include file search procedure, and the different types of compiler output. The chapter also describes how the compiler's 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.

● *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 AVR-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 AVR-specific low-level features.

● *The preprocessor* gives a brief overview of the preprocessor, including reference information about the different preprocessor directives, symbols, and other related information.

● *C/C++ standard library functions* gives an introduction to the C or C++ library functions, and summarizes the header files.

● *Segment reference* gives reference information about the compiler's use of segments.

● *The stack usage control file* describes the syntax and semantics of stack usage control files.

● *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*.

● Using the IDE for project management and building, is available in the *IDE Project Management and Building Guide for AVR.*

● Using the IAR C-SPY® Debugger, is available in the *C-SPY® Debugging Guide for AVR.*

● Programming for the IAR C/C++ Compiler for AVR, is available in the *IAR C/C++ Compiler User Guide for AVR.*

● Using the IAR XLINK Linker, the IAR XAR Library Builder, and the IAR XLIB Librarian, is available in the *IAR Linker and Library Tools Reference Guide*.

● Programming for the IAR Assembler for AVR, is available in the *IAR Assembler User Guide for AVR.*

● Performing a static analysis using C-STAT and the required checks, is available in the *C-STAT Static Analysis Guide*.

● 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*.

● Porting application code and projects created with a previous version of the IAR Embedded Workbench for AVR, is available in the *IAR Embedded Workbench® Migration Guide*.

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

THE ONLINE HELP SYSTEM

The context-sensitive online help contains:

● Information about 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. Note that if you select a function name in the editor window and press F1 while

using the CLIB C standard library, you will get reference information for the DLIB C/EC++ standard library.

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.

● Labrosse, Jean J. *Embedded Systems Building Blocks: Complete and Ready-To-Use Modules in C.* R&D Books.

● Mann, Bernhard. *C für Mikrocontroller.* Franzis-Verlag. [Written in German.]

● Meyers, Scott. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs.* Addison-Wesley.

● Meyers, Scott. *More Effective C++.* Addison-Wesley.

● Meyers, Scott. *Effective STL.* Addison-Wesley.

● Sutter, Herb. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions.* Addison-Wesley.

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

WEB SITES

Recommended web sites:

● The Microchip Technology web site, **www.microchip.com**, that contains information and news about the Microchip AVR 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**.

● The C++ programming language web site, **isocpp.org**.

This web site also has a list of recommended books about C++ programming.

● The C and C++ reference web site, **en.cppreference.com**.

Document conventions

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

When referring to a directory in your product installation, for example `avr\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench N.n\avr\doc`, where the initial digit of the version number reflects the initial digit of the version number of the IAR Embedded Workbench shared components.

TYPOGRAPHIC CONVENTIONS

The IAR Systems documentation set uses the following typographic conventions:

Style	Used for
`computer`	• Source code examples and file paths. • Text on the command line. • Binary, hexadecimal, and octal numbers.
parameter	A placeholder for an actual value used as a parameter, for example `filename.h` where `filename` represents the name of the file.
[option**]**	An optional part of a 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.
	Identifies instructions specific to the IAR Embedded Workbench® IDE interface.
	Identifies instructions specific to the command line interface.
	Identifies helpful tips and programming hints.

Table 1: Typographic conventions used in this guide

Style	Used for
⚠	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 AVR	IAR Embedded Workbench®
IAR Embedded Workbench® IDE for AVR	the IDE
IAR C-SPY® Debugger for AVR	C-SPY, the debugger
IAR C-SPY® Simulator	the simulator
IAR C/C++ Compiler™ for AVR	the compiler
IAR Assembler™ for AVR	the assembler
IAR XLINK Linker™	XLINK, the linker
IAR XAR Library Builder™	the library builder
IAR XLIB Librarian™	the librarian
IAR DLIB Runtime Environment™	the DLIB runtime environment
IAR CLIB Runtime Environment™	the CLIB runtime environment

Table 2: Naming conventions used in this guide

Part 1. Using the compiler

This part of the *IAR C/C++ Compiler User Guide for AVR* includes these chapters:

- Introduction to the IAR build tools

- Developing embedded applications

- Data storage

- Functions

- Linking overview

- Linking your application

- The DLIB runtime environment

- The CLIB runtime environment

- Assembler language interface

- Using C

- Using C++

- Application-related considerations

- Efficient coding for embedded applications.
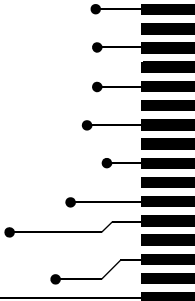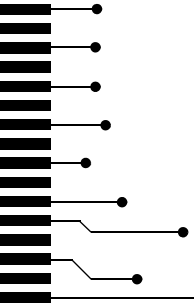
Introduction to the IAR build tools

- 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 AVR-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 for AVR*.

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 AVR 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 AVR-specific facilities.

IAR ASSEMBLER

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

THE IAR XLINK LINKER

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

To handle libraries, the library tools XAR and XLIB are included.

EXTERNAL TOOLS

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

IAR language overview

The IAR C/C++ Compiler for AVR 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 AVR*.

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 AVR DEVICES

The IAR C/C++ Compiler for AVR supports all devices based on the standard Microchip Technology AVR microcontroller.

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 many of the devices that are available at the time of the product release. You can find these files in the `avr\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 `avr\config` directory and the `avr\src\template` directory contain ready-made linker configuration files for all supported devices. The files have the filename extension `xcl` 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 74 as well as the *IAR Linker and Library Tools Reference Guide*.

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 `avr\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 AVR*.

EXAMPLES FOR GETTING STARTED

The `avr\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 AVR 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 250 for additional information.

For more information about the extended keywords, see the chapter *Extended keywords*. See also, *Data storage*, page 53 and *Functions*, page 63.

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 149.

Developing embedded applications

- Developing embedded software using IAR build tools

- The build process—an overview

- Application execution—an overview

- Building applications—an overview

- Basic project configuration

Developing embedded software using IAR build tools

Typically, embedded software written for a dedicated microcontroller is designed as an endless loop waiting for some external events to happen. The software is located in ROM and executes on reset. You must consider several hardware and software factors when you write this kind of software. To your help, you have compiler options, extended keywords, pragma directives, etc.

CPU FEATURES AND CONSTRAINTS

When developing software for the AVR microcontroller, you must consider some CPU features and constraints. For example, the instruction set, processor configuration, and memory model.

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

MAPPING OF MEMORY

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

As an embedded software developer, you must understand the features of the different types of memory. For example, on-chip RAM is often faster than other types of memories, and variables that are accessed often would in time-critical applications benefit from being placed here. Conversely, some configuration data might be 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 210. 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 74.

COMMUNICATION WITH PERIPHERAL UNITS

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

Standard peripheral units are defined in device-specific I/O header files with the filename extension h. See *Device support*, page 35. For an example, see *Accessing special function registers*, page 222.

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 64.

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 44.

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.

See also *Managing a multithreaded environment*, page 135.

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 IAR UBROF format.

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

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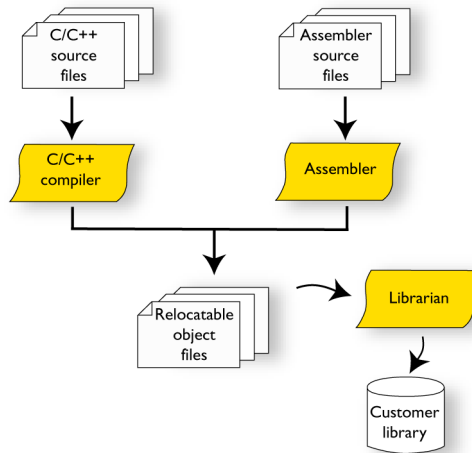
After the translation, you can choose to pack any number of modules into an archive, or in other words, a library. The important reason you should use libraries is that each module in a library is conditionally linked in the application, or in other words, is only included in the application if the module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

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*.

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

● Several object files and possibly certain libraries

● The standard library containing the runtime environment and the standard language functions

● 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

● Information about the output format.

The IAR XLINK Linker produces output according to your specifications. Choose the output format that suits your purpose. You might want to load the output to a

debugger—which means that you need output with debug information. Alternatively, you might want to load output to a flash loader or a PROM programmer—in which case you need output without debug information, such as Intel hex or Motorola S–records. The option -F can be used for specifying the output format.

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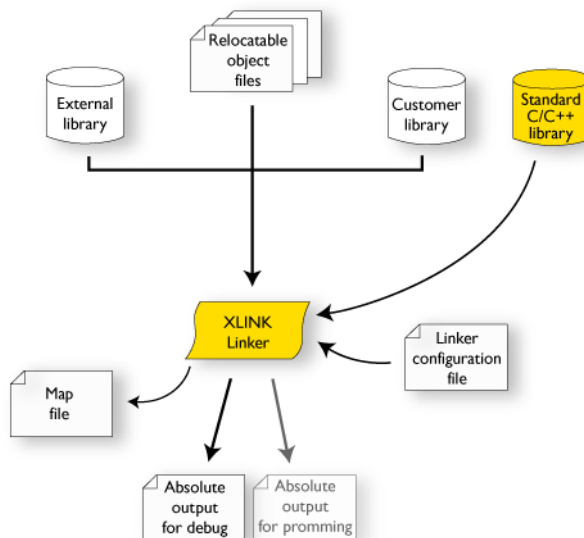
Note: The Standard C/C++ library contains support routines for the compiler, and the implementation of the C/C++ standard library functions.

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 *IAR Linker and Library Tools Reference Guide*.

AFTER LINKING

The IAR XLINK Linker produces an absolute object file in the output format you specify. 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 UBROF.

● Programming to a flash/PROM using a flash/PROM programmer.

This illustration shows the possible uses of the absolute output files:
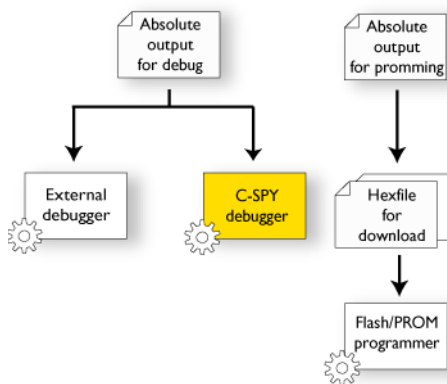
Application execution—an overview

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.s90` 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.

⏐ When an application is started, the system startup code first performs hardware initialization, such as initialization of the stack pointer to point at the end of the predefined stack area
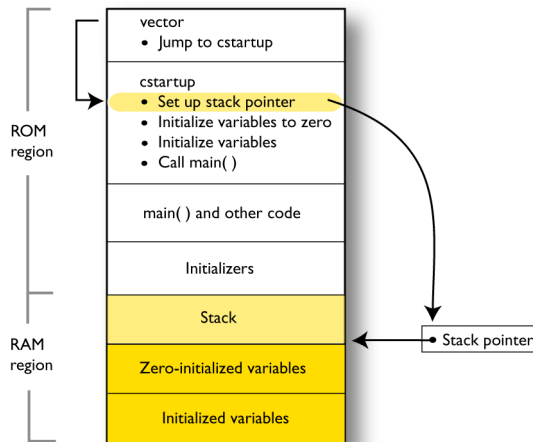
2 Then, memories that should be zero-initialized are cleared, in other words, filled with zeros.
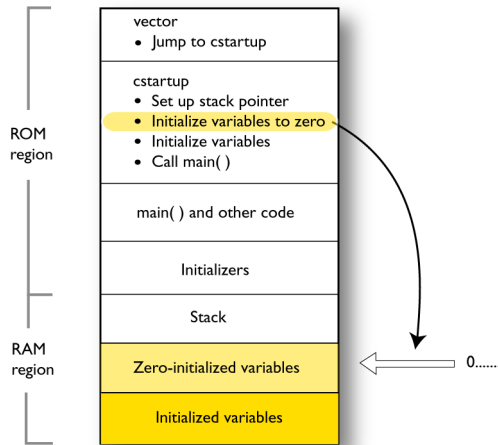
Typically, this is data referred to as *zero-initialized data*; variables declared as, for example, `int i = 0;`

3 For *initialized data*, data declared, for example, like `int i = 6;` the initializers are copied from ROM to RAM.
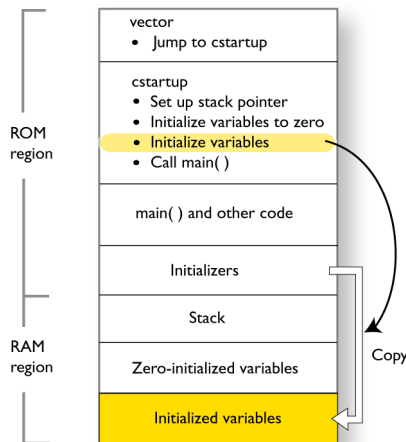
4 Finally, the `main` function is called.
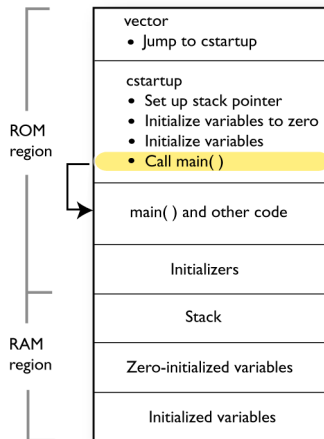
For more information about each stage, see *System startup and termination*, page 118. For more information about initialization of data, see *Initialization at system startup*, page 75.

Note: The AVR microcontroller is based on the Harvard architecture—thus code and data have separate memory spaces and require different access mechanisms. For more information, see the chapter *Data storage*.

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 121.

Building applications—an overview

In the command line interface, this line compiles the source file `myfile.c` into the object file `myfile.r90` using the default settings:

```
iccavr myfile.c
```

You must also specify some critical options, see *Basic project configuration*, page 48.

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

```
xlink myfile.r90 myfile2.r90 -o a.d90 -f my_configfile.xcl -r
```

In this example, `myfile.r90` and `myfile2.r90` are object files, and `my_configfile.xcl` is the linker configuration file. The option `-o` specifies the name of the output file. The option `-r` is used for specifying the output format UBROF, which can be used for debugging in C-SPY®.

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

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

Basic project configuration

This section gives an overview of the basic settings for the project setup that are needed to make the compiler and linker generate the best code for the AVR 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:

- Processor configuration
- Memory model

- Size of `double` floating-point type
- Optimization settings
- Runtime environment, see *Setting up the runtime environment*, page 103
- Customizing the XLINK configuration, see the chapter *Linking your application*.

In addition to these settings, 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 chapter *Compiler options* and the *IDE Project Management and Building Guide for AVR*, respectively.

PROCESSOR CONFIGURATION

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

The --cpu option versus the -v option

There are two processor options that can be used for configuring the processor support:

`--cpu=`*device* and `-v`*n*

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

Both options set up default behavior—implicit assumptions—but note that the `--cpu` option is more precise because it contains more information about the intended target than the more generic `-v` option. The `--cpu` option knows, for example, how much flash memory is available in the given target.

The `--cpu=`*device* option implicitly sets up all internal compiler settings needed to generate code for the processor variant you are using. These options are implicitly controlled when you use the `--cpu` option: `--eecr_address`, `--eeprom_size`, `--enhanced_core`, `--spmcr_address`, `-v` and `--64k_flash`.

Because these options are automatically set when you use the `--cpu` option, you cannot set them explicitly. For information about implicit assumptions when using the `-v` option, see *Summary of processor configuration for -v*, page 50. For more information about the generated code, see *-v*, page 275.

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

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

Summary of processor configuration for -v

This table summarizes the memory characteristics for each –v option:

Generic processor option	Available memory models	Function memory attribute	Max addressable data	Max module and/or program size
–v0 (default)	Tiny	__nearfunc	<= 256 bytes	<= 8 Kbytes
–v1	Tiny, Small	__nearfunc	<= 64 Kbytes	<= 8 Kbytes
–v2	Tiny	__nearfunc	<= 256 bytes	<= 128 Kbytes
–v3	Tiny, Small	__nearfunc	<= 64 Kbytes	<= 128 Kbytes
–v4	Small, Large, Huge	__nearfunc	<= 16 Mbytes	<= 128 Kbytes
–v5	Tiny, Small	__farfunc*	<= 64 Kbytes	<= 8 Mbytes
–v6	Small, Large, Huge	__farfunc*	<= 16 Mbytes	<= 8 Mbytes

Table 3: Summary of processor configuration

Note:

- *) When using the –v5 or the –v6 option, it is possible, for individual functions, to override the __farfunc attribute and instead use the __nearfunc attribute
- Pointers with function memory attributes have restrictions in implicit and explicit casts between pointers and between pointers and integer types. For details about the restrictions, see *Casting*, page 289.
- –v2: There are currently no devices that match this processor option, which has been added to support future devices
- All implicit assumptions for a given –v option are also true for corresponding --cpu options.

It is important to be aware of the fact that the –v option does not reflect the amount of used data, but the maximum amount of addressable data. This means that, for example, if you are using a microcontroller with 16 Mbytes addressable data, but you are not using more than 256 bytes or 64 Kbytes of data, you must still use either the –v4 or the –v6 option for 16 Mbytes data.

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 --64bit_doubles, you can make the compiler use 64-bit doubles. 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, and precision reduction. It also performs loop optimizations, such as induction variable elimination.

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

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

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

Data storage

- Introduction

- Memory types

- Memory models

- Storage of auto variables and parameters

- Dynamic memory on the heap

Introduction

The AVR microcontroller is based on the Harvard architecture—thus code and data have separate memory spaces and require different access mechanisms. Code and different types of data are located in memory spaces as follows:

- The internal flash space, which is used for code, `__flash` declared objects, and initializers

- The data space, which can consist of external ROM, used for constants, and RAM areas used for the stack, for registers, and for variables

- The EEPROM space, which is used for variables.

DIFFERENT WAYS TO STORE DATA

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

- Auto variables

  All variables that are local to a function, except those declared static, are stored either in registers or on the stack. These variables can be used as long as the function executes. When the function returns to its caller, the memory space is no longer valid. For more information, see *Storage of auto variables and parameters*, page 60.

- 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 *Memory types*, page 54.

- 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 61.

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 the near memory access method is called memory of near type, or simply near memory.

By selecting a *memory model*, you have selected a default memory type that your application will use. However, it is possible to specify—for individual variables or pointers—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.

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 attribute	Pointer size	Memory space	Address range	Max object size
`__tiny`	1 byte	Data	`0x0-0xFF`	127 bytes
`__near`	2 bytes	Data	`0x0-0xFFFF`	32 Kbytes-1
`__far`	3 bytes	Data	`0x0-0xFFFFFF` (16-bit pointer arithmetics)	32 Kbytes-1
`__huge`	3 bytes	Data	`0x0-0xFFFFFF`	16 Mbytes-1
`__tinyflash`	1 byte	Code	`0x0-0xFF`	127 bytes
`__flash`	2 bytes	Code	`0x0-0xFFFF`	32 Kbytes-1
`__farflash`	3 bytes	Code	`0x0-0xFFFFFF` (16-bit pointer arithmetics)	32 Kbytes-1
`__hugeflash`	3 bytes	Code	`0x0-0xFFFFFF`	8 Mbytes-1
`__eeprom`	1 bytes	EEPROM	`0x0-0xFF`	255 bytes
`__eeprom`	2 bytes	EEPROM	`0x0-0xFFFF`	64 Kbytes-1
`__io`	N/A	I/O space	`0x0-0x3F`	4 bytes
`__io`	N/A	Data	`0x60-0xFF`	4 bytes
`__ext_io`	N/A	Data	`0x100-0xFFFF`	4 bytes
`__generic`	2 bytes 3 bytes	Data or Code	The most significant bit (MSB) determines whether this pointer points to code space (1) or data space (0). The small generic pointer is generated for the processor options `-v0` and `-v1`.	32 Kbytes-1 8 Mbytes-1
`__regvar`	N/A	Data	`0x4-0x0F`	4 bytes

Table 4: Memory types and their corresponding memory attributes

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 250 for additional information.

For more information about each keyword, see *Descriptions of extended keywords*, page 300.

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 295.

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

```
__eeprom int i, j;
int __eeprom 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 __far Byte;
typedef Byte *BytePtr;
Byte aByte;
BytePtr aBytePointer;

/* Defines directly */
__far char aByte;
char __far *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 far memory is declared by:

```
int __far * 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 near memory. Like `MyPtr`, `MyPtr2` points to a character in far memory.

```
char __far * __near MyPtr2;
```

Whenever possible, pointers should be declared without memory attributes. For example, the functions in the standard library are all declared without explicit memory types.

Differences between pointer types

A pointer must contain information needed to specify a memory location of a certain memory type. This means that the pointer sizes are different for different memory types. In the IAR C/C++ Compiler for AVR, the size of the near and far pointers are 16 and 24 bits, respectively.

In the compiler, it is illegal to convert pointers between different types without explicit casts. For more information, see *Casting*, page 289.

For more information about pointers, see *Pointer types*, page 287.

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 eeprom memory.

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

__eeprom struct MyStruct gamma;
```

This declaration is incorrect:

```
struct MyStruct
{
  int mAlpha;
  __eeprom 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 far memory is declared. The function returns a pointer to an integer in eeprom 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.
`int __far MyB;`	A variable in far memory.
`__eeprom int MyC;`	A variable in eeprom memory.
`int * MyD;`	A pointer stored in default memory. The pointer points to an integer in default memory.
`int __far * MyE;`	A pointer stored in default memory. The pointer points to an integer in far memory.
`int __far * __eeprom MyF;`	A pointer stored in eeprom memory pointing to an integer stored in far memory.
`int __eeprom * MyFunction(` `    int __far *);`	A declaration of a function that takes a parameter which is a pointer to an integer stored in far memory. The function returns a pointer to an integer stored in eeprom 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 177.

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

All member functions except for constructors and destructors can be placed individually into a code memory in the same way as free functions.

For more information about C++ classes, see *Using IAR attributes with Classes*, page 177.

Memory models

Technically, the memory model specifies the default memory type attribute and the default data pointer attribute. This means that the memory model controls the following:

● The default placement of static and global variables, and constant literals

● Dynamically allocated data, for example data allocated with `malloc`, or, in C++, the operator `new`

● The default pointer type

● The placement of the runtime stack. For details about stack placement, see *CSTACK*, page 378.

For information about how to specify a memory type for individual objects, see *Using data memory attributes*, page 54.

SPECIFYING A MEMORY MODEL

Four memory models are implemented: Tiny, Small, Large, and Huge. These models are controlled by the `--memory_model` option. Each model has a default memory type and a default pointer size. The code size will also be reduced somewhat if the Tiny or Small memory model is used.

If you do not specify a memory model option, the compiler will use the Tiny memory model for all processor options, except for `-v4` and `-v6`, where the Small memory model will be used. For information about processor options, see *Summary of processor configuration for -v*, page 50.

Your project can only use one memory 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 and pointers by explicitly specifying a memory attribute, see *Using data memory attributes*, page 54.

This table summarizes the different memory models:

Memory model	Default memory attribute	Default data pointer	Max. stack range	Supported by processor option
Tiny	`__tiny`	`__tiny`	<= 256 bytes in the range `0-FF`	`-v0`, `-v1`, `-v2`, `-v3`, `-v5`
Small	`__near`	`__near`	<= 64 Kbytes in the range `0-FFFF`	`-v1`, `-v3`, `-v4`, `-v5`, `-v6`

Table 5: Memory model characteristics

Memory model	Default memory attribute	Default data pointer	Max. stack range	Supported by processor option
Large	__far	__far	<= 64 Kbytes in the range 0-FFFFFF, but never cross a 64-Kbyte boundary	-v4, -v6
Huge	__huge	__huge	<= 64 Kbytes in the range 0-FFFFFF, but never cross a 64-Kbyte boundary	-v4, -v6

Table 5: Memory model characteristics (Continued)

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

Use the --memory_model option to specify the memory model for your project; see --*memory_model, -m*, page 258.

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 193 and *Setting up stack memory*, page 91.

Advantages

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

It is possible for a function to call itself either directly or indirectly—a recursive function—and each invocation can store its own data on the stack.

Potential problems

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

```
int *MyFunction()
{
  int x;
  /* Do something here. */
  return &x; /* Incorrect */
}
```

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

Dynamic memory on the heap

Memory for objects allocated on the heap will live until the objects are explicitly released. This type of memory storage is 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`.

The compiler supports heaps in more than one memory types:

Memory type	Memory attribute	Section name	Used by default in memory model
tiny	`__tiny`	`TINY_HEAP`	Tiny
near	`__near`	`NEAR_HEAP`	Small
far	`__far`	`FAR_HEAP`	Large
huge	`__huge`	`HUGE_HEAP`	Huge

Table 6: Heaps supported in memory types

In DLIB, to use a specific heap, add the memory attribute in front of `malloc`, `free`, `alloc`, and `realloc`, for example `__near_malloc`. The default functions will use of the specific heap variants, depending on project settings such as data model.

For information about how to set up the size for heap memory, see *Setting up heap memory*, page 92.

POTENTIAL PROBLEMS

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

- Function storage

- 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:

- Control the storage of functions in memory
- 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 205. For information about the available intrinsic functions for accessing hardware operations, see the chapter *Intrinsic functions*.

Function storage

USING FUNCTION MEMORY ATTRIBUTES

It is possible to override the default placement for individual functions. Use the appropriate *function memory attribute* to specify this. These attributes are available:

Memory attribute	Address range	Pointer size	Used in processor option
__nearfunc	0-0x1FFFE (128 Kbytes)	16 bits	-v0, -v1, -v2, -v3, -v4
__farfunc	0-0x7FFFFE (8 Mbytes)	24 bits	-v5, -v6

Table 7: Function memory attributes

When using the -v5 or the -v6 option, it is possible, for individual functions, to override the `__farfunc` attribute and instead use the `__nearfunc` attribute. The default memory can be overridden by explicitly specifying a memory attribute in the function declaration or by using the #pragma type_attribute directive:

```
#pragma type_attribute=__nearfunc
void MyFunc(int i)
{
...
}
```

It is possible to call a `__nearfunc` function from a `__farfunc` function and vice versa. Only the size of the function pointer is affected.

It is possible to place functions into named segments using either the @ operator or the #pragma location directive. For more information, see *Controlling data and function placement in memory*, page 210.

Pointers with function memory attributes have restrictions in implicit and explicit casts between pointers and between pointers and integer types. For information about the restrictions, see *Casting*, page 289.

For syntax information and for more information about each attribute, see the chapter *Extended keywords*.

Primitives for interrupts, concurrency, and OS-related programming

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

- The extended keywords: `__interrupt`, `__nested`, `__task`, `__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 AVR 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 AVR 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 AVR microcontroller, the interrupt vector table always starts at the address `0x0` and is placed in the `INTVEC` segment. The interrupt vector is the offset into the interrupt vector table. The interrupt vector table contains pointers to interrupt routines, including the reset routine. The AT90S80515 device has 13 interrupt vectors and one reset vector. For this reason, you should specify 14 interrupt vectors, each of two bytes.

If a vector is specified in the definition of an interrupt function, the processor interrupt vector table is populated. It is also possible to define an interrupt function without a vector. This is useful if an application is capable of populating or changing the interrupt vector table at runtime.

Defining an interrupt function—an example

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

```
#pragma vector = 0x14
__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.

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 308.

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.

```c
/* 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();
}
```

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 214.

For more information about function inlining optimization, see *Function inlining*, page 217.

Linking overview

- Linking—an overview

- Segments and memory

- 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 XLINK 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 required parts of object libraries to produce an executable image containing machine code for the microcontroller you are using. XLINK can generate more than 30 industry-standard loader formats, in addition to the proprietary format UBROF which is used by the C-SPY debugger.

The linker will automatically load only those library modules that are actually needed by the application you are linking. Further, the linker eliminates segment parts that are not required. During linking, the linker performs a full C-level type checking across all modules.

The linker uses a *configuration file* where you can specify separate locations for code and data areas of your target system memory map.

The final output produced by the linker is an absolute, target-executable object file that can be downloaded to the microcontroller, to C-SPY, or to a compatible hardware debugging probe. Optionally, the output file can contain debug information depending on the output format you choose.

To handle libraries, the library tools XAR and XLIB can be used.

Segments and memory

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

WHAT IS A SEGMENT?

A *segment* is a container for pieces of data or code that should be mapped to a location in physical memory. Each segment consists of one or more *segment parts*. Normally, each function or variable with static storage duration is placed in its own segment part. A segment part is the smallest linkable unit, which allows the linker to include only those segment parts that are referred to. A segment can be placed either in RAM or in ROM. Segments that are placed in RAM generally do not have any content, they only occupy space.

Note: Here, ROM memory means all types of read-only memory, including flash memory.

The compiler uses several predefined segments for different purposes. Each segment is identified by a name that typically describes the contents of the segment, and has a *segment memory type* that denotes the type of content. In addition to the predefined segments, you can also define your own segments.

At compile time, the compiler assigns code and data to the various segments. The IAR XLINK Linker is responsible for placing the segments in the physical memory range, in accordance with the rules specified in the linker configuration file. Ready-made linker configuration files are provided, but, if necessary, they can be modified according to the requirements of your target system and application. It is important to remember that, from the linker's point of view, all segments are equal; they are simply named parts of memory.

Segment memory type

Each segment always has an associated segment memory type. In some cases, an individual segment has the same name as the segment memory type it belongs to, for example CODE. Make sure not to confuse the segment name with the segment memory type in those cases.

By default, the compiler uses these XLINK segment memory types:

Segment memory type	Used for
CODE	Executable code
XDATA	Data placed in EEPROM

Table 8: XLINK segment memory types

Segment memory type	Used for
DATA	Data placed in RAM

Table 8: XLINK segment memory types (Continued)

XLINK supports more segment memory types than the ones described above. However, they exist to support other types of microcontrollers.

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

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 make an application executable, the object files must be *linked*.

The IAR XLINK 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):

● Determines which modules to include in the application. Program modules are always included. Library modules are only included if they provide a definition for a global symbol that is referenced from an included module. If the object files containing library modules contain multiple definitions of variables or functions, only the first definition will be included. This means that the linking order of the object files is important.

● Determines which segment parts from the included modules to include in the application. Only those segments that are actually needed by the application are included. There are several ways to determine of which segment parts that are needed, for example, the `__root` object attribute, the `#pragma required` directive, and the `-g` linker option.

● Divides each segment that will be initialized by copying into two segments, one for the ROM part and one for the RAM part. The RAM part contains the label and the ROM part the actual bytes. The bytes are conceptually linked as residing in RAM.

● Determines where to place each segment according to the segment placement directives in the *linker configuration file*.

● Produces an absolute file that contains the executable image and any debug information. The contents of each needed segment in the relocatable input files is calculated using the relocation information supplied in its file and the addresses determined when placing segments. This process can result in one or more range errors if some of the requirements for a particular segment 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, produces a map file that lists the result of the segment placement, the address of each global symbol, and finally, a summary of memory usage for each module.

This illustration shows the linking process:
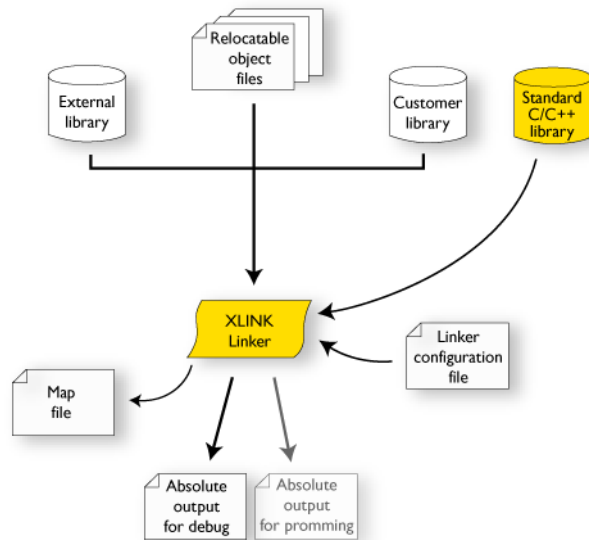
During the linking, XLINK might produce error messages and optionally a map file. In the map file you can see the result of the actual linking and is useful for understanding why an application was linked the way it was, for example, why a segment part was included. If a segment part is not included although you expect it to be, the reason is *always* that the segment part was not referenced to from an included part.

Note: To inspect the actual content of the object files, use XLIB. See the *IAR Linker and Library Tools Reference Guide*.

Placing code and data—the linker configuration file

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

In particular, the linker configuration file specifies:

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

The file consists of a sequence of linker commands. This means that the linking process will be governed by all commands in sequence.

THE CONTENTS OF THE LINKER CONFIGURATION FILE

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

- The CPU used:

  `-ca90`

  This specifies your target microcontroller.

- Definitions of constants used in the file. These are defined using the XLINK option `-D`. Symbols defined using `-D` can also be accessed from your application.

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

In the linker configuration file, numbers are generally specified in hexadecimal format. However, neither the prefix `0x` nor the suffix `h` is necessarily used.

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

For more information about the linker configuration file and how to customize it, see *Linking considerations*, page 87.

See also the *IAR Linker and Library Tools Reference Guide*.

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. Static variables can be divided into these categories:

- Variables that are initialized to a non-zero value
- Variables that are initialized to zero

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

STATIC DATA MEMORY SEGMENTS

The compiler generates a specific type of segment for each type of variable initialization.

The names of the segments consist of two parts—the *segment group name* and a *suffix*—for instance, NEAR_Z. There is a segment group for each memory type, where each segment in the group holds different categories of declared data. The names of the segment groups are derived from the memory type and the corresponding keyword, for example NEAR and `__near`.

Some of the declared data is placed in non-volatile memory, for example ROM/flash, and some of the data is placed in RAM. For this reason, it is also important to know the XLINK segment memory type of each segment. For more information about segment memory types, see *Segment memory type*, page 72.

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

Categories of declared data	Source	Segment type	Segment name*	Segment content
Zero-initialized data	`int i;`	Read/write data, zero-init	*MEMATTR_Z*	None
Zero-initialized data	`int i = 0;`	Read/write data, zero-init	*MEMATTR_Z*	None
Initialized data (non-zero)	`int i = 6;`	Read/write data	*MEMATTR_I*	None
Initialized data (non-zero)	`int i = 6;`	Read-only data	*MEMATTR_ID*	Initializer data for *MEMATTR_I*
Constants	`const int i = 6;`	Read-only data	*MEMATTR_C*	The constant
Non-initialized data	`__no_init int i;`	Read/write data	*MEMATTR_N*	None
Data placed in flash memory	`__memattr int i = 6;`	Read-only data	*MEMATTR_F*	The constant

Table 9: segments holding initialized data

* The actual segment group name—*MEMATTR*—depends on the memory where the variable is placed. See *Memory types*, page 54.

For more information about each segment, see the chapter *Segment reference*.

THE INITIALIZATION PROCESS

Initialization of data is handled by the system startup code. If you add more segments, you must update the system startup code accordingly.

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

● Segments that should be zero-initialized should only be placed in RAM.

● Segments that should be initialized, except for zero-initialized segments:

  The system startup code initializes the non-zero variables by copying a block of ROM to the location of the variables in RAM. This means that the data in the ROM segment with the suffix ID is copied to the corresponding I segment.

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

  ● The other segment is divided in *exactly* the same way

  ● It is legal to read and write the memory that represents the gaps in the sequence.

● Segments that contain constants do not need to be initialized; they should only be placed in flash/ROM

● Segments holding __no_init declared variables should not be initialized.

● Finally, global C++ object constructors are called.

For more information about and examples of how to configure the initialization, see *Linking considerations*, page 87.

Initialization of local aggregates at function invocation

Initialized aggregate auto variables—struct, union, and array variables local to a function—have the initial values in blocks of memory. As an auto variable is allocated either in registers or on the stack, the initialization has to take place every time the function is called. Assume the following example:

```
void f()
{
  struct block b = { 3, 4, 2, 3, 6634, 234 };
  ...
}
```

The initializers are copied to the b variable allocated on the stack each time the function is entered.

The initializers can either come from the code memory space (flash) or from the data memory space (optional external ROM). By default, the initializers are located in segments with the suffix _C and these segments are copied from external ROM to the stack.

If you use either the -y option or the --initializers_in_flash option, the aggregate initializers are located in segments with the suffix _F, which are copied from flash memory to the stack. The advantage of storing these initializers in flash is that valuable data space is not wasted. The disadvantage is that copying from flash is slower.

Initialization of constant objects

There are different ways of initializing constant objects.

By default, constant objects are placed in segments with the suffix _C, which are located in the optional external ROM that resides in the data memory space. The reason for this is that it must be possible for a default pointer—a pointer without explicit memory attributes—to point to the object, and a default pointer can only point to the data memory space.

However, if you do not have any external ROM in the data memory space, and for single chip applications you most likely do not have it, the constant objects have to be placed in RAM and initialized as any other non-constant variables. To achieve this, use the -y option, which means the objects are placed in segments with the suffix _ID.

If you want to place an object in flash memory, you can use any of the memory attributes __tinyflash, __flash, __farflash, or __hugeflash. The object becomes a flash object, which means you cannot take the address of it and store it in a default pointer. However, it is possible to store the address in either a __flash pointer or a __generic pointer, though neither of these are default pointers. Note that if you attempt to take the address of a constant __flash object and use it as a default pointer object, the compiler will issue an error. If you make an explicit cast of the object to a default pointer object, the error message disappears, instead there will be problems at runtime as the cast cannot copy the object from the flash memory to the data memory.

To access strings located in flash memory, you must use alternative library routines that expect flash strings. A few such alternative functions are provided—they are declared in the pgmspace.h header file. They are flash alternatives for some common C library functions with an extension _P. For your own code, you can always use the __flash keyword when passing the strings between functions. For reference information about the alternative functions, see *AVR-specific library functions*, page 370.

Stack usage analysis

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

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

INTRODUCTION TO STACK USAGE ANALYSIS

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

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

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

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

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

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

PERFORMING A STACK USAGE ANALYSIS

1 Enable stack usage analysis:

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

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

See the *IAR Linker and Library Tools Reference Guide* for information about the linker option `--enable_stack_usage`.

2 Enable the linker map file:

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

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

3 Link your project. Note that the linker will issue warnings related to stack usage under certain circumstances, see *Situations where warnings are issued*, page 84.

4 Review the linker map file, which now contains a stack usage chapter with a summary of the stack usage for each call graph root. For more information, see *Result of an analysis—the map file contents*, page 80.

5 For more details, analyze the call graph log, see *Call graph log*, page 84.

Note that there are limitations and sources of inaccuracy in the analysis, see *Limitations*, page 83.

You might need to specify more information to the linker to get a more representative result. See *Specifying additional stack usage information*, page 82

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

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

See the *IAR Linker and Library Tools Reference Guide* for information about the linker option `--stack_usage_control`.

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

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

or

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

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

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

See also *check that directive*, page 392 and *Stack considerations*, page 193.

RESULT OF AN ANALYSIS—THE MAP FILE CONTENTS

When stack usage analysis is enabled, the linker map file contains a stack usage chapter with a summary of the stack usage for each call graph root category, and lists the call

chain that results in the maximum stack depth for each call graph root. This is an example of what the stack usage chapter in the map file might look like:

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

  Call Graph Root Category  Max Use  Total Use
  ------------------------  -------  ---------
  interrupt                       4          4
  Program entry                 350        350

Program entry
  "__program_start": 0x0000c0f8

  Maximum call chain                          350 bytes

    "__program_start"                           0
    "main"                                      4
    "WriteObject"                              24
    "DumpObject"                                0
    "PrintObject"                               8
    "fprintf"                                   4
    "_PrintfLarge"                            126
    "_PutstrLarge"                            100
    "pad"                                      14
    "_PutcharsLarge"                           10
    "_FProut"                                   6
    "fputc"                                     6
    "_Fwprep"                                   6
    "fseek"                                     4
    "_Fspos"                                   14
    "fflush"                                    6
    "fflushOne"                                 6
    "__write"                                   0
    "__dwrite"                                 10
    "__DebugBreak"                              2

interrupt
  "DoStuff()": 0x0000e9ee

  Maximum call chain                            4 bytes

    "DoStuff()"                                 4
```

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

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

SPECIFYING ADDITIONAL STACK USAGE INFORMATION

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

You can:

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

```
function MyFunc: (CSTACK 32, RSTACK 4),
  calls MyFunc2,
  calls MyFunc3, MyFunc4: (CSTACK 16, RSTACK 4);

function [interrupt] MyInterruptHandler: 44;
```

See also *function directive*, page 393.

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

```
exclude MyFunc5, MyFunc6;
```

See also *exclude directive*, page 393.

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

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

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

See also *possible calls directive*, page 394 and *calls*, page 322.

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

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

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

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

See also *call graph root directive*, page 391 and *call_graph_root*, page 322.

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

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

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

```
no calls from [file.r90] to MyFunc13, MyFunc14;
```

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

For more information, see the chapter *The stack usage control file*, page 391.

LIMITATIONS

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

- The linker cannot always identify all functions in object modules that lack stack usage information. In particular, this might be a problem with object modules written in assembly language. You can provide stack usage information for such modules using a stack usage control file, and for assembly language modules you can also annotate the assembler source code with `CFI` directives to provide stack usage information. See the *IAR Assembler User Guide for AVR*.

- 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.

- The support for C++ is very incomplete. In particular virtual function calls, dynamic objects, and various indirect calls from the standard library (like the functions set up by `set_new_handler` and `register_callback`) are not supported at all.

- 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.

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

SITUATIONS WHERE WARNINGS ARE ISSUED

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

● There is a function without stack usage information.

● There is an indirect call site in the application for which a list of possible called functions has not been supplied.

● There are no known indirect calls, but there is an uncalled function that is not known to be a call graph root.

● The application contains recursion (a cycle in the call graph) for which no maximum recursion depth has been supplied, or which is of a form for which the linker is unable to calculate a reliable estimate of stack usage.

● There are calls to a function declared as a call graph root.

● You have used the stack usage control file to supply stack usage information for functions in a module that does not have such information, and there are functions referenced by that module which have not been mentioned as being called in the stack usage control file.

CALL GRAPH LOG

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

Example output:

```
Program entry:
0 __program_start [350]
  0 __data16_memzero [2]
    2 - [0]
  0 __data16_memcpy [2]
    0 memcpy [2]
      2 - [0]
    2 - [0]
  0 main [350]
    4 ParseObject [52]
      28 GetObject [28]
        34 getc [22]
          38 _Frprep [18]
            44 malloc [12]
              44 __data16_malloc [12]
                48 __data16_findmem [8]
                  52 __data16_free [4]
                    56 - [0]
                  52 __data16GetMemChunk [2]
                    54 - [0]
              46 - [0]
            44 __read [12]
              54 __DebugBreak [2]
                56 - [0]
        36 - [0]
      34 CreateObject [18]
        40 malloc [12] ***
    4 ProcessObject [326]
      8 ProcessHigh [76]
        34 ProcesMedium [50]
          60 ProcessLow [24]
            84 - [0]
      8 DumpObject [322]
        8 PrintObject [322]
          16 fprintf [314]
            20 _PrintfLarge [310]
        10 - [0]
    4 WriteObject [346]
      28 DumpObject [322] ***
    4 DestroyObject [28]
      28 free [4]
        28 __data16_free [4] ***
        30 - [0]
0 exit [38]
  0 _exit [38]
    4 _Close_all [34]
```

```
 8 fclose [30]
   14 _Fofree [4]
      14 free [4] ***
      16 - [0]
   14 fflush [24] ***
   14 free [4] ***
   14 __close [8]
      20 __DebugBreak [2] ***
   14 remove [8]
      20 __DebugBreak [2] ***
 8 __write [12] ***
2 __exit [8]
   8 __DebugBreak [2] ***
2 - [0]
```

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.

Linking your application

- Linking considerations

- Verifying the linked result of code and data placement

Linking considerations

When you set up your project in the IAR Embedded Workbench IDE, a default linker configuration file is automatically used based on your project settings and you can simply link your application. For the majority of all projects it is sufficient to configure the vital parameters that you find in **Project>Options>Linker>Config**.

When you build from the command line, you can use a ready-made linker command file provided with your product package.

The `config` directory contains the information required by XLINK, and are ready to be used as is. The only change, if any, you will normally have to make to the supplied configuration file is to customize it so it fits 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.

If you find that the default linker configuration file does not meet your requirements, you might want to consider:

- Placing segments
- Placing data
- Setting up stack memory
- Setting up heap memory
- Placing code
- Keeping modules
- Keeping symbols and segments
- Application startup
- Interaction between XLINK and your application
- Producing other output formats than UBROF

PLACING SEGMENTS

The placement of segments in memory is performed by the IAR XLINK Linker.

This section describes the most common linker commands and how to customize the linker configuration file to suit the memory layout of your target system. In demonstrating the methods, fictitious examples are used.

In demonstrating the methods, fictitious examples are used based on this memory layout:

- There is 1 Mbyte addressable memory.
- There is ROM memory in the address ranges `0x0000–0x1FFF`, `0x3000–0x4FFF`, and `0x10000–0x1FFFF`.
- There is RAM memory in the address ranges `0x8000–0xAFFF`, `0xD000–0xFFFF`, and `0x20000–0x27FFF`.
- There are two addressing modes for data, one for near memory and one for far memory.
- There is one stack and one heap.
- There are two addressing modes for code, one for nearfunc memory and one for farfunc memory.

Note: Even though you have a different memory map, for example if you have additional memory spaces (EEPROM) and additional segments, you can still use the methods described in the following examples.

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

For the result of each placement directive after linking, inspect the segment map in the list file (created by using the command line option `-x`).

General hints for placing segments

When you consider where in memory you should place your segments, it is typically a good idea to start placing large segments first, then placing small segments.

In addition, you should consider these aspects:

- Start placing the segments that must be placed on a specific address. This is, for example, often the case with the segment holding the reset vector.
- Then consider placing segments that hold content that requires continuous memory addresses, for example the segments for the stack and heap.

● When placing code and data segments for different addressing modes, make sure to place the segments in size order (the smallest memory type first).

Note: Before the linker places any segments in memory, the linker will first place the absolute segments.

Using the -Z command for sequential placement

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

The following illustrates how to use the -Z command to place the segment MYSEGMENTA followed by the segment MYSEGMENTB in CONST memory (that is, ROM) in the memory range 0x0000-0x1FFF.

```
-Z(CONST)MYSEGMENTA,MYSEGMENTB=0000-1FFF
```

To place two segments of different types continuous in the same memory area, do not specify a range for the second segment. In the following example, the MYSEGMENTA segment is first located in memory. Then, the rest of the memory range could be used by MYCODE.

```
-Z(CONST)MYSEGMENTA=0000-1FFF
-Z(CODE)MYCODE
```

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

```
-Z(CONST)MYSMALLSEGMENT=0000-01FF
-Z(CONST)MYLARGESEGMENT=0000-1FFF
```

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

Using the -P command for packed placement

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

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

```
-P(DATA)MYDATA=8000-AFFF,D000-FFFF
```

If your application has an additional RAM area in the memory range `0x6000-0x67FF`, you can simply add that to the original definition:

```
-P(DATA)MYDATA=8000-AFFF,D000-FFFF,6000-67FF
```

The linker can then place some parts of the MYDATA segment in the first range, and some parts in the second range. If you had used the `-Z` command instead, the linker would have to place all segment parts in the same range.

Note: Copy initialization segments—*BASENAME*_I and *BASENAME*_ID—and dynamic initialization segments must be placed using `-Z`.

Note: The AVR microcontroller is based on the Harvard architecture—thus code and data have separate memory spaces and require different access mechanisms. For more information, see the chapter *Data storage*.

PLACING DATA

Static memory is memory that contains variables that are global or declared static.

To get a clear understanding about how the data segments work, you must be familiar with the different memory types and the different memory models available in the compiler. For information about these details, see the chapter *Data storage*.

Placing static memory data segments

Depending on their memory attribute, static data is placed in specific segments. For information about the segments used by the compiler, see *Static data memory segments*, page 76.

For example, these commands can be used to place the static data segments:

```
/* First, the segments to be placed in ROM are defined. */
-Z(CONST)NEAR_C=0000-1FFF,3000-4FFF
-Z(CONST)FAR_C=0000-1FFF,3000-4FFF,10000-1FFFF
-Z(CONST)NEAR_ID,FAR_ID=010000-1FFFF

/* Then, the RAM data segments are placed in memory. */
-Z(DATA)NEAR_I,DATA16_Z,NEAR_N=8000-AFFF
-Z(DATA)FAR_I,FAR_Z,FAR_N=20000-27FFF
```

All the data segments are placed in the area used by on-chip RAM.

Placing located data

A variable that is explicitly placed at an address, for example by using the `#pragma location` directive or the `@` operator, is placed in for example either the `NEAR_AC` or the `NEAR_AN` segment. The former is used for constant-initialized data, and the latter for items declared as `__no_init`. The individual segment part of the segment knows its

location in the memory space, and it does not have to be specified in the linker configuration file.

Placing user-defined segments

If you create your own segments by using for example the `#pragma location` directive or the `@` operator, these segments must also be defined in the linker configuration file using the `-Z` or `-P` segment control directives.

SETTING UP STACK MEMORY

In this example, the data segment for holding the stack is called `CSTACK`. The system startup code initializes the stack pointer to point to the end of the stack segment.

Allocating a memory area for the stack is performed differently when using the command line interface, as compared to when using the IDE.

For more information about stack memory, see *Stack considerations*, page 193.

Stack size allocation in the IDE

Choose **Project>Options**. In the **General Options** category, click the **System** tab.

Add the required stack size in the dedicated text box.

Stack size allocation from the command line

The size of the `CSTACK` segment is defined in the linker configuration file.

The linker configuration file sets up a constant, representing the size of the stack, at the beginning of the file. Specify the appropriate size for your application, in this example 1024 bytes:

```
-D_..X_CSTACK_SIZE=400  /* 1024 bytes of stack size */
```

Note that the size is written hexadecimally, but not necessarily with the `0x` notation.

Placing the stack segment

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

```
-Z(DATA)CSTACK+_CSTACK_SIZE=8000-AFFF
```

Note:

● This range does not specify the size of the stack; it specifies the range of the available memory.

● The = allocates the `CSTACK` segment at the start of the memory area.

SETTING UP HEAP MEMORY

The heap contains dynamic data allocated by the C function `malloc` (or a corresponding function) or the C++ operator `new`.

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

● The linker segments used for the heap, see *Dynamic memory on the heap*, page 61

● The steps for allocating the heap size, which differs depending on which build interface you are using

● The steps involved for placing the heap segments in memory.

See also *Heap considerations*, page 194.

In this example, the data segment for holding the heap is called HEAP.

Heap size allocation in the IDE

Choose **Project>Options**. In the **General Options** category, click the **Heap Configuration** tab.

Add the required heap size in the dedicated text box.

Heap size allocation from the command line

The size of the HEAP segment is defined in the linker configuration file.

The linker configuration file sets up a constant, representing the size of the heap, at the beginning of the file. Specify the appropriate size for your application, in this example 1024 bytes:

```
-D_HEAP_SIZE=400    /* 1024 bytes for heap memory */
```

Note that the size is written hexadecimally, but not necessarily with the `0x` notation.

In many linker configuration files provided with IAR Embedded Workbench, these lines are prefixed with the comment character `//` because the IDE controls the heap size allocation. To make the directive take effect, remove the comment character.

 If you use a heap, you should allocate at least 512 bytes for it, to make it work properly.

Placing the heap segment

The actual HEAP segment is allocated in the memory area available for the heap:

```
-Z(DATA)HEAP+_HEAP_SIZE=8000-AFFF
```

Note: This range does not specify the size of the heap; it specifies the range of the available memory.

PLACING CODE

This section contains descriptions of the segments used for storing code and the interrupt vector table. For information about all segments, see *Summary of segments*, page 375.

Normal code

Depending on their memory attribute, functions are placed in various segments, in this example NEARFUNC and FARFUNC.

Placing the segments is a simple operation in the linker configuration file:

```
-P(CODE)NEARFUNC=0000-1FFF,3000-4FFF
-P(CODE)FARFUNC=0000-1FFF,3000-4FFF,10000-1FFFF
```

Here, the -P linker directive is used for allowing XLINK to split up the segments and pack their contents more efficiently. This is also useful if the memory range is non-continuous.

For information about segments holding normal code, see the chapter *Segment reference*.

Interrupt vectors

The interrupt vector table contains pointers to interrupt routines, including the reset routine. In this example, the table is placed in the segment INTVEC. The linker directive would then look like this:

```
-Z(CONST)INTVEC=0000-00FF
```

For more information about the interrupt vectors, see *Interrupt vectors and the interrupt vector table*, page 65.

C++ dynamic initialization

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

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

For example:

```
-Z(CONST)DIFUNCT=0000-1FFF,3000-4FFF
```

DIFUNCT must be placed using -Z. For more information, see *DIFUNCT*, page 378.

Additional compiler-generated segments

The compiler uses a set of internally generated segments, which are used for storing information that is vital to the operation of your application.

- The SWITCH segment which contains data statements used in the switch library routines. These tables are encoded in such a way as to use as little space as possible.
- The INITTAB segment contains the segment initialization description blocks that are used by the `__segment_init` function which is called by CSTARTUP. This table consists of a number of `SegmentInitBlock_Type` objects. This type is declared in the `segment_init.h` file which is located in the `avr\src\lib` directory.

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

```
-Z(CODE)SWITCH, INITTAB=0-1FFF
```

KEEPING MODULES

If a module is linked as a program module, it is always kept. That is, it will contribute to the linked application. However, if a module is linked as a library module, it is included only if it is symbolically referred to from other parts of the application that have been included. This is true, even if the library module contains a root symbol. To assure that such a library module is always included, use `-A` to make all modules in the file be treaded as if they were program modules:

```
-A file.r90
```

Use `-C` to makes all modules in the file be treated as if they were library modules:

```
-C file.r90
```

KEEPING SYMBOLS AND SEGMENTS

By default, XLINK removes any segments, segment parts, and global symbols that are not needed by the application. To retain a symbol that does not appear to be needed—or actually, the segment part it is defined in—you can either use the `__root` attribute on the symbol in your C/C++ source code or ROOT in your assembler source code, or use the XLINK option `-g`.

For information about included and excluded symbols and segment parts, inspect the map file (created by using the XLINK option `-xm`).

For more information about the linking procedure for keeping symbols and sections, see *The linking process in detail*, page 73.

APPLICATION STARTUP

By default, the point where the application starts execution is defined by the `__program_start` label, which is defined to point at the reset vector. The label is also communicated via the debugger information to any debugger.

To change the start point of the application to another label, use the XLINK option `-s`.

INTERACTION BETWEEN XLINK AND YOUR APPLICATION

Use the XLINK option `-D` to define symbols that can be used for controlling your 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.

To change a reference to one symbol to another symbol, use the XLINK command line option `-e`. 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 information about the addresses and sizes of all global (statically linked) symbols, inspect the entry list in the map file (the XLINK option `-xm`).

PRODUCING OTHER OUTPUT FORMATS THAN UBROF

XLINK can generate more than 30 industry-standard loader formats, in addition to the proprietary format UBROF which is used by the C-SPY debugger. For a complete list, see the *IAR Linker and Library Tools Reference Guide*. To specify a different output format than the default, use the XLINK option `-F`. For example:

```
-F intel-standard
```

Note that it can be useful to use the XLINK `-O` option to produce two output files, one for debugging and one for burning to ROM/flash.

Note also that if you choose to enable debug support using the `-r` option for certain low-level I/O functionality for mechanisms like file operations on the host computer etc, such debug support might have an impact on the performance and responsiveness of your application. In this case, the debug build will differ from your release build due to the debug modules included.

Verifying the linked result of code and data placement

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

SEGMENT TOO LONG ERRORS AND RANGE ERRORS

Code or data that is placed in a relocatable segment will have its absolute address resolved at link time. Note that it is not known until link time whether all segments will fit in the reserved memory ranges. If the contents of a segment do not fit in the address range defined in the linker configuration file, XLINK will issue a *segment too long* error.

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

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

LINKER MAP FILE

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

● A segment map which lists all segments in address order

● A module map which lists all segments, local symbols, and entries (public symbols) for every module in the program. All symbols not included in the output can also be listed

● A module summary which lists the contribution (in bytes) from each module

● A symbol list which contains every entry (global symbol) in every module.

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

Normally, XLINK will not generate an output file if any errors, such as range errors, occur during the linking process. Use the option **Always generate output** in the IDE, or the option -B on the command line, to generate an output file even if a non-fatal error was encountered.

For more information about the listing options and the linker listing, see the *IAR Linker and Library Tools Reference Guide*, and the *IDE Project Management and Building Guide for AVR*.

MANAGING MULTIPLE MEMORY SPACES

Output formats that do not support more than one memory space—like MOTOROLA and INTEL-HEX—might require up to one output file per memory space. This causes no problems if you are only producing output to one memory space (flash), but if you also

are placing objects in EEPROM or an external ROM in DATA space, the output format cannot represent this, and the linker issues this error message:

```
Error[e133]: The output format Format cannot handle multiple
address spaces. Use format variants (-y -O) to specify which
address space is wanted.
```

To limit the output to flash memory, make a copy of the linker configuration file for the derivative and memory model you are using, and put it in the project directory. Add this line to the file:

```
-y(CODE)
```

To produce output for the other memory space(s), you must generate one output file per memory space (because the output format you chose does not support more than one memory space). Use the XLINK option -o for this purpose.

For each additional output file, you must specify format, XLINK segment type, and file name. For example:

```
-Omotorola,(DATA)=external_rom.a90
-Omotorola,(XDATA)=eeprom.a90
```

Note: As a general rule, an output file is only necessary if you use non-volatile memory. In other words, output from the data space is only necessary if the data space contains external ROM.

For more information, see the *IAR Linker and Library Tools Reference Guide*.

The IAR Postlink utility

You can also use the IAR Postlink utility, delivered with the compiler. This application takes as input an object file (of the XLINK simple format) and extracts one or more of its XLINK segment types into one file (which can be in either Intel extended hex format or Motorola S-record format). For example, it can put all code segments into one file, and all EEPROM segments into another.

See the postlink.htm document for more information about IAR Postlink.

The DLIB runtime environment

- Introduction to the runtime environment

- Setting up the runtime environment

- Additional information on the runtime environment

- Managing a multithreaded environment

DLIB can be used with both the C and the C++ languages. CLIB, on the other hand, can only be used with the C language. For more information about CLIB, see the chapter *The CLIB runtime environment*.

Introduction to the runtime environment

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

This section contains information about:

RUNTIME ENVIRONMENT FUNCTIONALITY

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

- The *C/C++ standard library*, both its interface (provided in the system header files) and its implementation.
- Startup and exit code.
- Low-level I/O interface for managing input and output (I/O).
- Special compiler support, for instance functions for switch handling or integer arithmetics.

- Support for hardware features:
  - Direct access to low-level processor operations by means of *intrinsic* functions, such as functions for interrupt mask handling
  - Peripheral unit registers and interrupt definitions in include files
  - Special compiler support for accessing strings in flash memory, see *AVR-specific library functions*, page 370.

Runtime environment functions are provided.

The runtime library is delivered both as and (depending on your product package) as source files. The prebuilt libraries are available in different *configurations* to meet various needs, see *Runtime library configurations*, page 110. You can find the libraries in the product subdirectories `avr\lib` and `avr\src\lib`, respectively.

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

BRIEFLY ABOUT INPUT AND OUTPUT (I/O)

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

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

On a desktop computer or a server, the operating system is expected to provide I/O functionality to the application via the standard I/O interface in the runtime environment. However, in an embedded system, the runtime library cannot assume that such functionality is present, or even that there is an operating system at all. Therefore,

the low-level part of the standard I/O interface is not completely implemented by default:
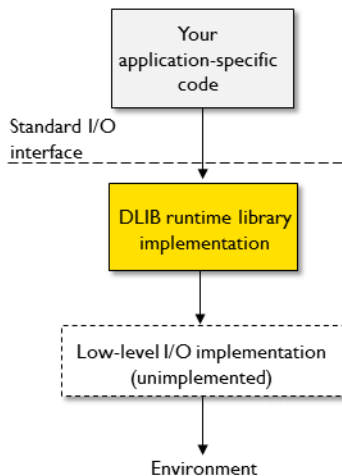

To make the standard I/O interface work, you can:

● Let the C-SPY debugger emulate I/O operations on the host computer, see *Briefly about C-SPY emulated I/O*, page 102

● *Retarget* the standard I/O interface to your target system by providing a suitable implementation of the interface, see *Briefly about retargeting*, page 102.

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

BRIEFLY ABOUT C-SPY EMULATED I/O

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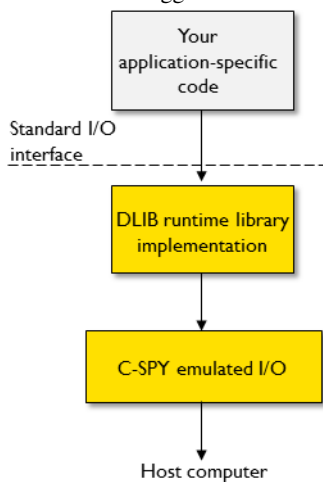
For example, when C-SPY emulated I/O is enabled:

● Standard character streams are directed to the C-SPY **Terminal I/O** window

● File system operations are performed on the host computer

● Time and date functions return the time and date of the host computer

● Termination and failed asserts break execution and notify the C-SPY debugger.

This behavior can be valuable during the early development of an application, for example in an application that uses file I/O before any flash file system I/O drivers are implemented, or if you need to debug constructions in your application that use stdin and stdout without the actual hardware device for input and output being available.

See *Setting up your runtime environment*, page 104 and *The C-SPY emulated I/O mechanism*, page 116.

BRIEFLY ABOUT RETARGETING

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

The standard I/O interface is large and complex. To make retargeting easier, the DLIB runtime environment is designed so that it performs all I/O operations through a small set of simple functions, which is referred to as the *DLIB low-level I/O interface*. By default, the functions in the low-level interface lack usable implementations. Some are

unimplemented, others have stub implementations that do not perform anything except returning error codes.

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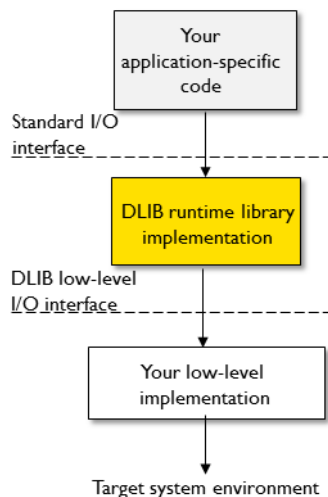
For example, if your application calls the functions printf and fputc in the standard I/O interface, the implementations of those functions both call the low-level function __write to output individual characters. To make them work, you just need to provide an implementation of the __write function—either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

For information about how to override library modules with your own implementations, see *Overriding library modules*, page 107. See also *The DLIB low-level I/O interface*, page 123 for information about the functions that are part of the interface.

Setting up the runtime environment

This section contains these tasks:

- *Setting up your runtime environment*, page 104

  A runtime environment with basic project settings to be used during the initial phase of development.

- *Retargeting—Adapting for your target system*, page 106
- *Overriding library modules*, page 107

● *Customizing and building your own runtime library*, page 108

See also:

● *Managing a multithreaded environment*, page 135 for information about how to adapt the runtime environment to treat all library objects according to whether they are global or local to a thread.

SETTING UP YOUR RUNTIME ENVIRONMENT

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

To set up the runtime environment:

1 Before you build your project, choose **Project>Options>General Options** to open the **Options** dialog box.

2 On the **Library Configuration** page, verify the following settings:

● **Library**: choose which *library* and *library configuration* to use. Typically, choose **Normal**, or **Full**.

   For information about the various library configurations, see *Runtime library configurations*, page 110.

   CLIB or DLIB—for more information about the libraries, see *C/C++ standard library overview*, page 361.

3 On the **Library Options** page, select **Auto** for both **Printf formatter** and **Scanf formatter**. This means that the linker will automatically choose the appropriate formatters based on information from the compiler. For more information about the available formatters and how to choose one manually, see *Formatters for printf*, page 113 and *Formatters for scanf*, page 114, respectively.

4 To enable C-SPY emulated I/O, choose **Project>Options>Linker>Output** and select the **Format** option that matches the level of support you need. See *Briefly about C-SPY emulated I/O*, page 102. Choose between:

Linker option in the IDE	Linker command line option	Description
Debug information for C-SPY	`-Fubrof`	Debug support for C-SPY, but without any support for C-SPY emulated I/O.

Table 10: Debug information and levels of C-SPY emulated I/O

Linker option in the IDE	Linker command line option	Description
With runtime control modules	`-r`	The same as `-Fubrof` but also limited support for C-SPY emulated I/O handling program abort, exit, and assertions.
With I/O emulation modules	`-rt`	Full support for C-SPY emulated I/O, which means the same as `-r`, but also support for I/O handling, and accessing files on the host computer during debugging.

Table 10: Debug information and levels of C-SPY emulated I/O (Continued)

Note: The C-SPY **Terminal I/O** window is not opened automatically; you must open it manually. For more information about this window, see the *C-SPY® Debugging Guide for AVR*.

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

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

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

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

```
-e__write_buffered=__write
```

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

For more information, see *Math functions*, page 116.

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

For information about which project settings affect the choice of library file, see *Runtime library configurations*, page 110.

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

RETARGETING—ADAPTING FOR YOUR TARGET SYSTEM

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

To adapt your runtime environment for your target system:

1 Adapt system initialization.

It is likely that you must adapt the system initialization, for example, your application might need to initialize interrupt handling, I/O handling, watchdog timers, etc. You do this by implementing the routine `__low_level_init`, which is executed before the data segments are initialized. See *System startup and termination*, page 118 and *System initialization*, page 122. Note that you can find device-specific examples on this in the example projects provided in the product installation; see the Information Center.

2 Adapt the runtime library for your target system. To implement such functions, you need a good understanding of the DLIB low-level I/O interface, see *Briefly about retargeting*, page 102.

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

- Standard streams for input and output

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

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

- File input and output

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

- `signal` and `raise`

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

- Time and date

   To make the time and date functions work, you must implement the functions `clock`, `__time32`, `__time64`, and `__getzone`. Whether you use `__time32` or `__time64` depends on which interface you use for `time_t`, see *time.h*, page 368.

- Assert, see *_ReportAssert*, page 129.
- Environment interaction

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

For more information about the functions, see *The DLIB low-level I/O interface*, page 123.

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

3 When you have implemented your functions of the low-level I/O interface, you must add your version of these functions to your project. For information about this, see *Overriding library modules*, page 107.

Note: If you have implemented a DLIB low-level I/O interface function and added it to a project that you have built with support for C-SPY emulated I/O, your low-level function will be used and not the functions provided with C-SPY emulated I/O. For example, if you implement your own version of `__write`, output to the C-SPY **Terminal I/O** window will not be supported. See *Briefly about C-SPY emulated I/O*, page 102.

4 Before you can execute your application on your target system, you must rebuild your project with a Release build configuration. This means that the linker will not include the C-SPY emulated I/O mechanism and the low-level I/O functions it provides. If your application calls any of the low-level functions of the standard I/O interface, either directly or indirectly, and your project does not contain these, the linker will issue an error for every missing low-level function. Also, note that the `NDEBUG` symbol is defined in a Release build configuration, which means asserts will no longer be generated. For more information, see *_ReportAssert*, page 129.

OVERRIDING LIBRARY MODULES

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

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

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

2 Modify the file.

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

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

Note: If you have implemented a DLIB low-level I/O interface function and added it to a project that you have built with support for C-SPY emulated I/O, your low-level function will be used and not the functions provided with C-SPY emulated I/O. For example, if you implement your own version of `__write`, output to the C-SPY **Terminal I/O** window will not be supported. See *Briefly about C-SPY emulated I/O*, page 102.

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

CUSTOMIZING AND BUILDING YOUR OWN RUNTIME LIBRARY

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

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

● There is no prebuilt library for the required combination of compiler options or hardware support

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

In those cases, you must:

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

● Set up a library project

● Make the required library customizations

● Build your customized runtime library

● Finally, make sure your application project will use the customized runtime library.

To set up a library project:

1 In the IDE, choose **Project>Create New Project** and use the library project template which can be used for customizing the runtime environment configuration. There is a library template for CLIB and a template for DLIB, see *Runtime library configurations*, page 110

Note that when you create a new library project from a template, the majority of the files included in the new project are the original installation files. If you are going to modify

these files, make copies of them first and replace the original files in the project with these copies.

2 Modify the project options in the created library project to suit your application, see *Basic project configuration*, page 48.

To customize the library functionality:

1 The library functionality is determined by a set of *configuration symbols*. The default values of these symbols are defined in the file `DLib_Defaults.h` which you can find in `avr\inc\DLIB\c`. This read-only file describes the configuration possibilities. Note that you should not modify this file.

In addition, your custom library has its own *library configuration file* `dlavrCustom.h`—which you can find in `avr\config\template\project`—and which sets up that specific library with the required library configuration. Customize this file by setting the values of the configuration symbols according to the application requirements. For example, to enable the `ll` qualifier in `printf` format strings, write in your library configuration file:

```
#define _DLIB_PRINTF_LONG_LONG 1
```

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

● *Configuration symbols for printf and scanf*, page 132
● *Configuration symbols for file input and output*, page 133
● *Locale*, page 133
● *Strtod*, page 135
● *Managing a multithreaded environment*, page 135

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

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

To build IAR Embedded Workbench projects from the command line, use the IAR Command Line Build Utility (`iarbuild.exe`). There is also a batch file (`build_libs.bat`) provided for building the library from the command line. For information about the build process and the IAR Command Line Build Utility, see the *IDE Project Management and Building Guide for AVR*.

To use the customized runtime library in your application project:

1 In the IDE, choose **Project>Options>General Options** and click the **Library Configuration** tab.

2 From the **Library** drop-down menu, choose **Custom DLIB**.

3 In the **Library file** text box, locate your library file.

4 In the **Configuration file** text box, locate your library configuration file.

Additional information on the runtime environment

This section gives additional information on the runtime environment:

- *Runtime library configurations*, page 110
- *Prebuilt runtime libraries*, page 111
- *Formatters for printf*, page 113
- *Formatters for scanf*, page 114
- *The C-SPY emulated I/O mechanism*, page 116
- *Math functions*, page 116
- *System startup and termination*, page 118
- *System initialization*, page 122
- *The DLIB low-level I/O interface*, page 123
- *Configuration symbols for printf and scanf*, page 132
- *Configuration symbols for file input and output*, page 133
- *Locale*, page 133
- *Strtod*, page 135

RUNTIME LIBRARY CONFIGURATIONS

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

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

These predefined library configurations are available:

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

Table 11: Library configurations

Note: In addition to these predefined library configurations, you can provide your own configuration, see *Customizing and building your own runtime library*, page 108

If you do not specify a library configuration explicitly you will get the default configuration. If you use a prebuilt runtime library, a configuration file that matches the runtime library file will automatically be used. See *Setting up your runtime environment*, page 104.

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

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

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

Use the `--dlib_config` compiler option, see *--dlib_config*, page 249.

The prebuilt libraries are based on the default configurations, see *Runtime library configurations*, page 110.

2 If you have built your own customized library, choose **Project>Options>Library Configuration>Library** and choose **Custom** to use your own configuration. For more information, see *Customizing and building your own runtime library*, page 108.

PREBUILT RUNTIME LIBRARIES

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

- Type of library
- Processor option (`-v`)
- Memory model option (`--memory_model`)
- AVR enhanced core option (`--enhanced_core`)
- Small flash memory option (`--64k_flash`)
- 64-bit doubles option (`--64bit_doubles`)
- Library configuration—Normal or Full.

To choose a library from the command line:

If you build your application from the command line, make the following settings:

- Specify which library file to use on the XLINK command line, like:

  dl*libname*.r90

- If you do not specify a library configuration, the default configuration will be used. However, you can specify the library configuration explicitly for the compiler:

  `--dlib_config C:\...\dl`*libname*`.h`

Note: All modules in the library have a name that starts with the character ? (question mark).

You can find the library files in the subdirectory `avr\lib\dlib` and the library configuration files in the `avr\inc\dlib` subdirectory.

Library filename syntax

The names of the libraries are constructed from these elements:

{*library*}	`dl` for the IAR DLIB runtime environment
{*target*}	is `avr`
{*memModel*}	Specifies the memory model:
	`t` = Tiny `s` = Small `l` = Large `h` = Huge
{*core*}	is `ec_mul` or `ec_nomul` when enhanced core is used depending on whether the `MUL` instruction is available or not, or `xmega` if the device is from the xmega family. If neither the enhanced core nor the xmega core is used, this value is not specified.
{*smallFlash*}	is `sf` when the small flash memory is available. When small flash memory is not available, this value is not specified
{*64bitDoubles*}	is `64` when 64-bit doubles are used. When 32-bit doubles are used, this value is not specified.
{*lib_config*}	Specifies the library configuration:
	`n` = Normal `f` = Full

For example, the library `dlavr-3s-ec_mul-sf-n.r90` is configured for the Small memory model, for the enhanced core with the `MUL` instruction available, for the small flash memory, and for the normal library configuration.

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

These are some examples of how to decode a library name:

Library	Generic processor option	Memory model	Core	Small flash	64-bit doubles	Library configuration
dlavr-3s-ec_mul-sf-n.r90	-v3	Small	X	X	--	Normal
dlavr-3s-ec_mul-64-f.r90	-v3	Small	X	--	X	Full

Table 12: Prebuilt libraries

FORMATTERS FOR PRINTF

The printf function uses a formatter called _Printf. The full version is quite large, and provides facilities not required in many embedded applications. To reduce the memory consumption, three smaller, alternative versions are also provided. Note that the wprintf variants are not affected.

This table summarizes the capabilities of the different formatters:

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

Table 13: Formatters for printf

† NoMb means without multibytes.

Note: It is possible to optimize these functions even further, but that requires that you rebuild the library. See *Configuration symbols for printf and scanf*, page 132.

The compiler can automatically detect which formatting capabilities are needed in a direct call to printf, if the formatting string is a string literal. This information is passed to the linker, which combines the information from all modules to select a suitable formatter for the application. However, if the formatting string is a variable, or if the call is indirect through a function pointer, the compiler cannot perform the

analysis, forcing the linker to select the Full formatter. In this case, you might want to override the automatically selected `printf` formatter.

To override the automatically selected printf formatter in the IDE:

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

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

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

1 Add one of these lines in the linker configuration file you are using:

```
-e_PrintfFull=_Printf
-e_PrintfFullNoMb=_Printf
-e_PrintfLarge=_Printf
-e_PrintfLargeNoMb=_Printf
_e_PrintfSmall=_Printf
-e_PrintfSmallNoMb=_Printf
-e_PrintfTiny=_Printf
-e_PrintfTinyNoMb=_Printf
```

To override the default AVR-specific `printf_P` formatter, type any of the following lines in your linker command file:

```
-e_small_write_P=_formatted_write_P
-e_medium_write_P=_formatted_write_P
```

Note: In the IDE, you can use **Project>Options>Linker>Extra Options** to override the default formatter for the AVR-specific library routines.

See also *printf_P*, page 371.

FORMATTERS FOR SCANF

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

This table summarizes the capabilities of the different formatters:

Formatting capabilities	Small/SmallNoMb†	Large/LargeNoMb†	Full/FullNoMb†
Basic specifiers c, d, i, o, p, s, u, X, x, and %	Yes	Yes	Yes
Multibyte support	Yes/No	Yes/No	Yes/No
Floating-point specifiers a, and A	No	No	Yes

Table 14: Formatters for scanf

Formatting capabilities	Small/ SmallNoMb†	Large/ LargeNoMb†	Full/ FullNoMb†
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 14: Formatters for scanf (Continued)

† NoMb means without multibytes.

Note: It is possible to optimize these functions even further, but that requires that you rebuild the library. See *Configuration symbols for printf and scanf*, page 132.

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

To manually specify the scanf formatter in the IDE:

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

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

To manually specify the scanf formatter from the command line:

1 Add one of these lines in the linker configuration file you are using:

```
-e_ScanfFull=_Scanf
-e_ScanfFullNoMb=_Scanf
-e_ScanfLarge=_Scanf
-e_ScanfLargeNoMb=_Scanf
_e_ScanfSmall=_Scanf
_e_ScanfSmallNoMb=_Scanf
```

To override the default AVR-specific scanf_P formatter, type the following line in your linker command file:

```
-e_medium_read_P=_formatted_read_P
```

Note: In the IDE, you can use **Project>Options>Linker>Extra Options** to override the default formatter for the AVR-specific library routines.

See also *scanf_P*, page 372.

THE C-SPY EMULATED I/O MECHANISM

The C-SPY emulated I/O mechanism works as follows:

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

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

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

4   The execution will then resume.

See also *Briefly about C-SPY emulated I/O*, page 102.

MATH FUNCTIONS

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

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

Smaller versions

The functions `cos`, `exp`, `log`, `log2`, `log10`, `__iar_Log` (a help function for `log`, `log2`,
and `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 specify individual math functions from the command line:

Redirect the default function names to these names when linking, using these options:

```
-e__iar_sin_small=sin
-e__iar_cos_small=cos
-e__iar_tan_small=tan
-e__iar_log_small=log
-e__iar_log2_small=log2
-e__iar_log10_small=log10
-e__iar_exp_small=exp
-e__iar_pow_small=pow
-e__iar_Sin_small=__iar_Sin
-e__iar_Log_small=__iar_Log

-e__iar_sin_smallf=sinf
-e__iar_cos_smallf=cosf
-e__iar_tan_smallf=tanf
-e__iar_log_smallf=logf
-e__iar_log2_smallf=log2f
-e__iar_log10_smallf=log10f
-e__iar_exp_smallf=expf
-e__iar_pow_smallf=powf
-e__iar_Sin_smallf=__iar_Sinf
-e__iar_Log_smallf=__iar_Logf

-e__iar_sin_smalll=sinl
-e__iar_cos_smalll=cosl
-e__iar_tan_smalll=tanl
-e__iar_log_smalll=logl
-e__iar_log2_smalll=log2l
-e__iar_log10_smalll=log10l
-e__iar_exp_smalll=expl
-e__iar_pow_smalll=powl
-e__iar_Sin_smalll=__iar_Sinl
-e__iar_Log_smalll=__iar_Logl
```

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 four 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 specify individual math functions from the command line:

Redirect the default function names to these names when linking, using these options:

```
-e__iar_sin_accurate=sin
-e__iar_cos_accurate=cos
-e__iar_tan_accurate=tan
-e__iar_pow_accurate=pow
-e__iar_Sin_accurate=__iar_Sin
-e__iar_Pow_accurate=__iar_Pow

-e__iar_sin_accuratef=sinf
-e__iar_cos_accuratef=cosf
-e__iar_tan_accuratef=tanf
-e__iar_pow_accuratef=powf
-e__iar_Sin_accuratef=__iar_Sinf
-e__iar_Pow_accuratef=__iar_Powf

-e__iar_sin_accuratel=sinl
-e__iar_cos_accuratel=cosl
-e__iar_tan_accuratel=tanl
-e__iar_pow_accuratel=powl
-e__iar_Sin_accuratel=__iar_Sinl
-e__iar_Pow_accuratel=__iar_Powl
```

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 pow or __iar_Pow, you must redirect both functions.

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.s90, _exit.s90, and low_level_init.c located in the avr\src\lib directory.

For information about how to customize the system startup code, see *System initialization*, page 122.

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:
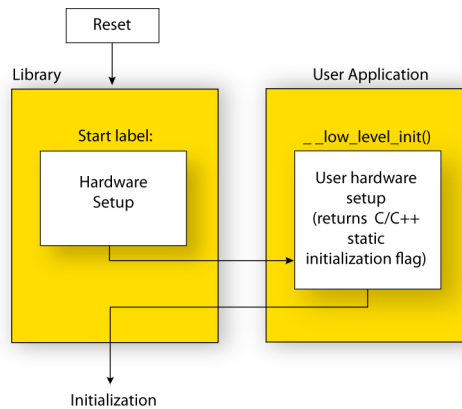
- When the CPU is reset it will start executing at the program entry label `__program_start` in the system startup code.
- The external data and address buses are enabled if needed.
- The stack pointers are initialized to the end of `CSTACK` and `RSTACK`, respectively.
- The function `__low_level_init` is called if you defined it, giving the application a chance to perform early initializations.

For the C/C++ initialization, it looks like this:
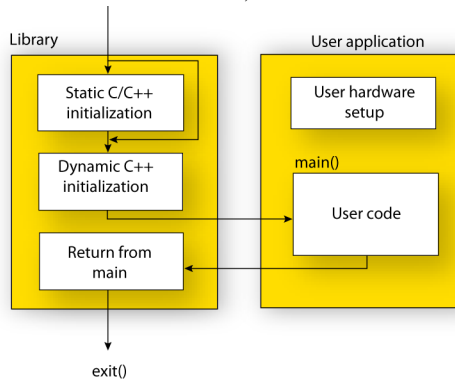
- Static and global variables are initialized except for `__no_init`, `__tinyflash`, `__flash`, `__farflash`, `__hugeflash`, and `__eeprom` declared variables. That is, zero-initialized variables are cleared and the values of other initialized variables are copied from ROM to RAM memory. This step is skipped if `__low_level_init` returns zero. For more information, see *Initialization at system startup*, page 75.

- Static C++ objects are constructed

- The `main` function is called, which starts the application.

System termination

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

An application can terminate normally in two different ways:

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

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

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

- Call functions registered to be executed when the application ends. This includes C++ destructors for static and global variables, and functions registered with the standard function `atexit`.
- 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 (and if using `atexit` is not sufficient), you can write your own implementation of the `__exit(int)` function.

The library files that you can override with your own versions are located in the `avr\src\lib` directory. See *Overriding library modules*, page 107.

C-SPY debugging support for system termination

If you have enabled C-SPY emulated I/O during linking, 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 emulate program termination. For more information, see *Briefly about C-SPY emulated I/O*, page 102.

SYSTEM INITIALIZATION

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

You can do this by implementing your own version of the routine `__low_level_init`, which is called from the file `cstartup` before the data segments are initialized. Modifying the file `cstartup.s90` directly should be avoided.

The code for handling system startup is located in the source files `cstartup.s90` and `low_level_init.c`, located in the `avr\src\lib` directory.

Note: Normally, you do not need to customize either of the files `cstartup.s90` or `_exit.s90`.

Note: Regardless of whether you implement your own version of `__low_level_init` or the file `cstartup.s90`, you do not have to rebuild the library.

Customizing __low_level_init

You can customize the routine `__low_level_init`, which is called from the system startup code before the data segments are initialized.

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

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 cstartup file

As noted earlier, you should not modify the file `cstartup.s90` if implementing your own version of `__low_level_init` is enough for your needs. However, if you do need to modify the file `cstartup.s90`, 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 107.

Note that you must make sure that the linker uses the start label used in your version of cstartup.s90. For information about how to change the start label used by the linker, read about the -s option in the *IAR Linker and Library Tools Reference Guide*.

THE DLIB LOW-LEVEL I/O INTERFACE

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

For more information about this, see *Briefly about input and output (I/O)*, page 100.

This table lists the functions in the DLIB low-level I/O interface:

Function in DLIB low-level I/O interface	Defined in source file
abort	abort.c
clock	clock.c
__close	close.c
__exit	xxexit.c
getenv	getenv.c
__getzone	getzone.c
__lseek	lseek.c
__open	open.c
raise	raise.c
__read	read.c
remove	remove.c
rename	rename.c
_ReportAssert	xreportassert.c
signal	signal.c
system	system.c
__time32, __time64	time.c
__write	write.c

Table 15: DLIB low-level I/O interface functions

Note: You should not use the low-level functions prefixed with __ directly in your application. Instead you should use the standard library functions that use these functions. For example, to write to stdout, you should use standard library functions like printf or puts, which in turn calls the low-level function __write. If you have forgot to implement a low-level function and your application will call that function via

a standard library function, the linker will issues an error when you link in release build configuration.

Note: If you implement your own variants of the functions in this interface, your variants will be used even though you have enabled C-SPY emulated I/O, see *Briefly about C-SPY emulated I/O*, page 102.

abort

Source file	`avr\src\lib\abort.c`
Description	Standard C library function that aborts execution.
C-SPY debug action	Notifies that the application has called abort.
Default implementation	Calls `__exit(EXIT_FAILURE)`.
See also	*Briefly about retargeting*, page 102
	System termination, page 121.

clock

Source file	`avr\src\lib\clock.c`
Description	Standard C library function that accesses the processor time.
C-SPY debug action	Returns the clock on the host computer.
Default implementation	Returns `-1` to indicate that processor time is not available.
See also	*Briefly about retargeting*, page 102.

__close

Source file	`avr\src\lib\close.c`
Description	Low-level function that closes a file.
C-SPY debug action	Closes the associated host file on the host computer.
Default implementation	None.

See also   *Briefly about retargeting*, page 102.

__exit

Source file   `avr\src\lib\xxexit.c`

Description   Low-level function that halts execution.

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

Default implementation   Loops forever.

See also   *Briefly about retargeting*, page 102

                            System termination, page 121.

getenv

Source file   `avr\src\lib\getenv.c`
                            `avr\src\lib\environ.c`

C-SPY debug action   Accesses the host environment.

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

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

                            `key=value\0`

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

                            For example:

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

                            If you need a more sophisticated environment variable handling, you should implement
                            your own getenv, and possibly putenv function.

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

See also   *Briefly about retargeting*, page 102.

__getzone

Source file   `avr\src\lib\getzone.c`

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

C-SPY debug action   Not applicable.

Default implementation   Returns `":"`.

See also   *Briefly about retargeting*, page 102.

__lseek

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

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

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

Default implementation   None.

See also   *Briefly about retargeting*, page 102.

__open

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

Description   Low-level function that opens a file.

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

Default implementation   None.

See also   *Briefly about retargeting*, page 102.

raise

Source file	`avr\src\lib\raise.c`
Description	Standard C library function that raises a signal.
C-SPY debug action	Not applicable.
Default implementation	Calls the signal handler for the raised signal, or terminates with call to `__exit(EXIT_FAILURE)`.
See also	*Briefly about retargeting*, page 102.

__read

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

Example The code in this example uses memory-mapped I/O to read from a keyboard, whose port is assumed to be located at `40`:

```
#include <stddef.h>

__no_init volatile unsigned char kbIO @ 40;

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

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

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

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

  return nChars;
}
```

For information about the handles associated with the streams, see *Retargeting—Adapting for your target system*, page 106.

For information about the @ operator, see *Controlling data and function placement in memory*, page 210.

See also *Briefly about retargeting*, page 102.

remove

Source file `avr\src\lib\remove.c`

Description Standard C library function that removes a file.

C-SPY debug action	Writes a message to the **Debug Log** window and returns -1.
Default implementation	Returns 0 to indicate success, but without removing a file.
See also	*Briefly about retargeting*, page 102.

rename

Source file	avr\src\lib\rename.c
Description	Standard C library function that renames a file.
C-SPY debug action	None.
Default implementation	Returns -1 to indicate failure.
See also	*Briefly about retargeting*, page 102.

_ReportAssert

Source file	avr\src\lib\xreportassert.c
Description	Low-level function that handles a failed assert.
C-SPY debug action	Notifies the C-SPY debugger about the failed assert.
Default implementation	Failed asserts are reported by the function __ReportAssert. By default, it prints an error message and calls abort. If this is not the behavior you require, you can implement your own version of the function.
	The assert macro is defined in the header file assert.h. To turn off assertions, define the symbol NDEBUG.
	In the IDE, the symbol NDEBUG is by default defined in a Release project and *not* defined in a Debug project. If you build from the command line, you must explicitly define the symbol according to your needs. See *NDEBUG*, page 359.
See also	*Briefly about retargeting*, page 102.

signal

Source file	avr\src\lib\signal.c

129

Description	Standard C library function that changes signal handlers.
C-SPY debug action	Not applicable.
Default implementation	As specified by Standard C. You might want to modify this behavior if the environment supports some kind of asynchronous signals.
See also	*Briefly about retargeting*, page 102.

system

Source file	avr\src\lib\system.c
Description	Standard C library function that executes commands.
C-SPY debug action	Notifies the C-SPY debugger that system has been called and then returns -1.
Default implementation	If you need to use the system function, you must implement it yourself. The system function available in the library returns 0 if a null pointer is passed to it to indicate that there is no command processor; otherwise it returns -1 to indicate failure. If this is not the functionality that you require, you can implement your own version. This does not require that you rebuild the library.
See also	*Briefly about retargeting*, page 102.

__time32, __time64

Source file	avr\src\lib\time.c avr\src\lib\time64.c
Description	Low-level functions that return the current calendar time.
C-SPY debug action	Returns the time on the host computer.
Default implementation	Returns -1 to indicate that calendar time is not available.
See also	*Briefly about retargeting*, page 102.

__write

Source file	avr\src\lib\write.c

Description	Low-level function that writes to stdout, stderr, or a file.
C-SPY debug action	Directs stdout and stderr to the **Terminal I/O** window. All other files will write to the associated host file.
Default implementation	None.

Example

The code in this example uses memory-mapped I/O to write to an LCD display, whose port is assumed to be located at address 40:

```
#include <stddef.h>

__no_init volatile unsigned char lcdIO @ 40;

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

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

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

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

  return nChars;
}
```

For information about the handles associated with the streams, see *Retargeting—Adapting for your target system*, page 106.

See also

Briefly about retargeting, page 102.

CONFIGURATION SYMBOLS FOR PRINTF AND SCANF

If the provided formatters do not meet your requirements (*Formatters for printf*, page 113 and *Formatters for scanf*, page 114), you can customize the Full formatters. Note that this 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 16: 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 17: Descriptions of scanf configuration symbols

To customize the formatting capabilities, you must;

1 Define the configuration symbols in the library configuration file according to your application requirements.

2 Rebuild the library, see *Customizing and building your own runtime library*, page 108.

CONFIGURATION SYMBOLS FOR FILE INPUT AND OUTPUT

File I/O is only supported by libraries with the Full library configuration, see *Runtime library configurations*, page 110, or in a customized library when the configuration symbol `__DLIB_FILE_DESCRIPTOR` is defined. If this symbol is not defined, functions taking a `FILE *` argument cannot be used.

To customize your library and rebuild it, see *Customizing and building your own runtime library*, page 108.

LOCALE

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

Depending on which library configuration you are using, you get different 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. See *Runtime library configurations*, page 110.

The DLIB runtime 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 and rebuilding the library

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

● The 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 *Customizing and building your own runtime library*, page 108.

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");
```

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 *Customizing and building your own runtime library*, page 108.

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).

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

MULTITHREAD SUPPORT IN THE DLIB RUNTIME ENVIRONMENT

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

● The heap (in other words when `malloc`, `new`, `free`, `delete`, `realloc`, or `calloc` is used).

● The 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 18: 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 the DLIB runtime environment with multithread support, the compiler option `--guard_calls` must be used to make sure that function-static variables with dynamic initializers are not initialized simultaneously by several threads.

ENABLING MULTITHREAD SUPPORT

There are no prebuilt runtime libraries with support for multithreading, which means you must first customize the library by adding the line `#define _DLIB_THREAD_SUPPORT 3` in the library configuration file and rebuild your library. See also *Customizing and building your own runtime library*, page 108.

To configure multithread support for use with threaded applications:

1 To enable multithread support:

On the command line, use the linker option `--threaded_lib` and use your own customized library.

In the IDE, choose **Project>Options>Linker>Extra Options** and specify the linker option `--threaded_lib`. If you are using C++, you must also choose **Project>Options>C/C++ Compiler>Extra Options** and specify the compiler option `--guard_calls` to make sure that function-static variables with dynamic initializers are not initialized simultaneously by several threads.

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

- Implement code for the library's system locks interface, see *System locks interface*, page 138.
- 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 `-e`), see *File streams locks interface*, page 138.
- Implement code that handles thread creation, thread destruction, and TLS access methods for the library, see *TLS handling*, page 138.

See also *Lock usage*, page 138 for information about used file streams locks and system locks.

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

3 Build your project.

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

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.

Lock usage

The number of locks that the DLIB runtime environment 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 runtime environment will initialize and destroy all the locks above.

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

TLS handling

The DLIB runtime environment 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 segment
- 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.
`__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 19: Macros for implementing TLS allocation

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

This is an example of how you can handle threads:

```c
#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.

The CLIB runtime environment

- Using a prebuilt runtime library

- Input and output

- System startup and termination

- Overriding default library modules

- Customizing system initialization

- C-SPY emulated I/O

Note that the CLIB runtime environment does not support any C99 functionality. For example, complex numbers and variable length arrays are not supported. Neither does CLIB support C++.

The legacy CLIB runtime environment is provided for backward compatibility and should not be used for new application projects.

For information about migrating from CLIB to DLIB, see the *IAR Embedded Workbench® Migration Guide*.

Using a prebuilt runtime library

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

- Type of library
- Processor option (-v)
- Memory model option (--memory_model)
- AVR enhanced core option (--enhanced_core)
- Small flash memory option (--64k_flash)
- 64-bit doubles option (--64bit_doubles).

CHOOSING A RUNTIME LIBRARY

The IDE includes the correct runtime library based on the options you select. See the *IDE Project Management and Building Guide for AVR* for more information.

Specify which runtime library file to use on the XLINK command line, for instance:

cl*libname*.r90

The CLIB runtime environment includes the C standard library. The linker will include only those routines that are required—directly or indirectly—by your application. For more information about the runtime libraries, see the chapter *C/C++ standard library functions*.

RUNTIME LIBRARY FILENAME SYNTAX

The runtime library names are constructed in this way:

{*type*}{*cpu*}{*memModel*}-{*core*}-{*smallFlash*}-{*64BitDoubles*}.r90

where

- {*type*} cl for the CLIB runtime library
- {*cpu*} is a value from 0 to 6, matching the -v option
- {*memModel*} is one of t, s, l, or h for the Tiny, Small, Large, or Huge memory model, respectively
- {*smallFlash*} is sf when the small flash memory is available. When small flash memory is not available, this value is not specified
- {*64BitDoubles*} is 64 when 64-bit doubles are used. When 32-bit doubles are used, this value is not specified.

These are some examples of how to decode a library name:

Library file	Generic processor option	Memory model	Core	Small flash	64-bit doubles
cl0t.r90	-v0	Tiny	--	--	--
cl1s-64.r90	-v1	Small	--	--	X
cl61-ec_mul-64.r90	-v6	Large	X	--	X

Table 20: Runtime libraries

Input and output

You can customize:

- The functions related to character-based I/O

● The formatters used by `printf`/`sprintf` and `scanf`/`sscanf`.

CHARACTER-BASED I/O

The functions `putchar` and `getchar` are the fundamental C functions for character-based I/O. For any character-based I/O to be available, you must provide definitions for these two functions, using whatever facilities the hardware environment provides.

The creation of new I/O routines is based on these files:

● `putchar.c`, which serves as the low-level part of functions such as `printf`

● `getchar.c`, which serves as the low-level part of functions such as `scanf`.

The code example below shows how memory-mapped I/O could be used to write to a memory-mapped I/O device:

```
__no_init volatile unsigned char devIO @ 40;

int putchar(int outChar)
{
  devIO = outChar;
  return outChar;
}
```

The exact address is a design decision. For example, it can depend on the selected processor variant.

For information about how to include your own modified version of `putchar` and `getchar` in your project build process, see *Overriding library modules*, page 107.

FORMATTERS USED BY PRINTF AND SPRINTF

The `printf` and `sprintf` functions use a common formatter, called `_formatted_write`. There are three variants of the formatter:

```
_large_write
_medium_write
_small_write
```

By default, the linker automatically uses the most appropriate formatter for your application.

_large_write

The `_large_write` formatter supports the C89 `printf` format directives.

_medium_write

The _medium_write formatter has the same format directives as _large_write, except that floating-point numbers are not supported. Any attempt to use a %f, %g, %G, %e, or %E specifier will produce a runtime error:

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

_medium_write is considerably smaller than the large version.

_small_write

The _small_write formatter works in the same way as _medium_write, except that it supports only the %%, %d, %o, %c, %s, and %x specifiers for integer objects, and does not support field width or precision arguments. The size of _small_write is 10–15% that of _medium_write.

Specifying the printf formatter in the IDE

1 Choose **Project>Options** and select the **General Options** category. Click the **Library Options** tab.

2 Choose the appropriate **Printf formatter** option, which can be either **Auto**, **Small**, **Medium**, or **Large**.

Specifying the printf formatter from the command line

To explicitly specify and override the formatter used by default, add one of the following lines to the linker configuration file:

```
-e_small_write=_formatted_write
-e_medium_write=_formatted_write
-e_large_write=_formatted_write
```

Customizing printf

For many embedded applications, sprintf is not required, and even printf with _small_write provides more facilities than are justified, considering the amount of memory it consumes. Alternatively, a custom output routine might be required to support particular formatting needs or non-standard output devices.

For such applications, a much reduced version of the printf function (without sprintf) is supplied in source form in the file intwri.c. This file can be modified to meet your requirements, and the compiled module inserted into the library in place of the original file; see *Overriding library modules*, page 107.

FORMATTERS USED BY SCANF AND SSCANF

As with the `printf` and `sprintf` functions, `scanf` and `sscanf` use a common formatter, called `_formatted_read`. There are two variants of the formatter:

```
_large_read
_medium_read
```

By default, the linker automatically uses the most appropriate formatter for your application.

_large_read

The `_large_read` formatter supports the C89 `scanf` format directives.

_medium_read

The `_medium_read` formatter has the same format directives as the large version, except that floating-point numbers are not supported. `_medium_read` is considerably smaller than the large version.

Specifying the scanf formatter in the IDE

1 Choose **Project>Options** and select the **General Options** category. Click the **Library Options** tab.

2 Choose the appropriate **Scanf formatter** option, which can be either **Auto**, **Medium** or **Large**.

Specifying the read formatter from the command line

To explicitly specify and override the formatter used by default, add one of the following lines to the linker configuration file:

```
-e_medium_read=_formatted_read
-e_large_read=_formatted_read
```

System startup and termination

This section describes the actions the runtime environment performs during startup and termination of applications.

The code for handling startup and termination is located in the source files `cstartup.s90`, `_exit.s90` and `low_level_init.c` located in the `avr\src\lib` directory.

Note: Normally, you do not need to customize either of the files `cstartup.s90` or `_exit.s90`.

SYSTEM STARTUP

When an application is initialized, several steps are performed:

● The custom function `__low_level_init` is called, giving the application a chance to perform early initializations

● The external data and address buses are enabled if needed

● The stack pointers are initialized to the end of CSTACK and RSTACK, respectively

● Static variables are initialized except for `__no_init __tinyflash`, `__flash`, `__farflash`, `__hugeflash`, and `__eeprom` declared variables; this includes clearing zero-initialized memory and copying the ROM image of the RAM memory of the remaining initialized variables

● The `main` function is called, which starts the application.

Note that the system startup code contains code for more steps than described here. The other steps are applicable to the DLIB runtime environment.

SYSTEM TERMINATION

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 cstartup code calls the `exit` function if `main` returns. The parameter passed to the `exit` function is the return value of `main`. The default `exit` function is written in assembler.

When the application is built in debug mode, C-SPY stops when it reaches the special code label `?C_EXIT`.

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

Overriding default library modules

The IAR CLIB Library contains modules which you probably need to override with your own customized modules, for example for character-based I/O, without rebuilding the entire library. For information about how to override default library modules, see *Overriding library modules*, page 107, in the chapter *The DLIB runtime environment*.

Customizing system initialization

For information about how to customize system initialization, see *System initialization*, page 122.

C-SPY emulated I/O

The low-level I/O interface is used for communication between the application being debugged and the debugger itself. The interface is simple: C-SPY will place breakpoints on certain assembler labels in the application. When code located at the special labels is about to be executed, C-SPY will be notified and can perform an action.

THE DEBUGGER TERMINAL I/O WINDOW

When code at the labels ?C_PUTCHAR and ?C_GETCHAR is executed, data will be sent to or read from the debugger window.

For the ?C_PUTCHAR routine, one character is taken from the output stream and written. If everything goes well, the character itself is returned, otherwise -1 is returned.

When the label ?C_GETCHAR is reached, C-SPY returns the next character in the input field. If no input is given, C-SPY waits until the user types some input and presses the Return key.

To make the **Terminal I/O** window available, the application must be linked with the XLINK option **With I/O emulation modules** selected. See the *IDE Project Management and Building Guide for AVR*.

TERMINATION

The debugger stops executing when it reaches the special label ?C_EXIT.

Assembler language interface

- Mixing C and assembler

- Calling assembler routines from C

- Calling assembler routines from C++

- Calling convention

- Call frame information

Mixing C and assembler

The IAR C/C++ Compiler for AVR 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 gives several benefits compared to using inline assembler:

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

This causes some overhead in the form of a function call and return instruction sequences, and the compiler will regard some registers as scratch registers. However, the compiler will also assume that all scratch registers are destroyed when calling a function written in assembler. In many cases, the overhead of the extra instructions is compensated by the work of 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 155.

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 165.

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 154, respectively.

INLINE ASSEMBLER

Inline assembler can be used for inserting assembler instructions directly into a C or C++ function.

The asm extended keyword and its alias __asm both insert 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.

Note: Not all assembler directives or operators can be inserted using these keywords.

The syntax is:

```
asm ("string");
```

The string can be a valid assembler instruction or a data definition assembler directive, but not a comment. You can write several consecutive inline assembler instructions, for example:

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

where \n (new line) separates each new assembler instruction. Note that you can define and use local labels in inline assembler instructions.

The following example demonstrates the use of the asm keyword. This example also shows the risks of using inline assembler.

```
bool flag;

void Func()
{
  while (!flag)
  {
    asm("IN R0, PIND \n"
        "STS flag, R0");
  }
}
```

Note: Because using symbols from inside the inline assembler code is not properly visible to all parts of the compiler, you must use #pragma required when you reference an external or module-local symbol only from inline assembler code. If you do not, you can get an undefined symbol error when compiling. See *required*, page 334.

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

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

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

● The compiler's various optimizations will disregard any effects of the inline sequences, which will not be optimized at all

● In general, assembler directives will cause errors or have no meaning. Data definition directives will however work as expected

● Auto variables cannot be accessed.

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

Calling assembler routines from C

An assembler routine that will be called from C must:

● Conform to the calling convention

● Have a PUBLIC entry-point label

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

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

or

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

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

CREATING SKELETON CODE

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

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

```
extern int gInt;
extern char gChar;

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

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

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

COMPILING THE 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:

```
iccavr 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 `s90`. 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.s90`.

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 165.

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);
  }
};
```

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

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

Calling convention

A calling convention is the way a function in a program calls another function. The compiler handles this automatically, but, if a function is written in assembler language,

you must know where and how its parameters can be found, how to return to the program location from where it was called, and how to return the resulting value.

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

The compiler provides two calling conventions—one old, which is used in version 1.x of the compiler, and one new which is the default. This section describes the calling conventions used by the compiler. These items are examined:

● Choosing a calling convention

● Function declarations

● C and C++ linkage

● Preserved versus scratch registers

● Function entrance

● Function exit

● Return address handling

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

CHOOSING A CALLING CONVENTION

You can choose between these calling conventions:

● The old calling convention offers a simple assembler interface. It is compatible with the calling convention used in version 1.x of the compiler. Even though this convention is not used by default, it is recommended for use when mixing C and assembler code

● The new calling convention is default. It is more efficient than the old calling convention, but also more complex to understand and subject to change in later versions of the compiler.

To choose another calling convention than the default, use the `--version1_calls` command line option. You can also declare individual functions to use the old calling convention by using the `__version1` function attribute, for example:

```
extern
__version_1 void doit(int arg);
```

For details about the `--version1_calls` option and the `__version1` attribute, see *--version1_calls*, page 276 and *__version_1*, page 315, respectively.

In the IDE, choose **Use ICCA90 1.x calling convention** on the **Project>C/C++ Compiler>Code** page.

Hints for using the default calling convention

The default calling convention is very complex, and therefore not recommended for use when calling assembler routines. However, if you intend to use it for your assembler routines, you should create a list file and see how the compiler assigns the different parameters to the available registers. For an example, see *Creating skeleton code*, page 153.

If you intend to use the default calling convention, you should *also* specify a value to the runtime model attribute `__rt_version` using the RTMODEL assembler directive:

```
RTMODEL "__rt_version", "value"
```

The parameter *value* should have the same value as the one used internally by the compiler. For information about what value to use, see the generated list file. If the calling convention changes in future compiler versions, the runtime model value used internally by the compiler will also change. Using this method gives a module consistency check, because the linker produces errors for mismatches between the values.

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 AVR 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.

For both calling conventions, these 14 registers can be used as scratch registers by a function:

R0-R3, R16-R23, and R30-R31

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.

For both calling conventions, these registers are preserved registers:

R4-R15 and R24-R27

Note that the registers R4-R15 can be locked from the command line and used for global register variables; see *--lock_regs*, page 258 and *__regvar*, page 312.

Special registers

For some registers, you must consider certain prerequisites:

- The stack pointer—register Y—must at all times point to the last element on the stack. In the eventuality of an interrupt, everything below the point the stack pointer points to, will be destroyed.

- If using the -v4 or -v6 processor option, the RAMPY register is part of the data stack pointer.

- If using a processor option which uses any of the registers EIND, RAMPX, or RAMPZ, these registers are treated as scratch registers.

FUNCTION ENTRANCE

During a function call, the calling function:

- passes the parameters, either in registers or on the stack

- pushes any other parameters on the data stack (CSTACK)

Control is then passed to the called function with the return address being automatically pushed on the return address stack (RSTACK).

The called function:

- stores any local registers required by the function on the data stack

- allocates space for its auto variables and temporary values

- proceeds to run the function itself.

Register parameters versus stack parameters

Parameters can be passed to a function using one of two basic methods: in registers or on the stack. It is much more efficient to use registers than to take a detour via memory, so the calling convention is designed to use registers as much as possible. Only a limited number of registers can be used for passing parameters; when no more registers are available, the remaining parameters are passed on the stack.

Hidden parameters

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

- A function returning structures or unions larger than 4 bytes gets an extra hidden parameter, which is a default pointer—depending on the used memory model— pointing to the location where the result should be stored. This pointer must be returned to the callee.

● For non-static C++ member functions, the `this` pointer is passed as the first parameter (but placed after the return structure pointer, if there is one). Note that static member functions do not have a `this` pointer.

Register parameters

For both calling conventions, the registers available for passing parameters are:

`R16–R23`

Parameters are allocated to registers using a first-fit algorithm, using parameter alignment requirements according to this table:

Parameters	Alignment	Passed in registers
8-bit values	I	R16, R17, R18, R19, R20, R21, R22, R23
16-bit values	2	R17:R16, R19:R18, R21:R20, R23:R22
24-bit values	4	R18:R17:R16, R22:R21:R20
32-bit values	4	R19:R18:R17:R16, R23:R22:R21:R20
64-bit values	8	R23:R22:R21:R20:R19:R18:R17:R16

Table 21: Registers used for passing parameters

Register assignment using the default calling convention

In the default calling convention, as many parameters as possible are passed in registers. The remaining parameters are passed on the stack. The compiler may change the order of the parameters in order to achieve the most efficient register usage.

The algorithm for assigning parameters to registers is quite complex in the default calling convention. For details, you should create a list file and see how the compiler assigns the different parameters to the available registers, see *Creating skeleton code*, page 153.

Below follows some examples of combinations of register assignment.

A function with the following signature (not C++):

`void Func(char __far * a, int b, char c, int d)`

would have a allocated to `R18:R17:R16`, b to `R21:R20` (alignment requirement prevents `R20:R19`), c to `R19` (first fit), and d to `R23:R22` (first fit).

Another example:

`void bar(char a, int b, long c, char d)`

This would result in a being allocated to `R16` (first fit), b to `R19:R18` (alignment), c to `R23:R22:R21:R20` (first fit), and d to `R17` (first fit).

A third example:

```
void baz(char a, char __far * b, int c, int d)
```

This would give, a being allocated to R16, b to R22:R21:R20, c to R19:R18, and d to the stack.

Register assignment using the old calling convention

In the old calling convention, the two left-most parameters are passed in registers if they are scalar and up to 32 bits in size.

This table shows some of the possible combinations:

Parameters*	Parameter 1	Parameter 2
f(b1,b2,…)	R16	R20
f(b1,w2,…)	R16	R20, R21
f(w1,l1,…)	R16, R17	R20, R21, R22, R23
f(l1,b2,…)	R16, R17, R18, R19	R20
f(l1,l2,…)	R16, R17, R18, R19	R20, R21, R22, R23

Table 22: Passing parameters in registers

* Where b denotes an 8-bit data type, w denotes a 16-bit data type, and l denotes a 32-bit data type. If the first and/or second parameter is a 3-byte pointer, it will be passed in R16–R19 or R20–R22 respectively.

Stack parameters and layout

A function call creates a stack frame as follows:

High
address

Stack parameters	
Saved registers	
Auto variables	
Temporary storage	← Y

Low
address
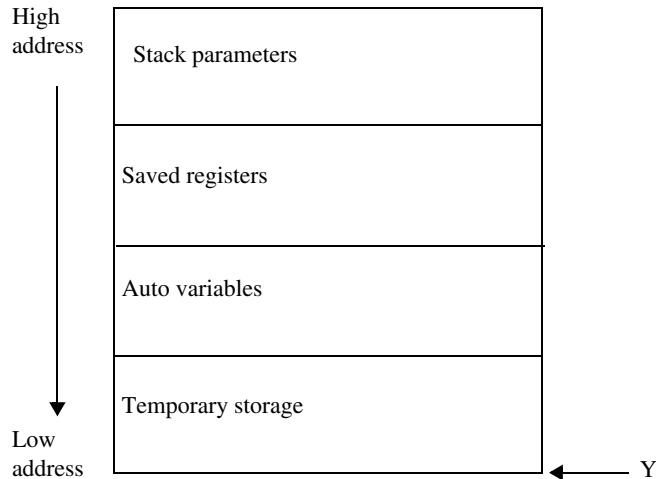
Note that only registers that are used will be saved.

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

For the old calling convention, the registers available for returning values are `R16-R19`.
The default calling convention can use the registers `R16-R23` for returning values.

Return values	Passed in registers
8-bit values	R16
16-bit values	R17:R16
24-bit values	R18:R17:R16
32-bit values	R19:R18:R17:R16
64-bit values	R23:R22:R21:R20:R19:R18:R17:R16

Table 23: Registers used for returning values

Note that the size of a returned pointer depends on the memory model in use; appropriate registers are used accordingly.

Stack layout at function exit

Normally, it is the responsibility of the called function to clean the stack. The only exception is for ellipse functions—functions with a variable argument list such as `printf`—for which it is the responsibility of the caller to clean the stack.

Return address handling

A function written in assembler language should, when finished, return to the caller. At a function call, the return address is automatically stored on the RSTACK (not the CSTACK).

Typically, a function returns by using the `RET` instruction.

RESTRICTIONS FOR SPECIAL FUNCTION TYPES

Interrupt functions

Interrupt functions differ from ordinary C functions in that:

- If used, flags and scratch registers are saved
- Calls to interrupt functions are made via interrupt vectors; direct calls are not allowed
- No arguments can be passed to an interrupt function
- Interrupt functions return by using the `RETI` function.

For more information about interrupt functions, see *Interrupt functions*, page 64.

Monitor functions

A monitor function causes interrupts to be disabled during execution of the function. At function entry, the status register `SREG` is saved and global interrupts are disabled. At function exit, the global interrupt enable bit (I) is restored in the `SREG` register, and thereby the interrupt status existing before the function call is also restored.

For more information about monitor functions, see *Monitor functions*, page 65.

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 R17:R16, and the return value is passed back to its caller in the register R17:R16.

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

```
        subi    R16,FF
        sbci    R17,FF
        ret
```

Example 2

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

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

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

The calling function must reserve 16 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 R17:R16. The return value is passed back to its caller in the register R17:R16.

Example 3

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

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

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

It is the responsibility of the calling function to allocate a memory location for the return value and pass a pointer to it as a hidden first parameter. The pointer to the location where the return value should be stored is passed in the first suitable register/register pair, which is R16, R17:R16, and R18:R17:R16 for the Tiny, Small, Large, and Huge memory model, respectively. The parameter x is passed in R19:R18, R19:R18, and R21:R20 for the Tiny, Small, Large, and Huge memory model, respectively.

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 `R17:R16`, and the return value is returned in `R16`, `R17:R16`, and `R18:R17:R16` for the Tiny, Small, and Large memory model, respectively.

FUNCTION DIRECTIVES

Note: This type of directive is primarily intended to support static overlay, a feature which is useful in some smaller microcontrollers. The IAR C/C++ Compiler for AVR does not use static overlay, because it has no use for it.

The function directives `FUNCTION`, `ARGFRAME`, `LOCFRAME`, and `FUNCALL` are generated by the compiler to pass information about functions and function calls to the IAR XLINK Linker. These directives can be seen if you use the compiler option **Assembler file** (`-lA`) to create an assembler list file.

For more information about the function directives, see the *IAR Assembler User Guide for AVR*.

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 AVR*.

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.

For AVR, the defined resources depend on which processor option you are using. You can find the resources in the list file section CFI Names.

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.

Using C

- C language overview

- Extensions overview

- IAR C language extensions

C language overview

The IAR C/C++ Compiler for AVR 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 151.

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

Note: CLIB does not support any C99 functionality. For example, complex numbers and variable length arrays are not supported.

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 169. 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 149. For information about available functions, see the chapter *Intrinsic functions*.

● Library functions

The DLIB runtime environment provides the C and C++ library definitions in the C/C++ standard library that apply to embedded systems. For more information, see *DLIB runtime environment—implementation details*, page 363. For information about AVR-specific library functions, see *AVR-specific library functions*, page 370.

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 169.
`-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 1>Language conformance** and select the appropriate option. Note that language extensions are enabled by default.

IAR C language extensions

The compiler provides a wide set of C language extensions. To help you to find the extensions required by your application, they are grouped like this in this section:

● *Extensions for embedded systems programming*—extensions specifically tailored for efficient embedded programming for the specific 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 172.

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 segment

    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 segment. For more information about using these features, see *Controlling data and function placement in memory*, page 210, and *location*, page 331.

- Alignment control

    Each data type has its own alignment; for more information, see *Alignment*, page 281. If you want to change the alignment the #pragma data_alignment directive 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 208.

- 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 284.

- 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 segment operators

The compiler supports getting the start address, end address, and size for a segment with these built-in segment operators:

`__segment_begin`	Returns the address of the first byte of the named segment.
`__segment_end`	Returns the address of the first byte *after* the named segment.
`__segment_size`	Returns the size of the named segment 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 segments defined in the linker configuration file.

These operators behave syntactically as if declared like:

```
void * __segment_begin(char const * segment)
void * __segment_end(char const * segment)
size_t __segment_size(char const * segment)
```

When you use the `@` operator or the `#pragma location` directive to place a data object or a function in a user-defined segment in the linker configuration file, the segment operators can be used for getting the start and end address of the memory range where the segments were placed.

The named *segment* must be a string literal and it must have been declared earlier with the `#pragma segment` directive. If the segment was declared with a memory attribute *memattr*, the type of the `__segment_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.

Example

In this example, the type of the `__segment_begin` operator is `void __near *`.

```
#pragma segment="MYSEGMENT" __near
...
segment_start_address = __segment_begin("MYSEGMENT");
```

See also *segment*, page 336, 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 289.

● 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 250.

● 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 250.

To take advantage of *Extended* Embedded C++ features in your source code, use the --eec++ compiler option. See --*eec++*, page 251.

To enable EC++ or EEC++ in the IDE, choose **Project>Options>C/C++ Compiler>Language 1** and select the appropriate standard.

EC++ feature descriptions

When you write C++ source code for the IAR C/C++ Compiler for AVR, 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 __near memory at address 60
  static __near __no_init int mI @ 60;

  // Locate a static function in __nearfunc memory
  static __nearfunc void F();

  // Locate a function in __nearfunc memory
  __nearfunc void G();

  // Locate a virtual function in __nearfunc memory
  virtual __nearfunc void H();

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

The this pointer

The this pointer used for referring to a class object or calling a member function of a class object will by default have the data memory attribute for the default data pointer type. This means that such a class object can only be defined to reside in memory from which pointers can be implicitly converted to a default data pointer. This restriction might also apply to objects residing on a stack, for example temporary objects and auto objects.

Class memory

To compensate for this limitation, a class can be associated with a *class memory type*. The class memory type changes:

- the this pointer type in all member functions, constructors, and destructors into a pointer to class memory
- the default memory for static storage duration variables—that is, not auto variables—of the class type, into the specified class memory
- the pointer type used for pointing to objects of the class type, into a pointer to class memory.

Example

```
class __far C
{
public:
  void MyF();        // Has a this pointer of type C __far *
  void MyF() const;  // Has a this pointer of type
                     // C __far const *
  C();               // Has a this pointer pointing into far
                     // memory
  C(C const &);      // Takes a parameter of type C __far
                     // const & (also true of generated copy
                     // constructor)
  int mI;
};

C Ca;              // Resides in far memory instead of the
                   // default memory
C __near Cb;       // Resides in near memory, the 'this'
                   // pointer still points into far memory

void MyH()
{
  C cd;            // Resides on the stack
}

C *Cp1;            // Creates a pointer to far memory
C __near *Cp2;     // Creates a pointer to near memory
```

Note: To place the C class in huge memory is not allowed, unless using the huge memory model, because a huge pointer cannot be implicitly converted into a __far pointer.

Whenever a class type associated with a class memory type, like C, must be declared, the class memory type must be mentioned as well:

```
class __far C;
```

Also note that class types associated with different class memories are not compatible types.

A built-in operator returns the class memory type associated with a class, __memory_of(*class*). For instance, __memory_of(C) returns __far.

When inheriting, the rule is that it must be possible to convert implicitly a pointer to a subclass into a pointer to its base class. This means that a subclass can have a *more* restrictive class memory than its base class, but not a *less* restrictive class memory.

```
class __far D : public C
{ // OK, same class memory
public:
  void MyG();
  int mJ;
};

class __near E : public C
{ // OK, near memory is inside far
public:
  void MyG() // Has a this pointer pointing into near memory
  {
    MyF();    // Gets a this pointer into far memory
  }
  int mJ;
};

class F : public C
{ // OK, will be associated with same class memory as C
public:
  void MyG();
  int mJ;
};
```

A `new` expression on the class will allocate memory in the heap associated with the class memory. A `delete` expression will naturally deallocate the memory back to the same heap. To override the default `new` and `delete` operator for a class, declare

```
void *operator new(size_t);
void operator delete(void *);
```

as member functions, just like in ordinary C++.

For more information about memory types, see *Memory types*, page 54.

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
}
```

NEW AND DELETE OPERATORS

There are operators for `new` and `delete` for each memory that can have a heap, that is, tiny, near, far, and huge memory.

```
// Assumes that there is a heap in both __near and __far memory
void __near *operator new __near(__near_size_t);
void __far  *operator new __far (__far_size_t);
void operator delete(void __near *);
void operator delete(void __far  *);

// And correspondingly for array new and delete operators
void __near *operator new[] __near(__near_size_t);
void __far  *operator new[] __far (__far_size_t);
void operator delete[](void __near *);
void operator delete[](void __far  *);
```

Use this syntax if you want to override both global and class-specific `operator new` and `operator delete` for any data memory.

Note that there is a special syntax to name the `operator new` functions for each memory, while the naming for the `operator delete` functions relies on normal overloading.

New and delete expressions

A new expression calls the operator new function for the memory of the type given. If a class, struct, or union type with a class memory is used, the class memory will determine the operator new function called. For example,

```
void MyF()
{
  // Calls operator new __huge(__huge_size_t)
  int __huge *p = new __huge int;

  // Calls operator new __near(__near_size_t)
  int __near *q = new int __near;

  // Calls operator new[] __near(__near_size_t)
  int __near *r = new __near int[10];

  // Calls operator new __huge(__huge_size_t)
  class __huge S
  {
  };
  S *s = new S;

  // Calls operator delete(void __huge *)
  delete p;
  // Calls operator delete(void __near  *)
  delete s;

  int __huge *t = new __far int;
  delete t; // Error: Causes a corrupt heap
}
```

Note that the pointer used in a delete expression must have the correct type, that is, the same type as that returned by the new expression. If you use a pointer to the wrong memory, the result might be a corrupt heap.

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 118.

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 AVR*.

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 176.

Templates and data memory attributes

For data memory attributes to work as expected in templates, two elements of the standard C++ template handling were changed—class template partial specialization matching and function template parameter deduction.

In Extended Embedded C++, the class template partial specialization matching algorithm works like this:

When a pointer or reference type is matched against a pointer or reference to a template parameter type, the template parameter type will be the type pointed to, stripped of any data memory attributes, if the resulting pointer or reference type is the same.

Example

```
// We assume that __far is the memory type of the default
// pointer.
template<typename> class Z {};
template<typename T> class Z<T *> {};

Z<int __near  *> Zn;   // T = int __near
Z<int __far *> Zf;     // T = int
Z<int *> Zd;           // T = int
Z<int __huge *> Zh;    // T = int __huge
```

In Extended Embedded C++, the function template parameter deduction algorithm works like this:

When function template matching is performed and an argument is used for the deduction; if that argument is a pointer to a memory that can be implicitly converted to a default pointer, do the parameter deduction as if it was a default pointer.

When an argument is matched against a reference, do the deduction as if the argument and the parameter were both pointers.

Example

```
// We assume that __far is the memory type of the default
// pointer.
template<typename T> void fun(T *);

void MyF()
{
  fun((int __near *) 0);   // T = int. The result is different
                           // than the analogous situation with
                           // class template specializations.
  fun((int *) 0);          // T = int
  fun((int __far  *) 0);   // T = int
  fun((int __huge *) 0);   // T = int __huge
}
```

For templates that are matched using this modified algorithm, it is impossible to get automatic generation of special code for pointers to "small" memory types. For "large" and "other" memory types (memory that cannot be pointed to by a default pointer) it is possible. To make it possible to write templates that are fully memory-aware—in the rare cases where this is useful—use the `#pragma basic_template_matching` directive in front of the template function declaration. That template function will then match without the modifications described above.

Example

```
// We assume that __far is the memory type of the default
// pointer.
#pragma basic_template_matching
template<typename T> void fun(T *);

void MyF()
{
  fun((int __near *) 0); // T = int __near
}
```

Non-type template parameters

It is allowed to have a reference to a memory type as a template parameter, even if pointers to that memory type are not allowed.

Example

```
extern int __near X;

template<__near int &y>
void Func()
{
  y = 17;
}

void Bar()
{
  Foo<X>();
}
```

The containers in the STL, like `vector` and `map`, are memory attribute aware. This means that a container can be declared to reside in a specific memory type which has the following consequences:

● The container itself will reside in the chosen memory

● Allocations of elements in the container will use a heap for the chosen memory

● All references inside it use pointers to the chosen memory.

Example

```
#include <vector>

vector<int> D;                     // D placed in default
memory,
                                   // using the default heap,
                                   // uses default pointers
vector<int __near> __near X;       // X placed in near memory,
                                   // heap allocation from
                                   // near, uses pointers to
                                   // near memory
vector<int __huge> __near Y;       // Y placed in near memory,
                                   // heap allocation from
                                   // huge, uses pointers to
                                   // huge memory
```

Note that this is illegal:

```
vector<int __near> __huge Z;
```

Note also that `map<key, T>`, `multimap<key, T>`, `hash_map<key, T>`, and `hash_multimap<key, T>` all use the memory of `T`. This means that the `value_type` of these collections will be `pair<key, const T>` *mem* where *mem* is the memory type of `T`. Supplying a key with a memory type is not useful.

Example

Note that two containers that only differ by the data memory attribute they use cannot be assigned to each other. Instead, the templated assign member method must be used.

```
#include <vector>

vector<int __near> X;
vector<int __huge> Y;

void MyF()
{
  // The templated assign member method will work
  X.assign(Y.begin(), Y.end());
  Y.assign(X.begin(), X.end());
}
```

The standard template library

The containers in the STL, like `vector` and `map`, are memory attribute aware. This means that a container can be declared to reside in a specific memory type which has the following consequences:

- The container itself will reside in the chosen memory
- Allocations of elements in the container will use a heap for the chosen memory
- All references inside it use pointers to the chosen memory.

Example

```
#include <vector>

vector<int> D;                    // D placed in default
memory,
                                  // using the default heap,
                                  // uses default pointers
vector<int __near> __near X;      // X placed in near memory,
                                  // heap allocation from
                                  // near, uses pointers to
                                  // near memory
vector<int __far> __near Y;       // Y placed in near memory,
                                  // heap allocation from
                                  // far, uses pointers to
                                  // far memory
```

Note that this is illegal:

```
vector<int __near> __far Z;
```

Note also that `map<key, T>`, `multimap<key, T>`, `hash_map<key, T>`, and `hash_multimap<key, T>` all use the memory of `T`. This means that the `value_type` of these collections will be `pair<key, const T>` *mem* where *mem* is the memory type of `T`. Supplying a key with a memory type is not useful.

Example

Note that two containers that only differ by the data memory attribute they use cannot be assigned to each other. Instead, the templated assign member method must be used.

```
#include <vector>

vector<int __near> X;
vector<int __far> Y;

void MyF()
{
  // The templated assign member method will work
  X.assign(Y.begin(), Y.end());
  Y.assign(X.begin(), X.end());
}
```

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
};
```

- It is allowed to specify an array with no size or size 0 as the last member of a struct. For example:

```
typedef struct
{
  int i;
  char ir[0]; // Zero-length array
};

typedef struct
{
  int i;
  char ir[];  // Zero-length array
};
```

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

Application-related considerations

- Stack considerations

- Heap considerations

- Interaction between the tools and your application

- Checksum calculation for verifying image integrity

Stack considerations

To make your application use stack memory efficiently, there are some considerations to be made.

AVAILABLE STACKS

The compiler uses the internal data stack, CSTACK, for a variety of user program operations. In addition, there is a specific stack for return addresses, RSTACK.

The return address stack

The return address stack is used for storing the return address when a CALL, RCALL, ICALL, or EICALL instruction is executed. Each call will use two or three bytes of return address stack. An interrupt will also place a return address on this stack.

The data segment used for holding the return address stack is called RSTACK.

To determine the size of the return address stack, see *Setting up stack memory*, page 91. Notice however that if the cross-call optimization has been used (-z9 without --no_cross_call), the value can be off by as much as a factor of six depending on how many times the cross-call optimizer has been run (--cross_call_passes). Each cross-call pass adds one level of calls, for example, two cross-call passes might result in a tripled stack usage.

If external SRAM is available, it is possible to place the stack there. However, the external memory is slower than the internal memory so moving the stacks to external memory will normally decrease the system performance; see *--enable_external_bus*, page 252.

Allocating a memory area for the stack is done differently using the command line interface compared to when using the IDE.

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 91, and *Saving stack space and RAM memory*, page 219.

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 segments used for the heap

● Allocating the heap size, see *Setting up heap memory*, page 92.

HEAP SEGMENTS IN DLIB

To access a heap in a specific memory, use the appropriate memory attribute as a prefix to the standard functions `malloc`, `free`, `calloc`, and `realloc`, for example:

`__near_malloc`

If you use any of the standard functions without a prefix, the function will be mapped to the default memory type near.

Each heap will reside in a segment with the name _HEAP prefixed by a memory attribute, for example NEAR_HEAP.

For information about available heaps, see *Dynamic memory on the heap*, page 61.

HEAP SEGMENTS IN CLIB

The memory allocated to the heap is placed in the segment 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 AVR 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.

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 -D. 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.
- Using the compiler operators __segment_begin, __segment_end, or __segment_size, or the assembler operators SFB, SFE, or SIZEOF on a named segment. These operators provide access to the start address, end address, and size of a contiguous sequence of segments with the same name
- The command line option -s 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:

```
-DNrOfElements=10
-DHeapSize=100
```

The linker configuration file can look like this:

```
-Z(DATA)MyHeap+HeapSize=20000-2FFFF
```

Add these lines to your application source code:

```
#include <stdlib.h>

/* Use symbol defined by an XLINK 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 an XLINK option.
 */
extern char HeapSize;

/* Declare the section that contains the heap. */
#pragma segment = "MYHEAP"

char *MyHeap()
{
  /* First get start of statically allocated section, */
  char *p = __segment_begin("MYHEAP");

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

Checksum calculation for verifying image integrity

This section contains information about checksum calculation:

- *Briefly about checksum calculation*, page 197
- *Calculating and verifying a checksum*, page 198
- *Troubleshooting checksum calculation*, page 203

BRIEFLY ABOUT CHECKSUM CALCULATION

You can use a checksum to verify that the image is the same at runtime as when the image's original checksum was generated. In other words, to verify that the image has not been corrupted.

This works as follows:

- You need an initial checksum.

  You can either use the linker to generate an initial checksum or you might have a third-party checksum available.

- You must generate a second checksum during runtime.

  You can either add specific code to your application source code for calculating a checksum during runtime or you can use some dedicated hardware on your device for calculating a checksum during runtime.

- You must add specific code to your application source code for comparing the two checksums and take an appropriate action if they differ.

  If the two checksums have been calculated in the same way, and if there are no errors in the image, the checksums should be identical. If not, you should first suspect that the two checksums were not generated in the same way.

No matter which solutions you use for generating the two checksum, you must make sure that both checksums are calculated *in the exact same way*. If you use the linker for the initial checksum and use a software-based calculation during runtime, you have full control of the generation for both checksums. However, if you are using a third-party checksum for the initial checksum or some hardware support for the checksum calculation during runtime, there might be additional requirements that you must consider.

For the two checksums, there are some choices that you must always consider and there are some choices to make only if there are additional requirements. Still, all of the details must be the same for both checksums.

Details always to consider:

- *Checksum range*

  The memory range (or ranges) that you want to verify by means of checksums. Typically, you might want to calculate a checksum for all ROM memory. However, you might want to calculate a checksum only for specific ranges. Remember that:

  - It is OK to have several ranges for one checksum.
  - Typically, the checksum must be calculated from the lowest to the highest address for every memory range.
  - Each memory range must be verified in the same order as defined (for example, `0x100-0x1FF,0x400-0x4FF` is not the same as `0x400-0x4FF,0x100-0x1FF`).

- If several checksums are used, you should place them in sections with unique names and use unique symbol names.
- A checksum should never be calculated on a memory range that contains a checksum or a software breakpoint.

- *Algorithm and size of checksum*

  You should consider which algorithm is most suitable in your case. There are two basic choices, Sum (a simple arithmetic algorithm) or CRC (which is the most commonly used algorithm). For CRC there are different sizes to choose for the checksum, 2 or 4 bytes where the predefined polynomials are wide enough to suit the size, for more error detecting power. The predefined polynomials work well for most, but possibly not for all data sets. If not, you can specify your own polynomial. If you just want a decent error detecting mechanism, use the predefined CRC algorithm for your checksum size, typically CRC16 or CRC32.

  Note that for an n-bit polynomial, the n:th bit is always considered to be set. For a 16-bit polynomial (for example, CRC16) this means that `0x11021` is the same as `0x1021`.

  For more information about selecting an appropriate polynomial for data sets with non-uniform distribution, see for example section 3.5.3 in *Tannenbaum, A.S., Computer Networks, Prentice Hall 1981, ISBN: 0131646990.*

- *Fill*

  Every byte in the checksum range must have a well-defined value before the checksum can be calculated. Typically, bytes with unknown values are *pad bytes* that have been added for alignment. This means that you must specify which fill pattern to be used during calculation, typically `0xFF` or `0x00`.

- *Initial value*

  The checksum must always have an explicit initial value.

In addition to these mandatory details, there might be other details to consider. Typically, this might happen when you have a third-party checksum, you want the checksum be compliant with the Rocksoft™ checksum model, or when you use hardware support for generating a checksum during runtime. The linker provides support for also controlling alignment, complement, bit order, and checksum unit size.

CALCULATING AND VERIFYING A CHECKSUM

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

I The CHECKSUM segment will only be included in your application if the segment appears to be needed. If the checksum is not needed by the application itself, use the linker option `-g__checksum` to force the segment to be included.

2 When configuring the linker to calculate a checksum, there are some basic choices to make:

- Checksum algorithm

  Choose which checksum algorithm you want to use. In this example, the CRC16 algorithm is used.

- Memory range

  Using the IDE, the checksum will by default be calculated for all placement directives (specified in the linker configuration file) for ROM-based memory. From the command line, you can specify any ranges.

- Fill pattern

  Specify a fill pattern—typically `0xFF` or `0x00`—for bytes with unknown values. The fill pattern will be used in all checksum ranges.

For more information, see *Briefly about checksum calculation*, page 197.

To run the linker from the IDE, choose **Project>Options>Linker>Checksum** and make your settings, for example:

In the simplest case, you can ignore (or leave with default settings) these options: **Complement**, **Bit order**, and **Checksum unit size**.

To make the linker create a checksum from the command line, use the `-J` linker option, for example like this:

```
-J2,crc16,,__checksum,CHECKSUM,1=0x8002-0x8FFF
```

The checksum will be created when you build your project and will be placed in the automatically generated segment `CHECKSUM`. If you are using your own linker configuration file or if you explicitly want to control the placement of the `CHECKSUM` segment, you must update your linker configuration file with placement information

accordingly. In that case, make sure to place the segment so that it is not part of the application's checksum calculation.

3 You can specify several ranges instead of only one range.

If you are using the IDE, perform these steps:

● Choose **Project>Options>Linker>Checksum** and make sure to deselect **Fill unused code memory**.

● Choose **Project>Options>Linker>Extra Options** and specify the ranges, for example like this:

```
-h(CODE)0-3FF,8002-8FFF
-J2,crc16,,,1=0-3FF,8002-8FFF
```
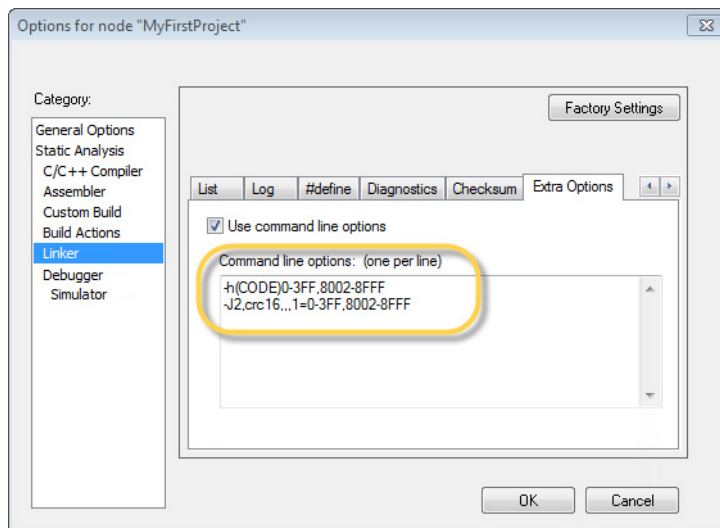

If you are using the command line, use the -J option and specify the ranges. for example like this:

```
-h(CODE)0-3FF,8002-8FFF
-J2,crc16,,,1=0-3FF,8002-8FFF
```

4 Add a function for checksum calculation to your source code. Make sure that the function uses the same algorithm and settings as for the checksum calculated by the linker. For example, a slow variant of the crc16 algorithm but with small memory footprint (in contrast to the fast variant that uses more memory):

```
unsigned short SmallCrc16(uint16_t
 sum,
                            unsigned char *p,
                            unsigned int len)
{
  while (len--)
  {
    int i;
    unsigned char byte = *(p++);

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

You can find the source code for this checksum algorithm in the `avr\src\linker` directory of your product installation.

5 Make sure that your application also contains a call to the function that calculates the checksum, compares the two checksums, and takes appropriate action if the checksum values do not match. This code gives an example of how the checksum can be calculated for your application and to be compared with the linker generated checksum:

```
/* The calculated checksum */

/* Linker generated symbols */
extern unsigned short const __checksum;
extern int __checksum_begin;
extern int __checksum_end;


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

  /* Run the checksum algorithm */
  calc = SmallCrc16(0,
                    (unsigned char *) &__checksum_begin,
                    ((unsigned char *) &__checksum_end -
                    ((unsigned char *) &__checksum_begin)+1));

  /* Fill the end of the byte sequence with zeros. */
  calc = SmallCrc16(calc, zeros, 2);

  /* Test the checksum */
  if (calc != __checksum)
  {
    printf("Incorrect checksum!\n");
    abort();    /* Failure */
  }

  /* Checksum is correct */
}
```

Note: Make sure to define the symbols `__checksum_begin` and `__checksum_end` in your linker configuration file.

6 Build your application project and download it.

During the build, the linker creates a checksum and places it in the specified symbol `__checksum` in the segment CHECKSUM.

7 Choose **Download and Debug** to start the C-SPY debugger.

During execution, the checksum calculated by the linker and the checksum calculated by your application should be identical.

TROUBLESHOOTING CHECKSUM CALCULATION

If the two checksums do not match, there are several possible causes. These are some troubleshooting hints:

- If possible, start with a small example when trying to get the checksums to match.
- Verify that the exact same memory range or ranges are used in both checksum calculations.

  To help you do this, The linker produces useful information in the map file about the exact addresses that were used and the order in which they were accessed.

- Make sure that all checksum symbols are excluded from all checksum calculations.

  In the map file, notice the checksum symbol and its size, and for information about its placement, check the module map or the entry list. Compare that placement with the checksum range.

- Verify that the checksum calculations use the same polynomial.
- Verify that the bits in the bytes are processed in the same order in both checksum calculations, from the least to the most significant bit or the other way around. You control this with the **Bit order** option (or from the command line, the -m parameter of the --checksum option).
- If you are using the small variant of CRC, check whether you need to feed additional bytes into the algorithm.

  The number of zeros to add at the end of the byte sequence must match the size of the checksum, in other words, one zero for a 1-byte checksum, two zeros for a 2-byte checksum, and four zeros for a 4-byte checksum.

- Any breakpoints in flash memory change the content of the flash. This means that the checksum which is calculated by your application will no longer match the initial checksum calculated by the linker. To make the two checksums match again, you must disable all your breakpoints in flash and any breakpoints set in flash by C-SPY internally. The stack plugin and the debugger option **Run to** both require C-SPY to set breakpoints. Read more about possible breakpoint consumers in the *C-SPY® Debugging Guide for AVR*.

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. This section provides useful information for efficient treatment of data:

- Locating strings in ROM, RAM or flash
- Using efficient data types
- Floating-point types
- Memory model and memory attributes for data
- Using the best pointer type
- Anonymous structs and unions.

LOCATING STRINGS IN ROM, RAM OR FLASH

With the IAR C/C++ Compiler for AVR there are three possible locations for storing strings:

- In external ROM in the data memory space
- In internal RAM in the data memory space
- In flash in the code memory space.

To read more about this, see *Placing code and data—the linker configuration file*, page 74.

Locating strings in flash

This can be done on individual strings or for the whole application/file using the option
`--string_literals_in_flash`. Examples on how to locate individual strings into
flash:

```
__flash char str1[] = "abcdef";
__flash char str2[] = "ghi";
__flash char __flash * pVar[] = { str1, str2 };
```

String literals cannot be put in flash automatically, but you can use a local static variable
instead:

```
#include <pgmspace.h>
void f (int i)
{
  static __flash char sl[] = "%d cookies\n";
  printf_P(sl, i);
}
```

This does not result in more code compared to allowing string literals in flash.

To use flash strings, you must use alternative library routines that expect flash strings.
A few such alternative functions are provided— they are declared in the `pgmspace.h`
header file. They are flash alternatives for some common C library functions with an
extension `_P`. For your own code, you can always use the `__flash` keyword when
passing the strings between functions.

For reference information about the alternative functions, see *AVR-specific library
functions*, page 370.

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 small and unsigned data types, (`unsigned char` and `unsigned short`)
  unless your application really requires signed values.

● Bitfields with sizes other than 1 bit should be avoided because they will result in
  inefficient code compared to bit operations.

● When using arrays, it is more efficient if the type of the index expression matches
  the index type of the memory of the array. For `__near` this is `int`.

● Using floating-point types on a microprocessor without a math co-processor is very
  inefficient, both in terms of code size and execution speed.

● Declaring a pointer parameter to point to `const` data might open for better
  optimizations in the calling function.

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 `--64bit_doubles` compiler option (**Use 64-bit doubles**).

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 285.

MEMORY MODEL AND MEMORY ATTRIBUTES FOR DATA

For many applications it is sufficient to use the memory model feature to specify the default memory for the data objects. However, for individual objects it might be necessary to specify other memory attributes in certain cases, for example:

- An application where some global variables are accessed from a large number of locations. In this case they can be declared to be placed in memory with a smaller pointer type
- An application where all data, with the exception of one large chunk of data, fits into the region of one of the smaller memory types
- Data that must be placed at a specific memory location.

The IAR C/C++ Compiler for AVR provides memory attributes for placing data objects in the different memory spaces, and for the DATA and CODE space (flash) there are different memory attributes for placing data objects in different memory types, see *Using data memory attributes*, page 54.

Efficient usage of memory type attributes can significantly reduce the application size.

For details about the memory types, see *Memory types*, page 54.

USING THE BEST POINTER TYPE

The generic pointers, pointers declared `__generic`, can point to all memory spaces, which makes them simple and also tempting to use. However, they carry a cost in that special code is needed before each pointer access to check which memory a pointer points to and taking the appropriate actions. Use the smallest pointer type that you can, and avoid any generic pointers unless necessary.

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 AVR 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 250, 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;
  };
} @ 40;


/* The variables are used here. */
void Test(void)
{
  IOPORT = 0;
  way = 1;
  out = 1;
}
```

This declares an I/O register byte `IOPORT` at address `40`. 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:

- Memory models

    Use the different compiler options for memory models to take advantage of the different addressing modes available for the microcontroller and thereby also place data objects in different parts of memory. For more information about memory models, see *Memory models*, page 59, and *Function storage*, page 63, respectively.

- Memory attributes

    Using IAR-specific keywords or pragma directives, you can override the default placement of functions, variables, and constants. For more information, see *Using function memory attributes*, page 63 and *Using data memory attributes*, page 54.

- 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 210.

- The @ operator and the #pragma location directive for segment placement.

    Using the @ operator or the #pragma location directive, you can place individual functions, variables, and constants in named segments. The placement of these segments can then be controlled by linker directives. For more information, see *Data and function placement in segments*, page 212.

- Using the --segment option, you can set the default segment for functions, variables, and constants in a particular module. For more information, see *--segment*, page 271.

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)

● `const` (with 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.

Note: All declarations of `__no_init` variables placed at an absolute address are *tentative definitions*. Tentatively defined variables are only kept in the output from the compiler if they are needed in the module being compiled. Such variables will be defined in all modules in which they are used, which will work as long as they are defined in the same way. The recommendation is to place all such declarations in header files that are included in all modules that use the variables.

Examples

In this example, a `__no_init` declared variable is placed at an absolute address. This is useful for interfacing between multiple processes, applications, etc:

```
__no_init volatile char alpha @ 0xFF2000;/* OK */
```

The next example contains two `const` declared objects. The first one is not initialized, and the second one is initialized to a specific value. Both objects are placed in ROM. This is useful for configuration parameters, which are accessible from an external interface. Note that in the second case, the compiler is not obliged to actually read from the variable, because the value is known.

```
#pragma location=0xFF2002
__no_init const int beta;                  /* OK */

const int gamma @ 0xFF2004 = 3;            /* OK */
```

In the first case, the value is not initialized by the compiler; the value must be set by other means. The typical use is for configurations where the values are loaded to ROM separately, or for special function registers that are read-only.

This shows incorrect usage:

```
int delta @ 0xFF2006;               /* Error, neither */
                                    /* "__no_init" nor "const".*/
```

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 SEGMENTS

The following methods can be used for placing data or functions in named segments other than default:

- The `@` operator, alternatively the `#pragma location` directive, can be used for placing individual variables or individual functions in named segments. The named segment can either be a predefined segment, or a user-defined segment. The variables must be declared either `__no_init` or `const`. If declared `const`, they can have initializers.
- The `--segment` option can be used for placing variables and functions, which are parts of the whole compilation unit, in named segments.

C++ static member variables can be placed in named segments just like any other static variable.

If you use your own segments, in addition to the predefined segments, the segments must also be defined in the linker configuration file using the `-Z` or the `-P` segment control directives.

Note: Take care when explicitly placing a variable or function in a predefined segment 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 segments can be controlled from the linker configuration file.

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

Examples of placing variables in named segments

In the following examples, a data object is placed in a user-defined segment. 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 segment 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 compiler will warn that segments that contain zero-initialized and initialized data must be handled manually. To do this, you must use the linker option -Q to separate the initializers into one separate segment and the symbols to be initialized to a different segment. You must then write source code that copies the initializer segment to the initialized segment, and zero-initialized symbols must be cleared before they are used.

As usual, you can use memory attributes to select a memory for the variable. Note that you must as always ensure that the segment is placed in the appropriate memory area when linking.

```
__near __no_init int alpha @ "MY_NEAR_NOINIT";/* Placed in
                                                near*/
```

This example shows incorrect usage:

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

Examples of placing functions in named segments

```
void f(void) @ "MY_FUNCTIONS";

void g(void) @ "MY_FUNCTIONS"
{
}

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

Specify a memory attribute to direct the function to a specific memory, and then modify the segment placement in the linker configuration file accordingly:

```
__nearfunc void f(void) @ "MY_NEARFUNC_FUNCTIONS";
```

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 332, 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 259.

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 248.

OPTIMIZATION LEVELS

The compiler supports different levels of optimizations. This table lists optimizations that are typically performed on each level:

Optimization level	Description
None (Best debug support)	Variables live through their entire scope
Low	Same as above but variables only live for as long as they are needed, not necessarily through their entire scope
	Dead code elimination
	Redundant label elimination
	Redundant branch elimination
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
	Cross call (when optimizing for size)
	Function inlining
	Code motion
	Type-based alias analysis

Table 25: 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 216.

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, or even occasionally display an incorrect value. At any time, if you experience difficulties when debugging your code, try lowering the optimization level.

SPEED VERSUS SIZE

At the high optimization level, the compiler balances between size and speed optimizations. However, it is possible to fine-tune the optimizations explicitly for either size or speed. They only differ in what thresholds that are used; speed will trade size for

speed, whereas size will trade speed for size. 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.

You can choose an optimization goal for each module, or even individual functions, using command line options and pragma directives (see *-O*, page 265 and *optimize*, page 332). For a small embedded application, this makes it possible to achieve acceptable speed performance while minimizing the code size: Typically, only a few places in the application need to be fast, such as the most frequently executed inner loops, or the interrupt handlers.

Rather than compiling the whole application with High (Balanced) optimization, you can use High (Size) in general, but override this to get High (Speed) optimization only for those functions where the application needs to be fast.

Because of the unpredictable way in which different optimizations interact, where one optimization can enable other optimizations, sometimes a function becomes smaller when compiled with High (Speed) optimization than if High (Size) is used. Also, using multi-file compilation (see *--mfc*, page 259) can enable many optimizations to improve both speed and size performance. It is recommended that you experiment with different optimization settings so that you can pick the best ones for your project.

FINE-TUNING ENABLED TRANSFORMATIONS

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

● Common subexpression elimination
● Function inlining
● Code motion
● Type-based alias analysis
● Static clustering
● Cross call

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 261.

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 67.

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 261.

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 263.

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 260.

Cross call

Common code sequences are extracted to local subroutines. This optimization, which is performed at optimization level High size, 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.

If you want to use cross calls at any other optimization level, use --do_cross_call.

For more information about related command line options, see *--no_cross_call*, page 261, *--do_cross_call*, page 250, and *--cross_call_passes*, page 242.

Facilitating good code generation

This section contains hints on how to help the compiler generate good code:

- *Writing optimization-friendly source code*, page 219
- *Saving stack space and RAM memory*, page 219
- *Function prototypes*, page 220

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 217. 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 214.

● Avoid using inline assembler. Instead, try writing the code in C/C++, use intrinsic functions, or write a separate module in assembler language. For more information, see *Mixing C and assembler*, page 149.

SAVING STACK SPACE AND RAM MEMORY

The following is a list of programming techniques that save memory and stack space:

● If stack space is limited, avoid long call chains and recursive functions.

● Avoid using large non-scalar types, such as structures, as parameters or return type. To save stack space, you should instead pass them as pointers or, in C++, as references.

FUNCTION PROTOTYPES

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

● Prototyped

● Kernighan & Ritchie C (K&R C)

Both styles are valid C, however it is strongly recommended to use the prototyped style, and provide a prototype declaration for each public function in a header that is included both in the compilation unit defining the function and in all compilation units using it.

The compiler will not perform type checking on parameters passed to functions declared using K&R style. Using prototype declarations will also result in more efficient code in some cases, as there is no need for type promotion for these functions.

To make the compiler require that all function definitions use the prototyped style, and that all public functions have been declared before being defined, use the **Project>Options>C/C++ Compiler>Language 1>Require prototypes** compiler option (`--require_prototypes`).

Prototyped style

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

```
int Test(char, int); /* Declaration */

int Test(char ch, int i) /* Definition */
{
  return i + ch;
}
```

Kernighan & Ritchie style

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

For example:

```
int Test();      /* Declaration */

int Test(ch, i) /* Definition */
char ch;
int i;
{
  return i + ch;
}
```

INTEGER TYPES AND BIT NEGATION

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

In some cases there might be warnings (for example, for constant conditional or pointless comparison), in others just a different result than what is expected. Under certain circumstances the compiler might warn only at higher optimizations, for example, if the compiler relies on optimizations to identify some instances of constant conditionals. In this example an 8-bit character, a 16-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 `0x0080`, and `~0x0080` becomes `0xFF7F`. 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 308.

For more information about the `volatile` type qualifier and the rules for accessing volatile objects, see *Declaring objects volatile*, page 291.

Protecting the eeprom write mechanism

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

ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for several AVR devices are included in the IAR product installation. The header files are named `iodevice.h` and define the processor-specific special function registers (SFRs).

To enable bit definitions in IAR Embedded Workbench, select the option **General Options>System>Enable bit definitions in I/O include files**.

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 `iom128.h`:

```
__io union
{
  unsigned char PORTE;     /* The sfrb as 1 byte */
  struct
  {
    unsigned char PORTE_Bit0:1,
                  PORTE_Bit1:1,
                  PORTE_Bit2:1,
                  PORTE_Bit3:1,
                  PORTE_Bit4:1,
                  PORTE_Bit5:1,
                  PORTE_Bit6:1,
                  PORTE_Bit7:1;
  };
} @ 0x1F;
```

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:

```
/* whole register access */
PORTE = 0x12;

/* Bitfield accesses */
PORTE_Bit0 = 1;
```

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

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. See the chapter *Linking overview* for more information.

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

Variables declared using the `__no_init` keyword could, for example, be large input buffers or mapped to special RAM that keeps its content even when the application is turned off.

For more information, see *__no_init*, page 311. Note that to use this keyword, language extensions must be enabled; see *-e*, page 250. For more information, see also *object_attribute*, page 332.

Part 2. Reference information

This part of the *IAR C/C++ Compiler User Guide for AVR* contains these chapters:

- External interface details

- Compiler options

- Data representation

- Extended keywords

- Pragma directives

- Intrinsic functions

- The preprocessor

- C/C++ standard library functions

- Segment reference

- The stack usage control file

- Implementation-defined behavior for Standard C

- Implementation-defined behavior for C89.
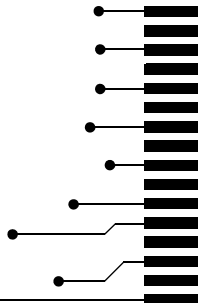
External interface details

- Invocation syntax

- Include file search procedure

- Compiler output

- Diagnostics

Invocation syntax

You can use the compiler either from the IDE or from the command line. See the *IDE Project Management and Building Guide for AVR* for information about using the compiler from the IDE.

COMPILER INVOCATION SYNTAX

The invocation syntax for the compiler is:

```
iccavr [options] [sourcefile] [options]
```

For example, when compiling the source file `prog.c`, use this command to generate an object file with debug information:

```
iccavr 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.

PASSING OPTIONS

There are three different ways of passing options to the compiler:

● Directly from the command line

Specify the options on the command line after the `iccavr` command, either before or after the source filename; see *Invocation syntax*, page 227.

● Via environment variables

The compiler automatically appends the value of the environment variables to every command line; see *Environment variables*, page 228.

● Via a text file, using the `-f` option; see *-f*, page 254.

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 7.*n*\avr\inc;c:\headers
QCCAVR	Specifies command line options; for example: QCCAVR=-lA asm.lst

Table 26: Compiler 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 255.

2 The directories specified using the `C_INCLUDE` environment variable, if any; see *Environment variables*, page 228.

3 The automatically set up library system include directories. See *--clib*, page 241, *--dlib*, page 248, and *--dlib_config*, page 249.

- 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:

```
iccavr ..\src\src.c -I..\include -I..\debugconfig
```

Then the following directories are searched in the order listed below for the file `config.h`, which in this example is located in the `dir\debugconfig` directory:

`dir\include`	Current file is `src.h`.
`dir\src`	File including current file (`src.c`).
`dir\include`	As specified with the first `-I` option.
`dir\debugconfig`	As specified with the second `-I` option.

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

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

For more information, see *Overview of the preprocessor*, page 351.

Compiler output

The compiler can produce the following output:

- A linkable object file

  The object files produced by the compiler use a proprietary format called UBROF, which stands for Universal Binary Relocatable Object Format. By default, the object file has the filename extension `r90`.

- Optional list files

  Various kinds of list files can be specified using the compiler option -l, see *-l*, page 256. 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 231.

- 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 230.

- 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 returns 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 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 compiler abort.
4	Internal errors occurred, making the compiler abort.

Table 27: Error return codes

Diagnostics

This section describes the format of the diagnostic messages and explains how diagnostic messages are divided into different levels of severity.

MESSAGE FORMAT

All diagnostic messages are issued as complete, self-explanatory messages. A typical diagnostic message from the compiler is produced in the form:

```
filename,linenumber level[tag]: message
```

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.

SEVERITY LEVELS

The diagnostic messages are divided into different levels of severity:

Remark

A diagnostic message that is produced when the compiler finds a source code construct that can possibly lead to erroneous behavior in the generated code. Remarks are by default not issued, but can be enabled, see *--remarks*, page 269.

Warning

A diagnostic message that is produced when the compiler finds a potential programming error or omission which is of concern, but which does not prevent completion of the compilation. Warnings can be disabled by use of the command line option `--no_warnings`, see *--no_warnings*, page 265.

Error

A diagnostic message that is produced when the compiler finds a construct which clearly violates the C or C++ language rules, such that code cannot be produced. An error will produce a non-zero exit code.

Fatal error

A diagnostic message that is produced when the compiler finds a condition that not only prevents code generation, but which makes further processing of the source code 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 the chapter *Compiler options*, for information about the compiler options that are available for setting severity levels.

See 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. 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, which can be seen in the header of the list files generated by the compiler

● Your license number

● The exact internal error message text

● The source file of the 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 228.

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\:

```
iccavr 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:

  ```
  iccavr 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:

  ```
  iccavr 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:

  ```
  iccavr 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:

  ```
  iccavr 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
--64bit_doubles	Forces the compiler to use 64-bit doubles
--64k_flash	Specifies a maximum of 64 Kbytes flash memory
--c89	Specifies the C89 dialect
--char_is_signed	Treats char as signed
--char_is_unsigned	Treats char as unsigned

Table 28: Compiler options summary

Command line option	Description
`--clib`	Uses the system include files for the CLIB library
`--cpu`	Specifies a specific device
`--cross_call_passes`	Cross call optimization
`-D`	Defines preprocessor symbols
`--debug`	Generates debug information
`--dependencies`	Lists file dependencies
`--diag_error`	Treats these as errors
`--diag_remark`	Treats these as remarks
`--diag_suppress`	Suppresses these diagnostics
`--diag_warning`	Treats these as warnings
`--diagnostics_tables`	Lists all diagnostic messages
`--disable_all_program_memory_load_instructions`	Disables the `LPM`/`ELPM` instructions
`--disable_direct_mode`	Disables direct addressing mode
`--disable_library_knowledge`	Disables the knowledge about the support routines in the library
`--disable_mul`	Disables the `MUL` instructions
`--disable_spm`	Disables the `SPM` instructions
`--discard_unused_publics`	Discards unused public symbols
`--dlib`	Uses the system include files for the DLIB library
`--dlib_config`	Uses the system include files for the DLIB library and determines which configuration of the library to use
`--do_cross_call`	Forces the compiler to run the cross call optimizer.
`-e`	Enables language extensions
`--ec++`	Specifies Embedded C++
`--eec++`	Specifies Extended Embedded C++
`--eecr_address`	Defines the EECR address
`--eeprom_size`	Specifies the EEPROM size
`--enable_external_bus`	Adds code to enable external data bus
`--enable_multibytes`	Enables support for multibyte characters in source files

Table 28: Compiler options summary (Continued)

Command line option	Description
--enable_restrict	Enables the Standard C keyword `restrict`
--enhanced_core	Enables enhanced instruction set
--error_limit	Specifies the allowed number of errors before compilation stops
-f	Extends the command line
--force_switch_type	Forces the switch type
--guard_calls	Enables guards for function static variable initialization
--header_context	Lists all referred source files and header files
-I	Specifies include file path
--initializers_in_flash	Places aggregate initializers in flash memory
-l	Creates a list file
--library_module	Creates a library module
--lock_regs	Locks registers
-m	Specifies a memory model
--macro_positions_in _diagnostics	Obtains positions inside macros in diagnostic messages
--memory_model	Specifies a memory model
--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*.
--module_name	Sets the object module name
--no_call_frame_info	Disables output of call frame information
--no_clustering	Disables static clustering optimizations

Table 28: Compiler options summary (Continued)

Command line option	Description
`--no_code_motion`	Disables code motion optimization
`--no_cross_call`	Disables cross-call optimization
`--no_cse`	Disables common subexpression elimination
`--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_rampd`	Uses `RAMPZ` instead of `RAMPD`
`--no_size_constraints`	Relaxes the normal restrictions for code size expansion when optimizing for speed.
`--no_static_destruction`	Disables destruction of C++ static variables at program exit
`--no_system_include`	Disables the automatic search for system include files
`--no_tbaa`	Disables type-based alias analysis
`--no_typedefs_in_diagnostics`	Disables the use of typedef names in diagnostics
`--no_ubrof_messages`	Excludes messages from UBROF files
`--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`.
`--omit_types`	Excludes type information
`--only_stdout`	Uses standard output only
`--output`	Sets the object filename
`--pending_instantiations`	Sets the maximum number of instantiations of a given C++ template.
`--predef_macros`	Lists the predefined symbols.
`--preinclude`	Includes an include file before reading the source file
`--preprocess`	Generates preprocessor output
`--public_equ`	Defines a global named assembler label
`-r`	Generates debug information. Alias for `--debug`.

Table 28: Compiler options summary (Continued)

Command line option	Description
`--relaxed_fp`	Relaxes the rules for optimizing floating-point expressions
`--remarks`	Enables remarks
`--require_prototypes`	Verifies that functions are declared before they are defined
`--root_variables`	Specifies variables as `__root`
`-s`	Optimizes for speed.
`--segment`	Changes a segment name
`--separate_cluster_for_ initialized_variables`	Separates initialized and non-initialized variables
`--silent`	Sets silent operation
`--spmcr_address`	Defines the SPMCR address
`--strict`	Checks for strict compliance with Standard C/C++
`--string_literals_in_flash`	Puts "string" in the `__nearflash` or `__farflash` segment
`--system_include_dir`	Specifies the path for system include files
`--use_c++_inline`	Uses C++ inline semantics in C99
`-v`	Specifies the processor variant
`--version`	
`--version1_calls`	Uses the `ICCA90` calling convention
`--version2_calls`	Uses the V2.10-V4.10B calling convention
`--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
`--xmcra_address`	Specifies where XMCRA is located
`-y`	Places constants and literals
`-z`	Optimizes for size
`--zero_register`	Specifies register `R15` as the zero register

Table 28: 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.

--64bit_doubles

Syntax   `--64bit_doubles`

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

See also   *Floating-point types*, page 285.

Project>Options>General Options>Target

--64k_flash

Syntax   `--64k_flash`

Description   This option tells the compiler that the intended target processor does not have more than 64 Kbytes program flash memory (small flash), and that the AVR core therefore does not have the `RAMPZ` register or the `ELPM` instruction.

This option can only be used together with the `-v2`, `-v3`, and `-v4` processor options.

Project>Options>General Options>Target (No RAMPZ register)

--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 167.

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

--clib

Syntax   `--clib`

Description   Use this option to use the system header files for the CLIB library; the compiler will automatically locate the files and use them when compiling.

Note: The CLIB library is used by default. To use the DLIB library, use the `--dlib` or the `--dlib_config` option instead.

See also   *--dlib*, page 248 and *--dlib_config*, page 249.

To set related options, choose:

Project>Options>General Options>Library Configuration

--cpu

Syntax   `--cpu=`*`processor`*

Parameters

processor   Specifies a specific device

Description   The compiler supports different processor variants. Use this option to select a specific processor variant for which the code will be generated.

Note that to specify the processor, you can use either the `--cpu` option or the `-v` option. The `--cpu` option is, however, more precise because it contains more information about the intended target than the more generic `-v` option.

See also   *-v*, page 275, *Processor configuration*, page 49.

To set related options, choose:

Project>Options>General Options>Target>Processor configuration

--cross_call_passes

Syntax   `--cross_call_passes=`*`N`*

Parameters

N   The number of times to run the cross call optimizer, which can be 1–5.

Description   Use this option to run the cross-call optimizer for decreasing the RSTACK usage. The default is to run it until no more improvements can be made.

Note: Use this option if you have a target processor with a hardware stack or a small internal return stack segment, RSTACK.

See also   *--no_cross_call*, page 261.

To set related options, choose:

Project>Options>C/C++ Compiler>Optimizations

-D

Syntax   `-D symbol[=value]`

Parameters

`symbol`	The name of the preprocessor symbol
`value`	The value of the preprocessor symbol

Description   Use this option to define a preprocessor symbol. If no value is specified, `1` is used. This option can be used one or more times on the command line.

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

`-Dsymbol`

is equivalent to:

`#define symbol 1`

To get the equivalence of:

`#define FOO`

specify the `=` sign but nothing after, for example:

`-DFOO=`

Project>Options>C/C++ Compiler>Preprocessor>Defined symbols

--debug, -r

Syntax   `--debug`
                          `-r`

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

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

Project>Options>C/C++ Compiler>Output>Generate debug information

--dependencies

Syntax

```
--dependencies[=[i|m|n][s]] {filename|directory|+}
```

Parameters

i (default)	Lists only the names of files
m	Lists in makefile style (multiple rules)
n	Lists in makefile style (one rule)
s	Suppresses system files
+	Gives the same output as -o, but with the filename extension d

See also *Rules for specifying a filename or directory as parameters*, page 234.

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.r90: c:\iar\product\include\stdio.h
foo.r90: 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:

```
%.r90 : %.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 234.

Description   Use this option to list all possible diagnostic messages to a named file. This can be
                      convenient, for example, if you have used a pragma directive to suppress or change the
                      severity level of any diagnostic messages, but forgot to document why.

                      Typically, this option cannot be given together with other options.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--disable_all_program_memory_load_instructions

Syntax   `--disable_all_program_memory_load_instructions`

Description   Use this option to disable the LPM/ELPM instructions.

To set this option, use: **Project>Options>C/C++ Compiler>Extra Options**.

--disable_direct_mode

Syntax   `--disable_direct_mode`

Description   Use this option to prevent the compiler from generating the direct addressing mode instructions LDS and STS.

To set this option, use: **Project>Options>C/C++ Compiler>Extra Options**.

--disable_library_knowledge

Syntax   `--disable_library_knowledge`

Description   Use this option to disable the knowledge about the support routines in the library.

To set this option, use: **Project>Options>C/C++ Compiler>Extra Options**.

--disable_mul

Syntax   `--disable_mul`

Description   Use this option to disable the MUL instructions.

To set this option, use: **Project>Options>C/C++ Compiler>Extra Options**.

--disable_spm

Syntax   `--disable_spm`

Description   Use this option to disable the SPM instruction.

To set this option, use: **Project>Options>C/C++ Compiler>Extra Options**.

--discard_unused_publics

Syntax   `--discard_unused_publics`

Description   Use this option to discard unused public functions and variables when compiling with the `--mfc` compiler option.

Note: Do not use this option only on parts of the application, as necessary symbols might be removed from the generated output. Use the object attribute `__root` to keep symbols that are used from outside the compilation unit, for example interrupt handlers. If the symbol does not have the `__root` attribute and is defined in the library, the library definition will be used instead.

See also   *--mfc*, page 259 and *Multi-file compilation units*, page 214.

Project>Options>C/C++ Compiler>Discard unused publics

--dlib

Syntax   `--dlib`

Description   Use this option to use the system header files for the DLIB library; the compiler will automatically locate the files and use them when compiling.

Note: The DLIB library is used by default: To use the CLIB library, use the `--clib` option instead.

See also   *--dlib_config*, page 249, *--no_system_include*, page 263, *--system_include_dir*, page 274, and *--clib*, page 241.

To set related options, choose:

Project>Options>General Options>Library Configuration

--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 234.
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
avr\lib. For examples and information about prebuilt runtime libraries, see *Prebuilt
runtime libraries*, page 111.

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 *Customizing and building your own runtime
library*, page 108.

Note: This option only applies to the IAR DLIB runtime environment.

To set related options, choose:

Project>Options>General Options>Library Configuration

--do_cross_call

Syntax   --do_cross_call

Description   Use this option to force the compiler to run the cross call optimizer, regardless of the
                   optimization level. The cross call optimizer is otherwise only run at high size
                   optimization.

See also   *-s*, page 271.

 To set related options, choose:

Project>Options>C/C++ Compiler>Optimizations

-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 169.

 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 176.

Project>Options>C/C++ Compiler>Language 1>C++

and

Project>Options>C/C++ Compiler>Language 1>C++ dialect>Extended Embedded C++

--eecr_address

Syntax	`--eecr_address` *address*

Parameters

address   The value of the `EECR` address. The default is `0x1C`.

Description   If you use the `-v` processor option, the `--eecr_address` option can be used for modifying the value of the `EECR` address.

If you use the `--cpu` processor option, the `--eecr_address` option is implicitly set, which means you should not use these options together.

See also   *-v*, page 275 and *--cpu*, page 242.

To set this option, use: **Project>Options>C/C++ Compiler>Extra Options**.

--eeprom_size

Syntax	`--eeprom_size=`*N*

Parameters

N   The size of the EEPROM in bytes, `0-65536`.

Description   Use this option to enable the `__eeprom` extended keyword by specifying the size of the built-in EEPROM.

To use the `__eeprom` extended keyword, language extensions must be enabled.

See also   *Function storage*, page 63, *-e*, page 250 and *language*, page 330.

To set related options, choose:

Project>Options>C/C++ Compiler>Code

--enable_external_bus

Syntax   `--enable_external_bus`

Description   Use this option to make the compiler add the special `__require` statement which makes XLINK include the code in `startup.s90` that enables the external data bus. Use this option if you intend to place RSTACK in external RAM.

Note: The code in `cstartup.s90` that enables the external data bus is preferably placed in `low_level_init` instead.

See also   *The return address stack*, page 193.

Project>Options>General Options>System

--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**

--enhanced_core

Syntax   `--enhanced_core`

Description   Use this option to allow the compiler to generate instructions from the enhanced
                        instruction set that is available in some AVR devices, for example ATmega161.

                        The enhanced instruction set consists of these instructions:

```
MUL
MOVW
MULS
MULSU
FMUL
FMULS
FMULSU
LPM Rd,Z
LPM Rd,Z+
ELPM Rd,Z
ELPM Rd,Z+
SPM
```

                        Project>Options>General Options>Target>Enhanced core

--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 234.

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**.

--force_switch_type

Syntax   `--force_switch_type={0|1|2}`

Parameters

0	Library call with switch table
1	Inline code with switch table
2	Inline compare/jump logic

Description   Use this option to set the switch type.

 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 135.

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 228.

Project>Options>C/C++ Compiler>Preprocessor>Additional include directories

--initializers_in_flash

Syntax   --initializers_in_flash

Description   Use this option to place aggregate initializers in flash memory. These initializers are
                       otherwise placed in the external segments _C or in the flash segments _F if the compiler
                       option -y was also specified.

                       An aggregate initializer—an array or a struct—is constant data that is copied to the stack
                       dynamically at runtime, in this case every time a function is entered.

                       The drawback of placing data in flash memory is that it takes more time to copy it; the
                       advantage is that it does not occupy memory in the data space.

                       Local variables with aggregate initializers are copied from the segments:

Data	Default	-y	--initializers_in_flash
auto aggregates	NEAR_C	NEAR_F	NEAR_F

Table 29: Accessing variables with aggregate initializers

Example
```
void Func ()
{
  char buf[4] = { '1', 'd', 'g', 't' };
  ...
}
```

See also   *-y*, page 279 and *Initialization of local aggregates at function invocation*, page 77.

                       To set related options, choose:

                       Project>Options>C/C++ Compiler>Code

-l

Syntax   -l[a|A|b|B|c|C|D][N][H] {*filename*|*directory*}

Parameters

a (default)   Assembler list file

A   Assembler list file with C or C++ source as comments

b   Basic assembler list file. This file has the same contents as a
                       list file produced with -la, except that no extra
                       compiler-generated information (runtime model attributes,
                       call frame information, frame size information) is included *

B	Basic assembler list file. This file has the same contents as a list file produced with -1A, 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 234.

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

--library_module

Syntax   `--library_module`

Description   Use this option to make the compiler generate a library module rather than a program module. A program module is always included during linking. A library module will only be included if it is referenced in your program.

Project>Options>C/C++ Compiler>Output>Module type>Library Module

257

--lock_regs

Syntax   `--lock_regs N`

Parameters

N   The number of registers to lock, `0-12`.

Description   Use this option to lock registers that are to be used for global register variables. When you use this option, the registers `R15` and downwards will be locked.

To maintain module consistency, make sure to lock the same number of registers in all modules.

Note: Locking more than nine registers might cause linking to fail. Because the pre-built libraries delivered with the product do not lock any registers, a library function might potentially use any of the lockable registers. Any such resource conflict between locked registers and compiler-used registers will be reported at link time.

If you need to lock any registers in your code, the library must therefore be rebuilt with the same set of locked registers.

To set related options, choose:

Project>Options>C/C++ Compiler>Code

--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**.

--memory_model, -m

Syntax   `--memory_model={tiny|t|small|s|large|l|huge|h}`

`-m{tiny|t|small|s|large|l<huge|h}`

Parameters

`tiny, t`   Specifies the Tiny memory model

`small, s`   Specifies the Small memory model

`large, l`	Specifies the Large memory model
`huge, h`	Specifies the Huge memory model

Description   Use this option to specify the memory model, for which the code is to be generated.

By default, the compiler generates code for the Tiny memory model for all processor options except `-v4` and `-v6` where the Small memory model is the default.

Use the `-m` or the `--memory_model` option if you want to generate code for a different memory model.

Example   To generate code for the Large memory model, give the command:

```
iccavr filename -ml
```

or:

```
iccavr filename --memory_model=large
```

See also   *Memory models*, page 59.

To set related options, choose:

Project>Options>General Options>Memory model

--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   `iccavr myfile1.c myfile2.c myfile3.c --mfc`

See also   *--discard_unused_publics*, page 248, *--output, -o*, page 267, and *Multi-file compilation units*, page 214.

Project>Options>C/C++ Compiler>Multi-file compilation

--module_name

Syntax	`--module_name=name`

Parameters

`name`	An explicit object module name

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

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

Project>Options>C/C++ Compiler>Output>Object module name

--no_call_frame_info

Syntax	`--no_call_frame_info`

Description   Normally, the compiler always generates call frame information in the output, to enable the debugger to display the call stack even in code from modules with no debug information. Use this option to disable the generation of call frame information.

See also   *Call frame information*, page 165.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--no_clustering

Syntax	`--no_clustering`

Description   Use this option to disable static clustering optimizations.

Note: This option has no effect at optimization levels below Medium.

See also   *Static clustering*, page 218.

Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Clustering of variables

--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 217.

                                    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.

                                    Use this option to disable the cross-call optimization. This is highly recommended if your target processor has a hardware stack or a small internal return stack segment, `RSTACK`, because this option reduces the usage of `RSTACK`.

                                    This optimization is performed at size optimization, level High. Note that, although the optimization can drastically reduce the code size, this optimization increases the execution time.

See also   *--cross_call_passes*, page 242, *Cross call*, page 218.

                                    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 216.

                                    Project>Options>C/C++ Compiler>Optimizations>Enable transformations>Common subexpression elimination

--no_inline

Syntax	`--no_inline`
Description	Use this option to disable function inlining.
See also	*Inlining functions*, page 67.

 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_rampd

Syntax	`--no_rampd`
Description	Use this option to make the compiler use the RAMPZ register instead of RAMPD. This option corresponds to the instructions LDS and STS.
	Note that this option is only useful on processor variants with more than 64 Kbytes data (-v4 and -v6).

 To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--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 215.

 To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--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   *System termination*, page 121.

 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*, page 248, *--dlib_config*, page 249, and *--system_include_dir*, page 274.

 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 217.

> **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_ubrof_messages

Syntax   `--no_ubrof_messages`

Description   Use this option to minimize the size of your application object file by excluding messages from the UBROF files. The file size can decrease by up to 60%. Note that the XLINK diagnostic messages will, however, be less useful when you use this option.

> 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.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--no_warnings

Syntax	`--no_warnings`
Description	By default, the compiler issues warning messages. Use this option to disable all warning messages.

This option is not available in the IDE.

--no_wrap_diagnostics

Syntax	`--no_wrap_diagnostics`
Description	By default, long lines in diagnostic messages are broken into several lines to make the message easier to read. Use this option to disable line wrapping of diagnostic messages.

This option is not available in the IDE.

-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

n	**None* (Best debug support)**
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 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 214.

Project>Options>C/C++ Compiler>Optimizations

--omit_types

Syntax   --omit_types

Description   By default, the compiler includes type information about variables and functions in the object output. Use this option if you do not want the compiler to include this type information in the output, which is useful when you build a library that should not contain type information. The object file will then only contain type information that is a part of a symbol's name. This means that the linker cannot check symbol references for type correctness.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--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 234.
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 `r90`. Use this option to explicitly specify a different output filename for the object code output.

This option is not available in the IDE.

--pending_instantiations

Syntax	`--pending_instantiations` *`number`*

Parameters

number	An integer that specifies the limit, where `64` is default. If `0` is used, there is no limit.

Description	Use this option to specify the maximum number of instantiations of a given C++ template that is allowed to be in process of being instantiated at a given time. This is used for detecting recursive instantiations.

Project>Options>C/C++ Compiler>Extra Options

--predef_macros

Syntax	`--predef_macros {`*`filename`*`|`*`directory`*`}`
Parameters	See *Rules for specifying a filename or directory as parameters*, page 234.
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 234.

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 234.

Description
Use this option to generate preprocessed output to a named file.

Project>Options>C/C++ Compiler>Preprocessor>Preprocessor output to file

--public_equ

Syntax
--public_equ *symbol*[=*value*]

Parameters

symbol	The name of the assembler symbol to be defined
value	An optional value of the defined assembler symbol

Description This option is equivalent to defining a label in assembler language using the `EQU`
directive and exporting it using the `PUBLIC` directive. This option can be used more than
once on the command line.

This option is not available in the IDE.

--relaxed_fp

Syntax `--relaxed_fp`

Description Use this option to allow the compiler to relax the language rules and perform more
aggressive optimization of floating-point expressions. This option improves
performance for floating-point expressions that fulfill these conditions:

● The expression consists of both single- and double-precision values
● The double-precision values can be converted to single precision without loss of
accuracy
● The result of the expression is converted to single precision.

Note 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

--root_variables

Syntax
`--root_variables`

Description
Use this option to apply the `__root` extended keyword to all global and static variables. This will make sure that the variables are not removed by the IAR XLINK Linker.

Note: The `--root_variables` option is always available, even if you do not specify the compiler option `-e`, language extensions.

To set related options, choose:

Project>Options>C/C++ Compiler>Code>Force generation of all global and static variables

-s

Syntax   -s[2|3|6|9]

Parameters

2	**None* (Best debug support)**
3 (default)	Low*
6	Medium
9	High (Maximum optimization). This corresponds to the -Oh option.

*The most important difference between -s2 and -s3 is that at level 2, all non-static variables will live during their entire scope.

Description   Use this option to make the compiler optimize the code for maximum execution speed. If no optimization option is specified, the optimization level 3 is used by default.

A low level of optimization makes it relatively easy to follow the program flow in the debugger, and, conversely, a high level of optimization makes it relatively hard.

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

Note: This option is available for backward compatibility. Use -o instead.

See also   *-O*, page 265 and *-z*, page 279

Project>Options>C/C++ Compiler>Optimizations>Speed

--segment

Syntax   --segment __*memory_attribute*=*NEWSEGMENTNAME*

Description   The compiler places functions and data objects into named segments which are referred to by the IAR XLINK Linker. Use the --segment option to perform one of these operations:

● To place all functions or data objects declared with the __*memory_attribute* in segments with names that begin with *NEWSEGMENTNAME*.

● To change the name of the segments SWITCH, INITTAB, CSTACK, RSTACK, and INTVEC to a different name.

This is useful if you want to place your code or data in different address ranges and you find the @ notation, alternatively the #pragma location directive, insufficient. Note

that any changes to the segment names require corresponding modifications in the linker configuration file.

Example

This command places the `__near int a;` defined variable in the `MYDATA_Z` segment:

```
--segment __near=MYDATA
```

This command names the segment that contains interrupt vectors to `MYINTS` instead of the default name `INTVEC`:

```
--segment intvec=MYINTS
```

See also

Controlling data and function placement in memory, page 210 and *Summary of extended keywords*, page 299.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--separate_cluster_for_initialized_variables

Syntax

```
--separate_cluster_for_initialized_variables
```

Description

Use this option to separate initialized and non-initialized variables when using variable clustering. The option makes the `*_ID` segments smaller but can generate larger code.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--silent

Syntax

```
--silent
```

Description

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

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

This option is not available in the IDE.

--spmcr_address

Syntax   `--spmcr_address` *address*

Parameters

    address   The SPMCR address, where `0x37` is the default.

Description   Use this option to set the SPMCR address.

 To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--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 169.

 Project>Options>C/C++ Compiler>Language 1>Language conformance>Strict

--string_literals_in_flash

Syntax   `--string_literals_in_flash`

Description   Use this option to put "string" in the `__nearflash` or `__farflash` segment depending on the processor option.

    When this option is used, library functions taking a string literal as a parameter will no longer be type-compatible. Use the `*_P` library function variants (for example `printf_P`).

See also   *AVR-specific library functions*, page 370.

 To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

--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 234.

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 249, and *--no_system_include*, page 263.

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 67

Project>Options>C/C++ Compiler>Language 1>C dialect>C99>C++ inline semantics

-v

Syntax	-v{0|1|2|3|4|5|6}

Parameters

0 (default)	The code space is physically limited to 8 Kbytes, and the RCALL/RJMP instructions are used for reaching the code space. Interrupt vectors are 2 bytes long. The compiler assumes that the index registers X, Y, and Z are eight bits wide when accessing the built-in SRAM. It also assumes that it is not possible to attach any external memory to the microcontroller and that it therefore should not generate any constant segment in data space (_C segment). Instead the compiler adds an implicit -y command line option. It will also try to place all aggregate initializers in flash memory, that is, the implicit --initializers_in_flash option is also added to the command line. Relative function calls are made.
1	The code space is physically limited to 8 Kbytes, and the RCALL/RJMP instructions are used for reaching the code space. Interrupt vectors are 2 bytes long. The compiler assumes that it is possible to have external memory and will therefore generate _C segments. Relative function calls are made.
2	The code space is physically limited to 128 Kbytes, and, if possible, the RCALL/RJMP instructions are used for reaching the code space. If that is not possible, CALL/JMP is used. Function calls through function pointers use ICALL/IJMP. 2 bytes are used for all function pointers. Interrupt vectors are 4 bytes long. The compiler assumes that the index registers X, Y, and Z are eight bits wide when accessing the built-in SRAM. It also assumes that it is not possible to attach any external memory to the microcontroller and that it should therefore not generate any constant segment in data space (_C segment). Instead the compiler adds an implicit -y command line option. It will also try to place all aggregate initializers in flash memory, that is, the implicit --initializers_in_flash option is also added to the command line.
3	The code space is physically limited to 128 Kbytes, and, if possible, the RCALL/RJMP instructions are used for reaching the code space. If that is not possible, CALL/JMP is used. Function calls through function pointers use ICALL/IJMP. Two bytes are used for all function pointers. Interrupt vectors are four bytes long. The compiler assumes that it is possible to have external memory and will therefore generate _C segments.

4	The code space is physically limited to 128 Kbytes, and, if possible, the RCALL/RJMP instructions are used for reaching the code space. If that is not possible, CALL/JMP is used. Function calls through function pointers use ICALL/IJMP. Two bytes are used for all function pointers. Interrupt vectors are four bytes long. The compiler assumes that it is possible to have external memory and will therefore generate _C segments.
5	Allows function calls through farfunc function pointers to cover the entire 8 Mbyte code space by using EICALL/EIJMP. Three bytes are used for function pointers. Interrupt vectors are four bytes long. The compiler assumes that it is possible to have external memory and will therefore generate _C segments.
6	Allows function calls through farfunc function pointers to cover the entire 8 Mbyte code space by using EICALL/EIJMP. Three bytes are used for function pointers. Interrupt vectors are four bytes long.The compiler assumes that it is possible to have external memory and will therefore generate _C segments.

Description   Use this option to select the processor device for which the code is to be generated.

See also   *--cpu*, page 242 and *Processor configuration*, page 49.

To set related options, choose:

Project>Options>General Options>Processor configuration

--version

Syntax   --version

Description   Use this option to make the compiler send version information to the console and then exit.

This option is not available in the IDE.

--version1_calls

Syntax   --version1_calls

Description   Use this option to make all functions and function calls use the calling convention of the A90 IAR Compiler, ICCA90, which is described in *Calling convention*, page 155.

To change the calling convention of a single function, use the `__version_1` function type attribute. See *__version_1*, page 315 for detailed information.

--version2_calls

Syntax   `--version2_calls`

Description   Use this option to select the V2.10-V4.10B calling convention.

See also   *Calling convention*, page 155.

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 167.

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

--xmcra_address

Syntax

`--xmcra_address address`

Parameters

address   The address of XMCRA. The default is `0x74`.

Description

Use this option to specify where XMCRA is located.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

-y

Syntax	`-y`

Description   Use this option to override the default placement of constants and literals.

Without this option, constants and literals are placed in an external `const` segment, *segmentbasename*`_C`. With this option, constants and literals will instead be placed in the initialized *segmentbasename*`_I` data segments that are copied from the *segmentbasename*`_ID` segments by the system startup code.

Note that `-y` is implicit in the tiny memory model.

This option can be combined with the option `--initializers_in_flash`.

See also   *--initializers_in_flash*, page 256

To set related options, choose:

Project>Options>C/C++ Compiler>Code

-z

Syntax	`-z[2|3|6|9]`

Parameters

2	**None* (Best debug support)**
3 (default)	Low*
6	Medium
9	High (Maximum optimization). This corresponds to the `-Ohz` option.

*The most important difference between `-z2` and `-z3` is that at level 2, all non-static variables will live during their entire scope.

Description   Use this option to make the compiler optimize the code for minimum size. If no optimization option is specified, the optimization level 3 is used by default.

A low level of optimization makes it relatively easy to follow the program flow in the debugger, and, conversely, a high level of optimization makes it relatively hard.

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

Note: This option is available for backward compatibility. Use `-O` instead.

See also   *-O*, page 265 and *-s*, page 271

Project>Options>C/C++ Compiler>Optimizations>Size

--zero_register

Syntax   `--zero_register`

Description   Use this option to make the compiler use register `R15` as zero register, which means that register `R15` is assumed to always contain zero.

This option can in some cases reduce the size of the generated code, especially in the Large memory model. The option might be incompatible with the supplied runtime libraries. The linker will issue a link-time error if any incompatibilities arise.

To set this option, use **Project>Options>C/C++ Compiler>Extra Options**.

Data representation

- Alignment

- Byte order

- 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 and pointers provide the most efficient code for your application.

Alignment

Every C data object has an alignment that controls how the object can be stored in memory. Should an object have an alignment of, for example, 4, it must be stored on an address that is divisible by 4.

The reason for the concept of alignment is that some processors have hardware limitations for how the memory can be accessed.

Assume that a processor can read 4 bytes of memory using one instruction, but only when the memory read is placed on an address divisible by 4. Then, 4-byte objects, such as `long` integers, will have alignment 4.

Another processor might only be able to read 2 bytes at a time; in that environment, the alignment for a 4-byte `long` integer might be 2.

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.

Note that with the `#pragma data_alignment` directive you can increase the alignment demands on specific variables.

ALIGNMENT ON THE AVR MICROCONTROLLER

The AVR microcontroller does not have any alignment restrictions.

Byte order

The AVR microcontroller stores data in little-endian byte order.

In the little-endian byte order, the *least* significant byte is stored at the lowest address in memory. The *most* significant byte is stored at the highest address.

In the big-endian byte order, the *most* significant byte is stored at the lowest address in memory. The *least* significant byte is stored at the highest address. If you use the big-endian byte order, it might be necessary to use the `#pragma bitfields=reversed` directive to be compatible with code for other compilers and I/O register definitions of some devices, see *Bitfields*, page 284.

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	1
`unsigned short`	16 bits	0 to 65535	1
`signed int`	16 bits	-32768 to 32767	1
`unsigned int`	16 bits	0 to 65535	1
`signed long`	32 bits	-2^{31} to $2^{31}-1$	1
`unsigned long`	32 bits	0 to $2^{32}-1$	1
`signed long long`	64 bits	-2^{63} to $2^{63}-1$	1

Table 30: Integer types

Data type	Size	Range	Alignment
unsigned long long	64 bits	0 to 2^{64}-1	1

Table 30: Integer types (Continued)

Signed variables are represented using the two's complement form.

BOOL

The `bool` data type is supported by default in the C++ language. If you have enabled language extensions, the `bool` type can also be used in C source code if you include the file `stdbool.h`. This will also enable the boolean values `false` and `true`.

THE LONG LONG TYPE

The `long long` data type is supported with one restriction:

The CLIB runtime library does not support the `long long` type.

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.

Note: The IAR CLIB Library has only rudimentary support for `wchar_t`.

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 AVR, plain integer types are treated as signed.

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 bitfields=reversed`, bitfields are placed from the most significant bit to the least significant bit in each container. See *bitfields*, page 321.

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:
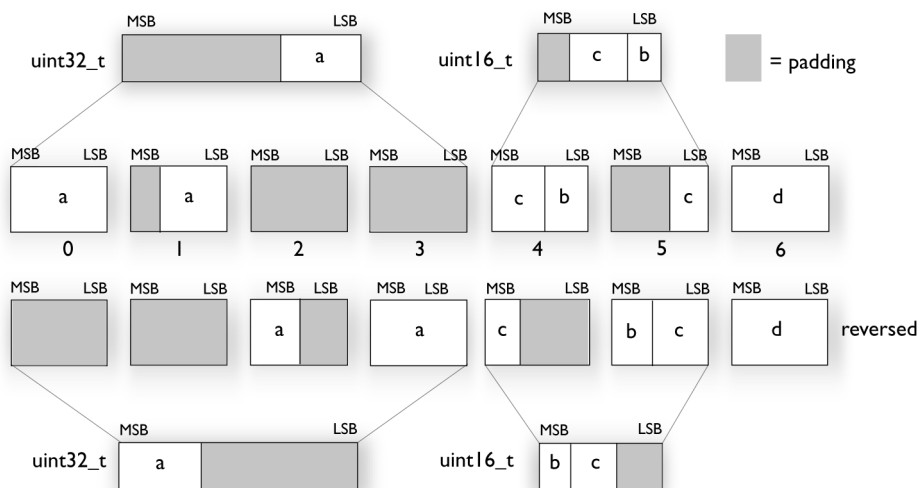
Basic data types—floating-point types

In the IAR C/C++ Compiler for AVR, 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 31: Floating-point types

Note: The size of double and long double depends on the --64bit_doubles option, see --*64bit_doubles*, page 240. 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:

31	30		23	22		0
S	Exponent			Mantissa		
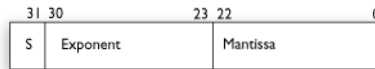
The exponent is 8 bits, and the mantissa is 23 bits.

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:

63	62		52	51		0
S	Exponent			Mantissa		
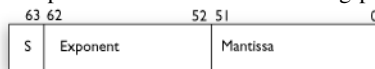
The exponent is 11 bits, and the mantissa is 52 bits.

The value of the number is:

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

The range of the number is at least:

±2.23E-308 to ±1.79E+308

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

REPRESENTATION OF SPECIAL FLOATING-POINT NUMBERS

This list describes the representation of special floating-point numbers:

- Zero is represented by zero mantissa and exponent. The sign bit signifies positive or negative zero.
- Infinity is represented by setting the exponent to the highest value and the mantissa to zero. The sign bit signifies positive or negative infinity.

- Not a number (NaN) is represented by setting the exponent to the highest positive value and the mantissa to a non-zero value. The value of the sign bit is ignored.

- Subnormal numbers are used for representing values smaller than what can be represented by normal values. The drawback is that the precision will decrease with smaller values. The exponent is set to 0 to signify that the number is subnormal, even though the number is treated as if the exponent was 1. Unlike normal numbers, subnormal numbers do not have an implicit 1 as the most significant bit (the MSB) of the mantissa. The value of a subnormal number is:

$$(-1)^S * 2^{(1-BIAS)} * 0.Mantissa$$

where BIAS is 127 and 1023 for 32-bit and 64-bit floating-point values, respectively.

Note: The IAR CLIB Library does not fully support the special cases of floating-point numbers, such as infinity, NaN, and subnormal numbers. A library function which gets one of these special cases of floating-point numbers as an argument might behave unexpectedly.

Pointer types

The compiler has two basic types of pointers: function pointers and data pointers.

FUNCTION POINTERS

The size of function pointers is always 16 or 24 bits, and they can address the entire memory. The internal representation of a function pointer is the actual address it refers to divided by two.

These function pointers are available:

Keyword	Address range	Pointer size	Description
__nearfunc	0–0x1FFFE	16 bits	Can be called from any part of the code memory, but must reside in the first 128 Kbytes of that space.
__farfunc	0–0x7FFFFE	24 bits	No restrictions on code placement.

Table 32: Function pointers

DATA POINTERS

Data pointers have three sizes: 8, 16, or 24 bits. These data pointers are available:

Keyword	Pointer size	Memory space	Index type	Address range
__tiny	8 bits	Data	signed char	0x–0xFF

Table 33: Data pointers

Keyword	Pointer size	Memory space	Index type	Address range
__near	16 bits	Data	signed int	0x0–0xFFFF
__far	24 bits	Data	signed int	0x0–0xFFFFFF (16-bit arithmetics)
__huge	24 bits	Data	signed long	0x0–0xFFFFFF
__tinyflash	8 bits	Code	signed char	0x0–0xFF
__flash	16 bits	Code	signed int	0x0–0xFFFF
__farflash	24 bits	Code	signed int	0x0–0xFFFFFF (16-bit arithmetic)
__hugeflash	24 bits	Code	signed long	0x0–0xFFFFFF
__eeprom	8 bits	EEPROM	signed char	0x0–0xFF
__eeprom	16 bits	EEPROM	signed int	0x0–0xFFFF
__generic	16 bits 24 bits	Data/Code	signed int signed long	The most significant bit (MSB) determines whether __generic points to CODE (1) or DATA (0). The small generic pointer is generated for the processor options –v0 and –v1.

Table 33: Data pointers (Continued)

Segmented data pointer comparison

Note that the result of using relational operators (<, <=, >, >=) on data pointers is only defined if the pointers point into the same object. For segmented data pointers, only the offset part of the pointer is compared when using these operators. For example:

```
void MyFunc(char __far * p, char __far * q)
{
  if (p == q)  /* Compares the entire pointers. */
    ...
  if (p < q) /* Compares only the low 16 bits of the pointers. */
    ...
}
```

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 via casting to the largest possible pointer that fits in the integer
- Casting a *data pointer* to a function pointer and vice versa is illegal
- Casting a *function pointer* to an integer type gives an undefined result
- Casting from a smaller pointer to a larger pointer is performed by zero extension
- Casting from a larger pointer to a smaller pointer 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 AVR, the type used for `size_t` depends on the processor option used:

Generic processor option	Typedef
-v0 and -v1	unsigned int
-v2, -v3, -v4, -v5, and -v6	unsigned long

Table 34: size_t typedef

When using the Large or Huge memory model, the typedef for `size_t` is `unsigned long int`.

Note that some data memory types might be able to accommodate larger, or only smaller, objects than the memory pointed to by default pointers. In this case, the type of the result of the `sizeof` operator could be a larger or smaller unsigned integer type. There exists a corresponding `size_t` typedef for each memory type, named after the memory type. In other words, `__near_size_t` for `__near` memory.

ptrdiff_t

`ptrdiff_t` is the signed integer type of the result of subtracting two pointers. This table shows the `typedef` of `ptrdiff_t` depending on the processor option:

Generic processor option	Typedef
–v0 and –v1	unsigned int
–v2, –v3, –v4, –v5, and –v6	unsigned long

Table 35: ptrdif_t typedef

Note that subtracting pointers other than default pointers could result in a smaller or larger integer type. In each case, this integer type is the signed integer variant of the corresponding size_t type.

Note: It is sometimes possible to create an object that is so large that the result of subtracting two pointers in that object is negative. See this example:

```
char buff[60000];          /* Assuming ptrdiff_t is a 16-bit */
char *p1 = buff;           /* signed integer type. */
char *p2 = buff + 60000;
ptrdiff_t diff = p2 - p1;  /* Result: -5536 */
```

intptr_t

intptr_t is a signed integer type large enough to contain a void *. In the IAR C/C++ Compiler for AVR, the type used for intptr_t is long int when using the Large or Huge memory model and int in all other cases.

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 inherit the alignment requirements of their members. In addition, the size of a struct is adjusted to allow arrays of aligned structure objects.

GENERAL LAYOUT

Members of a struct 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:
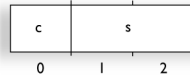
The alignment of the structure is 1 byte, and the size is 3 bytes.

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 AVR are described
below.

Rules for accesses

In the IAR C/C++ Compiler for AVR, 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 8-bit types.

These object types are treated in a special way:

Type of object	Treated as
Global register variables	Treated as non-triggering volatiles
__io declared variables located in 0x-0x1F	An assignment to a bitfield always generates a write access, in some cases also a read access. If only one bit is updated— set or cleared—the sci/cbi instructions are executed.

Table 36: Type of volatile accesses treated in a special way

For all combinations of object types not listed, only the rule that states that all accesses
are preserved applies.

DECLARING OBJECTS VOLATILE AND CONST

If you declare a volatile object const, it will be write-protected but it will still be
stored in RAM memory as the C standard specifies.

To store the object in read-only memory instead, but still make it possible to access it as a `const volatile` object, follow this example:

```
/* Header */
extern int const xVar;
#define x (*(int const volatile *) &xVar)

/* Source that uses x */
int DoSomething()
{
  return x;
}

/* Source that defines x */
#pragma segment = "FLASH"
int const xVar @ "FLASH" = 6;
```

The segment `FLASH` contains the initializers. They must be flashed manually 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.

If no ROM is available in the `DATA` memory space, use the compiler option `–y` to control the placement of constants. For more information, see *-y*, page 279.

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 *Segment 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 AVR 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 300. For information about how to use attributes to modify data, see the chapter *Data storage*. For information about how to use attributes to modify functions, see the chapter *Functions*.

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 250.

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 *function memory attributes*:

`__nearfunc`, `__farfunc`

Available *data memory attributes*:

`__tiny`, `__near`, `__far`, `__huge`, `__regvar`, `__eeprom`, `__tinyflash`, `__flash`, `__farflash`, `__hugeflash`, `__generic`, `__io`, and `__ext_io`.

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):

`__interrupt`, `__task`, `__version_1`, `__version_2`, `__version_3`, `__version_4`, `__x`, `__x_z`, `__z`, `__z_x`

Available *data type attributes*:

, `const`, `volatile`

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:

```
__near int i;
int __near j;
```

Both i and j are placed in near 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 __near * p;        /* pointer to integer in near memory */

int * __near p;        /* pointer in near memory */

__near int * p;        /* pointer in near 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 __near int d16_int;
d16_int *q1;
```

d16_int is a typedef for integers in near 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=__near
int * q2;
```

The variable q2 is placed in near memory.

For more examples of using memory attributes, see *More examples*, page 57.

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);
```

To declare a function pointer, use this syntax:

```
int (__nearfunc * fp) (double);
```

After this declaration, the function pointer `fp` points to near memory.

An easier way of specifying storage is to use type definitions:

```
typedef __nearfunc void FUNC_TYPE(int);
typedef FUNC_TYPE *FUNC_PTR_TYPE;
FUNC_TYPE func();
FUNC_PTR_TYPE funcptr;
```

Note that `#pragma type_attribute` can be used together with a `typedef` declaration.

OBJECT ATTRIBUTES

Normally, object attributes affect the internal functionality of functions and data objects, but not directly how the function is called or how the data is accessed. This means that an object attribute does not normally need to be present in the declaration of an object.

These object attributes are available:

- Object attributes that can be used for variables:

  `__no_alloc`, `__no_alloc16`, `__no_alloc_str`, `__no_alloc_str16`, `__no_init`, `__ro_placement`, `__no_runtime_init`

- Object attributes that can be used for functions and variables:

  `location`, `@`, `__root`

- Object attributes that can be used for functions:

  `__intrinsic`, `__monitor`, `__nested`, `__noreturn`, `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 210. For more information about `vector`, see *vector*, page 339.

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
__eeprom	Controls the storage of data objects in code memory space
__ext_io	Controls the storage of data objects in I/O memory space Supports I/O instructions; used for SFRs
__far	Controls the storage of data objects in data memory space
__farflash	Controls the storage of data objects in code memory space
__farfunc	Controls the storage of functions in code memory space
__flash	Controls the storage of data objects in code memory space
__generic	Declares a generic pointer
__huge	Controls the storage of data objects in data memory space
__hugeflash	Controls the storage of data objects in code memory space
__interrupt	Specifies interrupt functions
__intrinsic	Reserved for compiler internal use only
__io	Controls the storage of data objects in I/O memory space Supports I/O instructions; used for SFRs
__monitor	Specifies atomic execution of a function
__near	Controls the storage of data objects in data memory space
__nearfunc	Controls the storage of functions in code memory space
__nested	Implements a nested interrupt
__no_alloc, __no_alloc16	Makes a constant available in the execution file
__no_alloc_str, __no_alloc_str16	Makes a string literal available in the execution file
__no_init	Places a data object in non-volatile memory
__no_runtime_init	Declares initialized variables that are not initialized at runtime.

Table 37: Extended keywords summary

Extended keyword	Description
__noreturn	Informs the compiler that the function will not return
__raw	Prevents saving used registers in interrupt functions
__regvar	Places a data object in a register
__root	Ensures that a function or variable is included in the object code even if unused
__ro_placement	Places const volatile data in read-only memory.
__task	Relaxes the rules for preserving registers
__tiny	Controls the storage of data objects in data memory space
__tinyflash	Controls the storage of data objects in code memory space
__version_1	Specifies the old calling convention; available for backward compatibility
__version_2	Specifies the V2.10-V4.10B calling convention
__version_4	Specifies the default calling convention
__x	Places the first pointer of the parameter list in register X
__x_z	Places the first pointer of the parameter list in register X and the second one in register Z
__z	Places the first pointer of the parameter list in register Z
__z_x	Places the first pointer of the parameter list in register Z and the second one in register X

Table 37: Extended keywords summary (Continued)

Descriptions of extended keywords

This section gives detailed information about each extended keyword.

__eeprom

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 295.

Description

The __eeprom memory attribute overrides the default storage of variables given by the selected memory model and places individual initialized and non-initialized variables in the built-in EEPROM of the AVR microcontroller. These variables can be used like any other variable and provide a convenient way to access the built-in EEPROM.

You can also override the supplied support functions to make the `__eeprom` memory attribute access an EEPROM or flash memory that is placed externally but not on the normal data bus, for example on an I2C bus.

To do this, you must write a new EEPROM library routines file and include it in your project. The new file must use the same interface as the `eeprom.s90` file in the `avr\src\lib` directory (visible register use, the same entry points and the same semantics).

Variables declared `__eeprom` are initialized only when a downloadable linker output file is downloaded to the system, and not every time the system startup code is executed.

You can also use the `__eeprom` attribute to create a pointer explicitly pointing to an object located in the EEPROM memory.

Storage information	• Memory space: EEPROM memory space
	• Address range: `0-0xFF` (255 bytes) if `--eeprom_size` <= 256 bytes, and `0-0xFFFF` (64 Kbytes) if `--eeprom_size` > 256 bytes.
	• Maximum object size: 255 bytes if `--eeprom_size` <= 256 bytes, and 64 Kbytes-1 if `--eeprom_size` > 256 bytes.
	• Pointer size: 1byte if `--eeprom_size` <= 256 bytes, and 2 bytes if `--eeprom_size` > 256 bytes.

Example   `__eeprom int x;`

See also   *Memory types*, page 54 and *--eeprom_size*, page 251.

__ext_io

Syntax   Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 295.

Description   The `__ext_io` memory attribute overrides the default storage of variables given by the selected memory model and allows individual objects to be accessed by use of the special I/O instructions in the AVR microcontroller. The `__ext_io` memory attribute implies that the object is `__no_init` and `volatile`.

Your application may access the AVR I/O system by using the memory-mapped internal special function registers (SFRs). To access the AVR I/O system efficiently, the `__ext_io` memory attribute should be included in the code.

Storage information   • Memory space: Data memory space
          • Address range: `0x100-0xFFFF` (64 Kbytes)

- Maximum object size: 4 bytes.
- Pointer size: Pointers not allowed.

Example	`__ext_io int x;`
See also	*Memory types*, page 54.

__far

Syntax	Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 295.
Description	The `__far` memory attribute overrides the default storage of variables given by the selected memory model and places individual variables and constants in far memory. You can also use the `__far` attribute to create a pointer explicitly pointing to an object located in the far memory.
Storage information	● Memory space: Data memory space ● Address range: `0-0xFFFFFF` ● Maximum object size: 32 Kbytes-1. An object cannot cross a 64-Kbyte boundary. ● Pointer size: 3 bytes. Arithmetics is only performed on the two lower bytes, except comparison which is always performed on the entire 24-bit address.
Example	`__far int x;`
See also	*Memory types*, page 54.

__farflash

Syntax	Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 295.
Description	The `__farflash` memory attribute overrides the default storage of variables given by the selected memory model and places individual variables and constants in flash (code) memory. You can also use the `__farflash` attribute to create a pointer explicitly pointing to an object located in the flash (code) memory. **Note:** It is preferable to declare flash objects as `const`. The `__farflash` memory attribute is only available for AVR chips with more than 64 Kbytes of flash memory.
Storage information	● Memory space: Code memory space

- Address range: `0-0xFFFFFF`
- Maximum object size: 32 Kbytes-1. An object cannot cross a 64-Kbyte boundary.
- Pointer size: 3 bytes. Arithmetics is only performed on the two lower bytes, except comparison which is always performed on the entire 24-bit address.

Example	`__farflash int x;`
See also	*Memory types*, page 54.

__farfunc

Syntax

Follows the generic syntax rules for memory type attributes that can be used on functions, see *Type attributes*, page 295.

Description

The `__farfunc` memory attribute overrides the default storage of functions given by the selected code model and places individual functions in code memory. You can also use the `__farfunc` attribute to create a pointer explicitly pointing to a function located in `FARCODE` memory (`0-0xFFFFFF`).

The default code pointer for the `-v5` and `-v6` processor options is `__farfunc`, and it only affects the size of the function pointers.

Note that pointers with function memory attributes have restrictions in implicit and explicit casts when casting between pointers and also when casting between pointers and integer types.

It is possible to call a `__nearfunc` function from a `__farfunc` function and vice versa. Only the size of the function pointer is affected.

Storage information

- Memory space: Code memory space
- Address range: `0-0x7FFFFE` (8 Mbytes)
- Pointer size: 3 bytes

Example

`__farfunc void myfunction(void);`

See also

Function storage, page 63.

__flash

Syntax

Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 295.

Description   The `__flash` memory attribute overrides the default storage of variables given by the selected memory model and places individual variables and constants in flash (code) memory.

Note that it is preferable to declare flash objects as constant. The `__flash` keyword is available for most AVR processors.

Because the AVR microcontrollers have only a small amount of on-board RAM, this memory should not be wasted on data that could be stored in flash memory (of which there is much more). However, because of the architecture of the processor, a default pointer cannot access the flash memory. The `__flash` keyword is used to identify that the constant should be stored in flash memory.

A header file called `pgmspace.h` is installed in the `avr\inc` directory, to provide some standard C library functions for taking strings stored in flash memory as arguments.

You can also use the `__flash` attribute to create a pointer explicitly pointing to an object located in the flash (code) memory.

Note: The `__flash` option cannot be used in a reduced ATtiny core.

Storage information   ● Memory space: Code memory space

● Address range: `0-0xFFFF` (64 Kbytes)

● Maximum object size: 32 Kbytes-1.

● Pointer size: 2 bytes.

Example   A program defines a couple of strings that are stored in flash memory:

```
__flash char str1[] = "Message 1";
__flash char str2[] = "Message 2";
```

The program creates a `__flash` pointer to point to one of these strings, and assigns it to `str1`:

```
char __flash *msg;
  msg=str1;
```

Using the `strcpy_P` function declared in `pgmspace.h`, a string in flash memory can be copied to another string in RAM as follows:

```
strcpy_P(dest, msg);
```

See also   *Memory types*, page 54.

__generic

Syntax	Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 295.

Description

The `__generic` pointer type attribute declares a generic pointer that can point to objects in both code and data space. The size of the generic pointer depends on which processor option is used.

The most significant bit (MSB) determines whether `__generic` points to Code (MSB=1) or Data (MSB=0).

It is not possible to place objects in a generic memory area, only to point to it. When using generic pointers, make sure that objects that have been declared `__far` and `__huge` are located in the range `0x0-0x7FFFFF`. Objects may still be placed in the entire data address space, but a generic pointer cannot point to objects in the upper half of the data address space.

The `__generic` keyword cannot be used with the `#pragma type_attribute` directive for a pointer.

Access through a `__generic` pointer is implemented with inline code. Because this type of access is slow and generates a lot of code, `__generic` pointers should be avoided when possible.

Storage information

- Memory space: Data or code memory space
- Address range: `0-0x7FFF` (32 Kbytes) for the processor options `-v0` and `-v1`, `0-07FFFFF` (8 Mbytes) for the processor options `-v2` to `-v6`
- Maximum object size: 32 Kbytes-1 if the pointer size is 2 bytes, and 8 Mbytes-1 if the pointer size is 3 bytes.
- Generic pointer size: 1 + 15 bits when the processor options `-v0` and `-v1` are used. 1 + 23 bits when the processor options `-v2` to `-v6` are used.

See also

Memory types, page 54.

__huge

Syntax	Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 295.

Description

The `__huge` memory attribute overrides the default storage of variables given by the selected memory model and places individual variables and constants in huge memory.

You can also use the `__huge` attribute to create a pointer explicitly pointing to an object located in the huge memory.

Storage information	● Memory space: Data memory space
	● Address range: `0-0xFFFFFF` (16 Mbytes)
	● Maximum object size: 16 Mbytes-1.
	● Pointer size: 3 bytes.

Example   `__huge int x;`

See also   *Memory types*, page 54.

__hugeflash

Syntax   Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 295.

Description   The `__hugeflash` memory attribute overrides the default storage of variables given by the selected memory model and places individual variables and constants in flash (code) memory. You can also use the `__hugeflash` attribute to create a pointer explicitly pointing to an object located in the flash (code) memory.

Note that it is preferable to declare flash objects as constant. The `__hugeflash` memory attribute is only available for AVR microcontrollers with at least 64 Kbytes of flash memory.

Storage information   ● Memory space: Code memory space
● Address range: `0-0x7FFFFF` (8 Mbytes)
● Maximum object size: 8 Mbytes-1.
● Pointer size: 3 bytes.

Example   `__hugeflash int x;`

See also   *Memory types*, page 54.

__interrupt

Syntax	See *Syntax for type attributes used on functions*, page 297.

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 `iodevice.h`, where *device* corresponds to the selected device, contains predefined names for the existing interrupt vectors.

To make sure that the interrupt handler executes as fast as possible, you should compile it with `-Ohs`, or use `#pragma optimize=speed` if the module is compiled with another optimization goal.

Example
```
#pragma vector=0x14
__interrupt void my_interrupt_handler(void);
```

See also   *Interrupt functions*, page 64, *vector*, page 339, and *INTVEC*, page 385.

__intrinsic

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

__io

Syntax   Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 295.

Description   The `__io` memory attribute overrides the default storage of variables given by the selected memory model and places individual variables and constants in I/O memory.

Storage information
- Memory space: Data memory space
- Address range: `0-0xFFFF` (64 Kbytes)
- Maximum object size: 4 bytes (32 bits).
- Pointer size: Pointers not allowed.

Example   `__io int x;`

See also	*Memory types*, page 54.

__monitor

Syntax	See *Syntax for object attributes*, page 298.
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 65. 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 348, respectively.

__near

Syntax	Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 295.
Description	The __near memory attribute overrides the default storage of variables given by the selected memory model and places individual variables and constants in near memory. You can also use the __near attribute to create a pointer explicitly pointing to an object located in the near memory.
Storage information	• Memory space: Data memory space
	• Address range: `0-0xFFFF` (64 Kbytes)
	• Maximum object size: 32 Kbytes-1.
	• Pointer size: 2 bytes.
Example	`__near int x;`
See also	*Memory types*, page 54.

__nearfunc

Syntax	Follows the generic syntax rules for memory type attributes that can be used on functions, see *Type attributes*, page 295.

Description   The `__nearfunc` memory attribute overrides the default storage of functions given by
                  the selected code model and places individual functions in near memory.

                  Functions declared `__nearfunc` can be called from the entire code memory area, but
                  must reside in the first 128 Kbytes of the code memory.

                  The default for the `-v0` to `-v4` processor options is `__nearfunc`, and it only affects the
                  size of the function pointers.

                  Note that pointers with function memory attributes have restrictions in implicit and
                  explicit casts when casting between pointers and also when casting between pointers
                  and integer types.

                  It is possible to call a `__nearfunc` function from a `__farfunc` function and vice versa.
                  Only the size of the function pointer is affected.

                  You can also use the `__nearfunc` attribute to create a pointer explicitly pointing to a
                  function located in CODE memory (`0-0xFFFF`).

Storage information   ● Memory space: Code memory space
                      ● Address range: `0-0x1FFFE` (128 Kbytes)
                      ● Pointer size: 2 bytes

Example   `__nearfunc void myfunction(void);`

See also   *Function storage*, page 63.

__nested

Syntax   Follows the generic syntax rules for type attributes that can be used on functions, see
                  Type attributes, page 295.

Description   Use the `__nested` keyword to implement a nested interrupt, in other words a new
                  interrupt can be served inside an interrupt function. A nested interrupt service routine
                  acts like a normal interrupt service routine except that it sets the interrupt enable bit
                  before any registers are saved.

Example   `__nested __interrupt void myInterruptFunction()`

__no_alloc, __no_alloc16

Syntax   See *Syntax for object attributes*, page 298.

Description   Use the `__no_alloc` or `__no_alloc16` object attribute on a constant to make the constant available in the executable file without occupying any space in the linked application.

You cannot access the contents of such a constant from your application. You can take its address, which is an integer offset to the segment of the constant. The type of the offset is `unsigned long` when `__no_alloc` is used, and `unsigned short` when `__no_alloc16` is used.

Example   `__no_alloc const struct MyData my_data @ "XXX" = {...};`

See also   *__no_alloc_str, __no_alloc_str16*, page 310.

__no_alloc_str, __no_alloc_str16

Syntax   `__no_alloc_str(`*string_literal* `@ `*segment*`)`

and

`__no_alloc_str16(`*string_literal* `@ `*segment*`)`

where

string_literal	The string literal that you want to make available in the executable file.
segment	The name of the segment to place the string literal in.

Description   Use the `__no_alloc_str` or `__no_alloc_str16` operators to make string literals available in the executable file without occupying any space in the linked application.

The value of the expression is the offset of the string literal in the segment. For `__no_alloc_str`, the type of the offset is `unsigned long`. For `__no_alloc_str16`, the type of the offset is `unsigned short`.

Example

```
#define MYSEG "YYY"
#define X(str) __no_alloc_str(str @ MYSEG)

extern void dbg_printf(unsigned long fmt, ...)

#define DBGPRINTF(fmt, ...) dbg_printf(X(fmt), __VA_ARGS__)

void
foo(int i, double d)
{
  DBGPRINTF("The value of i is: %d, the value of d is: %f",i,d);
}
```

Depending on your debugger and the runtime support, this could produce trace output on the host computer. Note that there is no such runtime support in C-SPY, unless you use an external plugin module.

See also
__no_alloc, __no_alloc16, page 310.

__no_init

Syntax
See *Syntax for object attributes*, page 298.

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 223.

__no_runtime_init

Syntax
Follows the generic syntax rules for object attributes, see *Object attributes*, page 298.

Description
The `__no_runtime_init` keyword is used for declaring initialized variables for which the initialization is performed when programming the device. These variables are not initialized at runtime during system startup.

Note: The `__no_runtime_init` keyword cannot be used in combination with the `typedef` keyword.

Example
`__no_runtime_init int myarray[10];`

__noreturn

Syntax   See *Syntax for object attributes*, page 298.

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);

__raw

Syntax   See *Syntax for object attributes*, page 298.

Description   This keyword prevents saving used registers in interrupt functions.

Example   __raw __interrupt void my_interrupt_function()

__regvar

Syntax   Follows the generic syntax rules for type attributes that can be used on data, see *Type
                attributes*, page 295.

Description   The __regvar extended keyword is used for declaring that a global or static variable
                should be placed permanently in the specified register or registers. The registers R4-R15
                can be used for this purpose, provided that they have been locked with the
                --lock_regs compiler option.

                Note:

                ● An object declared __regvar cannot have an initial value.
                ● The__regvar option cannot be used in a reduced ATtiny core.

Storage information   ● Memory space: Data memory space
                ● Address range: 4-F
                ● Maximum object size: 4 bytes (32 bits).
                ● Pointer size: Pointers not allowed.

Example
: `__regvar __no_init int counter @ 14;`

This will create the 16-bit integer variable counter, which will always be available in `R15:R14`. At least two registers must be locked.

See also
: *--memory_model, -m*, page 258, *--lock_regs*, page 258, and *Preserved registers*, page 158.

__root

Syntax
: See *Syntax for object attributes*, page 298.

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 modules, segments, and the link process, see the *IAR Linker and Library Tools Reference Guide.*

__ro_placement

Syntax
: See *Syntax for object attributes*, page 298.

Description
: The `__ro_placement` attribute specifies that a data object should be placed in read-only memory. There are two cases where you might want to use this object attribute:

● Data objects declared `const volatile` are by default placed in read-write memory. Use the `__ro_placement` object attribute to place the data object in read-only memory instead.

● In C++, a data object declared `const` and that needs dynamic initialization is placed in read-write memory and initialized at system startup. If you use the `__ro_placement` object attribute, the compiler will give an error message if the data object needs dynamic initialization.

You can only use the `__ro_placement` object attribute on `const` objects.

In some cases (primarily involving simple constructors), the compiler will be able to optimize C++ dynamic initialization of a data object into static initialization. In that case no error message will be issued for the object.

Example   `__ro_placement const volatile int x = 10;`

__task

Syntax   See *Syntax for type attributes used on functions*, page 297.

Description   This keyword allows functions to relax the rules for preserving registers. Typically, the keyword is used on the start function for a task in an RTOS.

By default, functions save the contents of used preserved registers on the stack upon entry, and restore them at exit. Functions that are declared `__task` do not save all registers, and therefore require less stack space.

Because a function declared `__task` can corrupt registers that are needed by the calling function, you should only use `__task` on functions that do not return or call such a function from assembler code.

The function `main` can be declared `__task`, unless it is explicitly called from the application. In real-time applications with more than one task, the root function of each task can be declared `__task`.

Example   `__task void my_handler(void);`

__tiny

Syntax   Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 295.

Description   The `__tiny` memory attribute overrides the default storage of variables given by the selected memory model and places individual variables and constants in tiny memory. You can also use the `__tiny` attribute to create a pointer explicitly pointing to an object located in the tiny memory.

Storage information
- Memory space: Data memory space
- Address range: `0-0xFF` (256 bytes)
- Maximum object size: 127 bytes.
- Pointer size: 1 byte.

Example   `__tiny int x;`

See also   *Memory types*, page 54.

__tinyflash

Syntax
Follows the generic syntax rules for memory type attributes that can be used on data objects, see *Type attributes*, page 295.

Description
The `__tinyflash` memory attribute overrides the default storage of variables given by the selected memory model and places individual variables and constants in flash (code) memory. You can also use the `__tinyflash` attribute to create a pointer explicitly pointing to an object located in the flash (code) memory.

Storage information
● Memory space: Code memory space
● Address range: `0-0xFF` (256 bytes)
● Maximum object size: 127 bytes.
● Pointer size: 1 byte.

Example
`__tinyflash int x;`

See also
Memory types, page 54.

__version_1

Syntax
Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 295.

Description
The `__version_1` keyword is available for backward compatibility of the interface for calling assembler routines from C. It makes a function use the calling convention of the A90 IAR C Compiler instead of the default calling convention. The `__version_1` calling convention is preferred when calling assembler functions from C.

Example
`__version_1 int func(int arg1, double arg2)`

See also
Calling convention, page 155.

__version_2

Syntax
Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 295.

Description
The `__version_2` keyword is available for backward compatibility of the interface for calling assembler routines from C. The keyword makes a function use the V2.10-V4.10B calling convention instead of the default calling convention.

Example
```
__version_2 int func(int arg1, double arg2)
```

See also
Calling convention, page 155.

__version_4

Syntax
Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 295.

Description
The `__version_4` keyword specifies the default calling convention.

Example
```
__version_4 int func(int arg1, double arg2)
```

See also
Calling convention, page 155.

__x

Syntax
Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 295.

Description
The `__x` keyword places the first pointer of the parameter list in the register X. This keyword is used by parts of the runtime library.

Note: This keyword can give very efficient code in a local perspective, but should be used with care because it changes the calling convention and might have a negative effect on the size of the entire application.

__x_z

Syntax
Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 295.

Description
The `__x_z` keyword places the first pointer of the parameter list in the register X and the second one in register Z. This keyword is used by parts of the runtime library.

Note: This keyword can give very efficient code in a local perspective, but should be used with care because it changes the calling convention and might have a negative effect on the size of the entire application.

__z

Syntax	Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 295.
Description	The __z keyword places the first pointer of the parameter list in the register z. This keyword is used by parts of the runtime library.
	Note: This keyword can give very efficient code in a local perspective, but should be used with care because it changes the calling convention and might have a negative effect on the size of the entire application.

__z_x

Syntax	Follows the generic syntax rules for type attributes that can be used on functions, see *Type attributes*, page 295.
Description	The __z_x keyword places the first pointer of the parameter list in the register z and the second one in register x. This keyword is used by parts of the runtime library.
	Note: This keyword can give very efficient code in a local perspective, but should be used with care because it changes the calling convention and might have a negative effect on the size of the entire application.

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
`basic_template_matching`	Makes a template function fully memory-attribute aware.
`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.
`constseg`	Places constant variables in a named segment.
`cstat_disable`	See the *C-STAT Static Analysis Guide*.
`cstat_enable`	See the *C-STAT Static Analysis Guide*.
`cstat_restore`	See the *C-STAT Static Analysis Guide*.
`cstat_suppress`	See the *C-STAT Static Analysis Guide*.
`data_alignment`	Gives a variable a higher (more strict) alignment.
`dataseg`	Places variables in a named segment.
`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.

Table 38: Pragma directives summary

Pragma directive	Description
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.
language	Controls the IAR Systems language extensions.
location	Specifies the absolute address of a variable, or places groups of functions or variables in named segments.
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.
__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	This directive is an alias for #pragma segment.
segment	Declares a segment name to be used by intrinsic functions.
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.
vector	Specifies the vector of an interrupt or trap function.

Table 38: Pragma directives summary (Continued)

Pragma directive	Description
weak	Makes a definition a weak definition, or creates a weak alias for a function or a variable.
type_attribute	Adds type attributes to a declaration or to definitions.

Table 38: Pragma directives summary (Continued)

Note: For portability reasons, see also *Recognized pragma directives (6.10.6)*, page 407.

Descriptions of pragma directives

This section gives detailed information about each pragma directive.

basic_template_matching

Syntax	`#pragma basic_template_matching`

Description   Use this pragma directive in front of a template function declaration to make the function fully memory-attribute aware, in the rare cases where this is useful. That template function will then match the template without the modifications, see *Templates and data memory attributes*, page 184.

Example

```
#pragma basic_template_matching
template<typename T> void fun(T *);

void MyF()
{
  fun((int __near *) 0); // T = int __near
}
```

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 284.

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 79

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 79

constseg

Syntax	`#pragma constseg=[`*`__memoryattribute`* `]{`*`SEGMENT_NAME`*`|default}`

Parameters

`__memoryattribute`	An optional memory attribute denoting in what memory the segment will be placed; if not specified, default memory is used.
`SEGMENT_NAME`	A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.
`default`	Uses the default segment for constants.

Description	Use this pragma directive to place constant variables in a named segment. The segment name cannot be a segment name predefined for use by the compiler and linker. The setting remains active until you turn it off again with the `#pragma constseg=default` directive.
| Example | ```
#pragma constseg=__huge MY_CONSTANTS
const int factorySettings[] = {42, 15, -128, 0};
#pragma constseg=default
``` |

data_alignment

Syntax	`#pragma data_alignment=`*`expression`*

Parameters

`expression`	A constant which must be a power of two (1, 2, 4, etc.).

Description   Use this pragma directive to give the immediately following variable a higher (more strict) alignment of the start address than it would otherwise have. This directive can be used on variables with static and automatic storage duration.

When you use this directive on variables with automatic storage duration, there is an upper limit on the allowed alignment for each function, determined by the calling convention used.

Note: Normally, the size of a variable is a multiple of its alignment. The `data_alignment` directive only affects the alignment of the variable's start address, and not its size, and can thus be used for creating situations where the size is not a multiple of the alignment.

dataseg

Syntax   `#pragma dataseg=[__memoryattribute]{SEGMENT_NAME|default}`

Parameters

`__memoryattribute`	An optional memory attribute denoting in what memory the segment will be placed; if not specified, default memory is used.
`SEGMENT_NAME`	A user-defined segment name; cannot be a segment name predefined for use by the compiler and linker.
`default`	Uses the default segment.

Description   Use this pragma directive to place variables in a named segment. The segment name cannot be a segment name predefined for use by the compiler and linker. The variable will not be initialized at startup, and can for this reason not have an initializer, which means it must be declared `__no_init`. The setting remains active until you turn it off again with the `#pragma dataseg=default` directive.

Example
```
#pragma dataseg=__huge MY_SECTIONSEGMENT
__no_init char myBuffer[1000];
#pragma dataseg=default
```

default_function_attributes

Syntax   `#pragma default_function_attributes=[ attribute...]`

where *attribute* can be:

type_attribute
object_attribute

```
@ segment_name
```

Parameters

`type_attribute`	See *Type attributes*, page 295.
`object_attribute`	See *Object attributes*, page 298.
`@ segment_name`	See *Data and function placement in segments*, page 212.

Description

Use this pragma directive to set default segment 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 segment MYSEG" */
#pragma default_function_attributes = @ "MYSEG"
int fun1(int x) { return x + 1; }
int fun2(int x) { return x - 1;
/* Stop placing functions into MYSEG */
#pragma default_function_attributes =
```

has the same effect as:

```
int fun1(int x) @ "MYSEG" { return x + 1; }
int fun2(int x) @ "MYSEG" { return x - 1; }
```

See also

location, page 331

object_attribute, page 332

type_attribute, page 338

default_variable_attributes

Syntax

```
#pragma default_variable_attributes=[ attribute...]
```

where `attribute` can be:

```
type_attribute
object_attribute
@ segment_name
```

325

Parameters

type_attribute	See *Type attributes*, page 295.
object_attributes	See *Object attributes*, page 298.
@ *segment_name*	See *Data and function placement in segments*, page 212.

Description

Use this pragma directive to set default segment placement, type attributes, and object attributes for declarations and definitions of variables with static storage duration. The default settings are only used for declarations and definitions that do not specify type or object attributes or location in some other way.

Specifying a default_variable_attributes pragma directive with no attributes restores the initial state of no such defaults being applied to variables with static storage duration.

Example

```
/* Place following variables in segment MYSEG" */
#pragma default_variable_attributes = @ "MYSEG"
int var1 = 42;
int var2 = 17;
/* Stop placing variables into MYSEG */
#pragma default_variable_attributes =
```

has the same effect as:

```
int var1 @ "MYSEG" = 42;
int var2 @ "MYSEG" = 17;
```

See also

location, page 331

object_attribute, page 332

type_attribute, page 338

diag_default

Syntax

```
#pragma diag_default=tag[,tag,...]
```

Parameters

tag	The number of a diagnostic message, for example the message number Pe177.

Description

Use this pragma directive to change the severity level back to the default, or to the severity level defined on the command line by any of the options --diag_error, --diag_remark, --diag_suppress, or --diag_warnings, for the diagnostic

messages specified with the tags. This level remains in effect until changed by another diagnostic-level pragma directive.

See also   *Diagnostics*, page 231.

diag_error

Syntax   `#pragma diag_error=`*tag*`[,`*tag*`,...]`

Parameters

    tag   The number of a diagnostic message, for example the message number `Pe177`.

Description   Use this pragma directive to change the severity level to `error` for the specified diagnostics. This level remains in effect until changed by another diagnostic-level pragma directive.

See also   *Diagnostics*, page 231.

diag_remark

Syntax   `#pragma diag_remark=`*tag*`[,`*tag*`,...]`

Parameters

    tag   The number of a diagnostic message, for example the message number `Pe177`.

Description   Use this pragma directive to change the severity level to `remark` for the specified diagnostic messages. This level remains in effect until changed by another diagnostic-level pragma directive.

See also   *Diagnostics*, page 231.

diag_suppress

Syntax   `#pragma diag_suppress=`*tag*`[,`*tag*`,...]`

Parameters

    tag   The number of a diagnostic message, for example the message number `Pe117`.

Description | Use this pragma directive to suppress the specified diagnostic messages. This level remains in effect until changed by another diagnostic-level pragma directive.

See also | *Diagnostics*, page 231.

diag_warning

Syntax | `#pragma diag_warning=tag[,tag,...]`

Parameters

tag | The number of a diagnostic message, for example the message number `Pe826`.

Description | Use this pragma directive to change the severity level to `warning` for the specified diagnostic messages. This level remains in effect until changed by another diagnostic-level pragma directive.

See also | *Diagnostics*, page 231.

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 228.

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 67.

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 250 and *--strict*, page 273.

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 segment name; cannot be a segment name predefined for use by the compiler and linker.

Description   Use this pragma directive to specify the location—the absolute address—of the global or static variable whose declaration follows the pragma directive. The variable must be declared either `__no_init` or `const`. Alternatively, the directive can take a string specifying a segment for placing either a variable or a function whose declaration follows the pragma directive. Do not place variables that would normally be in different segments (for example, variables declared as `__no_init` and variables declared as `const`) in the same named segment.

Example
```
#pragma location=0x2000
__no_init volatile char PORT1; /* PORT1 is located at address
                                     0x2000 */

#pragma segment="FLASH"
#pragma location="FLASH"
__no_init char PORT1; /* PORT1 is located in segment 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 segment */
```
Note: This still places the variables in DATA memory, and not in CODE.

See also   *Controlling data and function placement in memory*, page 210 and *Placing user-defined segments*, page 91.

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 298.
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 295. |

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_crosscall, disables interprocedural cross call
	no_crossjump, disables interprocedural cross jump
	no_cse, disables common subexpression elimination
	no_inline, disables function inlining
	no_relays, disables interprocedural relays
	no_tbaa, disables type-based alias analysis
	no_unroll, disables loop unrolling

Description

Use this pragma directive to decrease the optimization level, or to turn off some specific optimizations. This pragma directive only affects the function that follows immediately after the directive.

The parameters size, balanced, speed, and no_size_constraints only have effect on the high optimization level and only one of them can be used as it is not possible to optimize for speed and size at the same time. It is also not possible to use preprocessor macros embedded in this pragma directive. Any such macro will not be expanded by the preprocessor.

Note: If you use the #pragma optimize directive to specify an optimization level that is higher than the optimization level you specify using a compiler option, the pragma directive is ignored.

Example

```
#pragma optimize=speed
int SmallAndUsedOften()
{
  /* Do something here. */
}

#pragma optimize=size
int BigAndSeldomUsed()
{
  /* Do something here. */
}
```

See also

Fine-tuning enabled transformations, page 216.

__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 268.

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 segment 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.

See also   *Inline assembler*, page 151

rtmodel

Syntax   `#pragma rtmodel="key","value"`

Parameters

`"key"`	A text string that specifies the runtime model attribute.
`"value"`	A text string that specifies the value of the runtime model attribute. Using the special value * is equivalent to not defining the attribute at all.

Description

Use this pragma directive to add a runtime model attribute to a module, which can be used by the linker to check consistency between modules.

This pragma directive is useful for enforcing consistency between modules. All modules that are linked together and define the same runtime attribute key must have the same value for the corresponding key, or the special value *. It can, however, be useful to state explicitly that the module can handle any runtime model.

A module can have several runtime model definitions.

Note: The predefined compiler runtime model attributes start with a double underscore. To avoid confusion, this style must not be used in the user-defined attributes.

Example   `#pragma rtmodel="I2C","ENABLED"`

The linker will generate an error if a module that contains this definition is linked with a module that does not have the corresponding runtime model attributes defined.

__scanf_args

Syntax

```
#pragma __scanf_args
```

Description

Use this pragma directive on a function with a scanf-style format string. For any call to that function, the compiler verifies that the argument to each conversion specifier (for example %d) is syntactically correct.

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;
}
```

segment

Syntax

```
#pragma segment="NAME" [__memoryattribute]
```

alias

```
#pragma section="NAME" [__memoryattribute] [align]
```

Parameters

NAME	The name of the segment.
__memoryattribute	An optional memory attribute identifying the memory the segment will be placed in; if not specified, default memory is used.
align	Specifies an alignment for the segment. The value must be a constant integer expression to the power of two.

Description

Use this pragma directive to define a segment name that can be used by the segment operators __segment_begin, __segment_end, and __segment_size. All segment declarations for a specific segment must have the same memory type attribute and alignment.

The *align* and the *__memoryattribute* parameters are only relevant when used together with the segment operators __segment_begin, __segment_end, and

__segment_size. If you consider using *align* on an individual variable to achieve a higher alignment, you must instead use the #pragma data_alignment directive.

If an optional memory attribute is used, the return type of the segment operators __segment_begin and __segment_end is:

```
void __memoryattribute *.
```

Note: To place variables or functions in a specific segment, use the #pragma location directive or the @ operator.

Example	#pragma segment="MYNEAR" __near 4
See also	*Dedicated segment operators*, page 171 and the chapters *Linking overview* and *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 295.

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 `__near` is defined:

```
#pragma type_attribute=__near
int x;
```

This declaration, which uses extended keywords, is equivalent:

```
__near 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.
● 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`.

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
`__delay_cycles`	Inserts a time delay
`__DES_decryption`	Decrypts according to Digital Encryption Standard
`__DES_encryption`	Encrypts according to Digital Encryption Standard
`__disable_interrupt`	Disables interrupts
`__enable_interrupt`	Enables interrupts
`__extended_load_program_memory`	Returns one byte from code memory
`__fractional_multiply_signed`	Generates an `FMULS` instruction
`__fractional_multiply_signed_with_unsigned`	Generates an `FMULSU` instruction
`__fractional_multiply_unsigned`	Generates an `FMUL` instruction
`__get_interrupt_state`	Returns the interrupt state
`__indirect_jump_to`	Generates an `IJMP` instruction
`__insert_opcode`	Assigns a value to a processor register
`__lac`	Provides access to the `LAC` instruction
`__las`	Provides access to the `LAS` instruction
`__lat`	Provides access to the `LAT` instruction
`__load_program_memory`	Returns one byte from program memory
`__multiply_signed`	Generates a `MULS` instruction

Table 39: Intrinsic functions summary

Intrinsic function	Description
`__multiply_signed_with_unsigned`	Generates a `MULSU` instruction
`__multiply_unsigned`	Generates a `MUL` instruction
`__no_operation`	Inserts a instruction
`__require`	Sets a constant literal as required
`__restore_interrupt`	Restores the interrupt flag
`__reverse`	Reverses the byte order of a value
`__save_interrupt`	Saves the state of the interrupt flag
`__set_interrupt_state`	Restores the interrupt state
`__sleep`	Inserts a `SLEEP` instruction
`__swap_nibbles`	Swaps bit 0-3 with bit 4-7
`__watchdog_reset`	Inserts a watchdog reset instruction
`__xch`	Provides access to the `XCH` instruction

Table 39: Intrinsic functions summary (Continued)

Descriptions of intrinsic functions

This section gives reference information about each intrinsic function.

__delay_cycles

Syntax

```
void __delay_cycles(unsigned long int);
```

Description

Makes the compiler generate code that takes the given amount of cycles to perform, that is, it inserts a time delay that lasts the specified number of cycles.

The specified value must be a constant integer expression and not an expression that is evaluated at runtime.

__DES_decryption

Syntax

```
unsigned long long __DES_decryption(unsigned long long data,
unsigned long long key);
```

where:

data   The data to decrypt

key	The key to decrypt against

Description   Decrypts according to the Digital Encryption Standard (DES). `__DES_decryption` performs a DES decryption of *data* against *key* and returns the decrypted data.

Note: This intrinsic function is available only for cores with support for the DES instruction.

__DES_encryption

Syntax
```
unsigned long long __DES_encryption(unsigned long long data,
unsigned long long key);
```
where:

data	The data to encrypt
key	The key to encrypt against

Description   Encrypts according to the Digital Encryption Standard (DES). `__DES_encryption` performs a DES encryption of *data* against *key* and returns the encrypted data.

This intrinsic function is available only for cores with support for the DES instruction.

__disable_interrupt

Syntax
```
void __disable_interrupt(void);
```

Description   Disables interrupts by inserting the instruction.

__enable_interrupt

Syntax
```
void __enable_interrupt(void);
```

Description   Enables interrupts by inserting the instruction.

__extended_load_program_memory

Syntax
```
unsigned char __extended_load_program_memory(unsigned char
__farflash *);
```

Description   Returns one byte from code memory.

Use this intrinsic function to access constant data in code memory.

__fractional_multiply_signed

Syntax
```
signed int __fractional_multiply_signed(signed char, signed
char);
```

Description   Generates an FMULS instruction.

__fractional_multiply_signed_with_unsigned

Syntax
```
signed int __fractional_multiply_signed_with_unsigned(signed
char, unsigned char);
```

Description   Generates an FMULSU instruction.

__fractional_multiply_unsigned

Syntax
```
unsigned int __fractional_multiply_unsigned(unsigned char,
unsigned char);
```

Description   Generates an FMUL instruction.

__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`.

__indirect_jump_to

Syntax

```
void __indirect_jump_to(unsigned long);
```

Description

Jumps to the address specified by the argument by using the `IJMP` or `EIJMP` instruction, depending on the generic processor option:

IJMP	-v0 to -v4
EIJMP	-v5 and -v6

__insert_opcode

Syntax

```
void __insert_opcode(unsigned short);
```

Description

Inserts a `DW unsigned` directive.

__lac

Syntax

```
unsigned char __lac(unsigned char, unsigned char *);
```

Description

Provides access to the `LAC` (Load And Clear) instruction.

__las

Syntax   `unsigned char __las(unsigned char, unsigned char *);`

Description   Provides access to the LAS (Load And Set) instruction.

__lat

Syntax   `unsigned char __lat(unsigned char, unsigned char *);`

Description   Provides access to the LAT (Load And Toggle) instruction.

__load_program_memory

Syntax   `unsigned char __load_program_memory(unsigned char __flash *);`

Description   Returns one byte from code memory. The constants must be placed within the first 64 Kbytes of memory.

__multiply_signed

Syntax   `signed int __multiply_signed(signed char, signed char);`

Description   Generates a MULS instruction.

__multiply_signed_with_unsigned

Syntax   `signed int __multiply_signed_with_unsigned(signed char, unsigned char);`

Description   Generates a MULSU instruction.

__multiply_unsigned

Syntax   `unsigned int __multiply_unsigned(unsigned char, unsigned char);`

Description   Generates a MUL instruction.

__no_operation

Syntax   `void __no_operation(void);`

Description   Inserts a `NOP` instruction.

__require

Syntax   `void __require(void *);`

Description   Sets a constant literal as required.

One of the prominent features of the IAR XLINK Linker is its ability to strip away anything that is not needed. This is a very good feature because it reduces the resulting code size to a minimum. However, in some situations you may want to be able to explicitly include a piece of code or a variable even though it is not directly used.

The argument to `__require` could be a variable, a function name, or an exported assembler label. It must, however, be a constant literal. The label referred to will be treated as if it would be used at the location of the `__require` call.

Example   In this example, the copyright message will be included in the generated binary file even though it is not directly used.

```
#include <intrinsics.h>
char copyright[] = "Copyright 2011 by XXXX";
void main(void)
{
  __require(copyright);
  [... the rest of the program ...]
}
```

__restore_interrupt

Syntax   `void __restore_interrupt(unsigned char oldState);`

Description   Restores the interrupt flag to the specified state.

Note: The value of `oldState` must be the result of a call to the `__save_interrupt` intrinsic function.

__reverse

Syntax

```
unsigned int __reverse(unsigned int);
```

Description

Reverses the byte order of the value given as parameter. Avoid using __reverse in complex expressions as it might introduce extra register copying.

Example

```
signed int        __reverse( signed int);
unsigned long     __reverse(unsigned long);
signed long       __reverse( signed long);
void __far *      __reverse(void __far *);  /* Only on -v4 */
                                            /* and -v6 */
void __huge *     __reverse(void __huge *); /* Only on -v4 */
                                            /* and -v6 */
void __farflash * __reverse(void __farflash *);
          /* Only on -v2 through -v6 with > 64k flash memory */
void __hugeflash * __reverse(void __hugeflash *);
          /* Only on -v2 through -v6 with > 64k flash memory */
```

__save_interrupt

Syntax

```
unsigned char __save_interrupt(void);
```

Description

Saves the state of the interrupt flag in the byte returned. This value can then be used for restoring the state of the interrupt flag with the __restore_interrupt intrinsic function.

Example

```
unsigned char oldState;

oldState = __save_interrupt();
__disable_interrupt();

/* Critical section goes here */

__restore_interrupt(oldState);
```

__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.

__sleep

Syntax   `void __sleep(void);`

Description   Inserts a `SLEEP` instruction.

__swap_nibbles

Syntax   `unsigned char __swap_nibbles(unsigned char);`

Description   Swaps bit 0-3 with bit 4-7 of the parameter and returns the swapped value.

__watchdog_reset

Syntax   `void __watchdog_reset(void);`

Description   Inserts a watchdog reset instruction.

__xch

Syntax   `unsigned char __xch(unsigned char, unsigned char *);`

Description   Provides access to the `XCH` (Exchange) 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 AVR 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 359.

- Preprocessor output

  Use the option --preprocess to direct preprocessor output to a named file, see *--preprocess*, page 268.

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 262.

__BUILD_NUMBER__

Description   A unique integer that identifies the build number of the compiler currently in use. The build number does not necessarily increase with a compiler that is released later.

__CORE__

Description   An integer that identifies the processor variant in use. The value reflects the setting of the processor option -vn,

__COUNTER__

Description   A macro that expands to a new integer each time it is expanded, starting at zero (0) and counting up.

__cplusplus

Description   An integer which is defined when the compiler runs in any of the C++ modes, otherwise it is undefined. When defined, its value is 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.

__CPU__

Description   A symbol that identifies the processor variant in use. The value reflects the setting of the processor option -vn,

__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.

__device__

Description

An integer that identifies the processor variant in use. The value reflects the setting of the `--cpu` option. *device* corresponds exactly to the device name, except for FpSLic, which uses the predefined symbol `__AT94Kxx__`. For example, the symbol is `__AT90S2313__` when the `--cpu=2313` option is used, and `__ATmega163__`, when `--cpu=m163` is used.

__DOUBLE__

Description

An integer that identifies the size of the data type `double`. The symbol is defined to 32 or 64, depending on the setting of the option `--64bit_doubles`.

__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 262.

__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 250 and *__PRETTY_FUNCTION__*, page 356.

__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 250 and *__PRETTY_FUNCTION__*, page 356.

__HAS_EEPROM__

Description   A symbol that determines whether there is internal EEPROM available or not. When this symbol is defined, there is internal EEPROM available. When this symbol is not defined, there is no EEPROM available.

__HAS_EIND__

Description   A symbol that determines whether the instruction EIND is available or not. When this symbol is defined, the instruction EIND is available. When this symbol is not defined, the EIND instruction is not available.

__HAS_ELPM__

Description   A symbol that determines whether the instruction ELPM is available or not. When this symbol is defined, the instruction ELPM is available. When this symbol is not defined, the ELPM instruction is not available.

__HAS_ENHANCED_CORE__

Description   A symbol that determines whether the enhanced core is used or not. The symbol reflects the `--enhanced_core` option and is defined when the enhanced core is used. When this symbol is not defined, the enhanced core is not used.

__HAS_FISCR__

Description   A symbol that determines whether the instruction `FISCR` is available or not. When this symbol is defined, the instruction `FISCR` is available. When this symbol is not defined, the `FISCR` instruction is not available.

__HAS_MUL__

Description   A symbol that determines whether the instruction `MUL` is available or not. When this symbol is defined, the instruction `MUL` is available. When this symbol is not defined, the `MUL` instruction is not available.

__HAS_RAMPD__

Description   A symbol that determines whether the register `RAMPD` is available or not. When this symbol is defined, the register `RAMPD` is available. When this symbol is not defined, the `RAMPD` register is not available.

__HAS_RAMPX__

Description   A symbol that determines whether the register `RAMPX` is available or not. When this symbol is defined, the register `RAMPX` is available. When this symbol is not defined, the `RAMPX` register is not available.

__HAS_RAMPY__

Description   A symbol that determines whether the register `RAMPY` is available or not. When this symbol is defined, the register `RAMPY` is available. When this symbol is not defined, the `RAMPY` register is not available.

__HAS_RAMPZ__

Description A symbol that determines whether the register RAMPZ is available or not. When this symbol is defined, the register RAMPZ is available. When this symbol is not defined, the RAMPZ register is not available.

__IAR_SYSTEMS_ICC__

Description An integer that identifies the IAR compiler platform. The current value is 9. 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.

__ICCAVR__

Description An integer that is set to 1 when the code is compiled with the IAR C/C++ Compiler for AVR.

__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 byte order and is defined to 1 (little-endian).

__MEMORY_MODEL__

Description An integer that identifies the memory model in use. The value reflects the setting of the --memory_model option and is defined to 1 for the Tiny memory model, 2 for the Small memory model, 3 for the Large memory model, and 4 for the Huge memory model.

__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 250 and _*func*_, page 354.

__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.

__TID__

Description
A symbol that expands to the target identifier which contains these parts:

- `t`, a target identifier, which is unique for each IAR compiler. For the AVR microcontroller, the target identifier is `9.0`.
- `c`, the value of the `--cpu` or `-v` option.
- `m`, the value reflects the setting of the `--memory_model` option; the value is defined to `1` for Tiny, `2` for Small, `3` for Large, and `4` for Huge.

The `__TID__` value is constructed as:

```
((t << 8) | (c << 4) | m)
```

You can extract the values as follows:

```
t = (__TID__ >> 8) & 0x7F; /* target id */
c = (__TID__ >> 4) & 0x0F; /* cpu core */
```

```
m = __TID__ & 0x0F;          /* memory model */
```

To find the value of the target identifier for the current compiler, execute:

```
printf("%ld", (__TID__ >> 8) & 0x7F)
```

Note: The use of `__TID__` is not recommended. We recommend that you use the symbols `__ICCAVR__` and `__CORE__` instead.

__TIME__

Description   A string that identifies the time of compilation in the form `"hh:mm:ss"`.

This symbol is required by Standard C.

__TIMESTAMP__

Description   A string constant that identifies the date and time of the last modification of the current source file. The format of the string is the same as that used by the `asctime` standard function (in other words, `"Tue Sep 16 13:03:52 2014"`).

__TINY_AVR__

Description   A symbol that determines whether the reduced tiny AVR core is used or not. When this symbol is defined, the reduced tiny AVR core is used. When this symbol is not defined, the reduced tiny AVR core is not used.

__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.

__VERSION_1_CALLS__

Description   A symbol that expands to 1 if the used calling convention is the old calling convention used in compiler version 1.x. If zero, the new calling convention is used. For more information about calling conventions, see *Calling convention*, page 155.

__XMEGA_CORE__

Description   A symbol that determines whether the xmega core is used or not. When this symbol is
                     defined, the xmega core is used. When this symbol is not defined, the xmega core is not
                     used.

__XMEGA_USB__

Description   A symbol that determines whether the xmega usb core is used or not. When this symbol
                     is defined, the xmega usb core is used. When this symbol is not defined, the xmega usb
                     core is not used.

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   *_ReportAssert*, page 129.

#warning message

Syntax

`#warning message`

where `message` can be any string.

Description

Use this preprocessor directive to produce messages. Typically, this is useful for assertions and other trace utilities, similar to the way the Standard C `#error` directive is used. This directive is not recognized when the `--strict` compiler option is used.

C/C++ standard library functions

- C/C++ standard library overview

- DLIB runtime environment—implementation details

- CLIB runtime environment—implementation details

- AVR-specific library functions

For detailed reference information about the library functions, see the online help system.

C/C++ standard library overview

The compiler comes with two different implementations of the C/C++ standard library:

The IAR DLIB Runtime Environment is a complete implementation of the C/C++ standard library, compliant with Standard C and C++. This library also supports floating-point numbers in IEEE 754 format and it can be configured to include different levels of support for locale, file descriptors, multibyte characters, etc.

The IAR CLIB Runtime Environment is a light-weight implementation of the C standard library, which is not fully compliant with Standard C. Neither does it fully support floating-point numbers in IEEE 754 format and it does not support C++.

For more information about customization, see the chapter *The DLIB runtime environment* and *The CLIB runtime environment*, respectively.

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 choose a runtime library, see *Basic project configuration*, page 48. The linker will include only those routines that are required—directly or indirectly—by your application.

See also *Overriding library modules*, page 107 for information about how you can override library modules with your own versions.

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 `-e` linker option.

REENTRANCY

A function that can be simultaneously invoked in the main application and in any number of interrupts is reentrant. A library function that uses statically allocated data is therefore not reentrant.

Most parts of the DLIB runtime environment are reentrant, but the following functions and parts are not reentrant because they need static data:

● Heap functions—`malloc`, `free`, `realloc`, `calloc`, etc. and the C++ operators `new` and `delete`

● Locale functions—`localeconv`, `setlocale`

● Multibyte functions—`mblen`, `mbrlen`, `mbrtowc`, `mbsrtowc`, `mbtowc`, `wcrtomb`, `wcsrtomb`, `wctomb`

● Rand functions—`rand`, `srand`

● Time functions—`asctime`, `localtime`, `gmtime`, `mktime`

● The miscellaneous functions `atexit`, `perror`, `strerror`, `strtok`

● Functions that use files or the heap in some way. This includes `scanf`, `sscanf`, `getchar`, `getwchar`, `putchar`, and `putwchar`. In addition, if you are using the options `--enable_multibyte` and `--dlib_config=Full`, the `printf` and `sprintf` functions (or any variants) can also use the heap.

For the CLIB library, the `qsort` function and functions that use files in some way are non-reentrant. This includes `printf`, `scanf`, `getchar`, and `putchar`. However, the functions `sprintf` and `sscanf` are reentrant.

Functions that can set `errno` are not reentrant, because an `errno` value resulting from one of these functions can be destroyed by a subsequent use of the function before it is read. This applies to math and string conversion functions, among others.

Remedies for this are:

● Do not use non-reentrant functions in interrupt service routines

● Guard calls to a non-reentrant function by a mutex, or a secure region, etc.

THE LONGJMP FUNCTION

⚠ A `longjmp` is in effect a jump to a previously defined `setjmp`. Any variable length arrays or C++ objects residing on the stack during stack unwinding will not be destroyed. This can lead to resource leaks or incorrect application behavior.

DLIB runtime environment—implementation details

The DLIB runtime environment provides most of the important C and C++ standard library definitions that apply to embedded systems. These are of the following types:

● Adherence to a free-standing implementation of Standard C. The library supports most of the hosted functionality, but you must implement some of its base functionality. For 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 AVR features. See the chapter *Intrinsic functions* for more information.

● Special compiler support for accessing strings in flash memory, see *AVR-specific library functions*, page 370.

In addition, the DLIB runtime environment includes some added C functionality, see *Added C functionality*, page 367.

C HEADER FILES

This section lists the C header files specific to the DLIB runtime environment. 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
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

Table 40: Traditional Standard C header files—DLIB

Header file	Usage
wctype.h	Classifying wide characters

Table 40: 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 41: 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
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 42: Standard template library header files

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

Table 43: New Standard C header files—DLIB

Header file	Usage
climits	Testing integer type properties
clocale	Adapting to different cultural conventions
cmath	Computing common mathematical functions
csetjmp	Executing non-local goto statements
csignal	Controlling various exceptional conditions
cstdarg	Accessing a varying number of arguments
cstdbool	Adds support for the bool data type in C.
cstddef	Defining several useful types and macros
cstdint	Providing integer characteristics
cstdio	Performing input and output
cstdlib	Performing a variety of operations
cstring	Manipulating several kinds of strings
ctime	Converting between various time and date formats
cwchar	Support for wide characters
cwctype	Classifying wide characters

Table 43: 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 DLIB runtime environment includes some added C functionality.

The following include files provide these features:

- fenv.h
- stdio.h
- stdlib.h
- string.h

fenv.h

In fenv.h, trap handling support for floating-point numbers is defined with the functions fegettrapenable and fegettrapdisable. No floating-point status flags are supported.

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, *__time32, __time64*, page 130.

`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.

`__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.

CLIB runtime environment—implementation details

The CLIB runtime environment provides most of the important C standard library definitions that apply to embedded systems. These are of the following types:

● Standard C library definitions available for user programs. These are documented in this chapter.

- The system startup code; see the chapter *The CLIB runtime environment* in this guide.
- Runtime support libraries; for example low-level floating-point routines.
- Intrinsic functions, allowing low-level use of AVR features. See the chapter *Intrinsic functions* for more information.
- Special compiler support for accessing strings in flash memory, see *AVR-specific library functions*, page 370.

LIBRARY DEFINITIONS SUMMARY

This table lists the C header files specific to the CLIB runtime environment:

Header file	Description
assert.h	Assertions
ctype.h*	Character handling
errno.h	Error return values
float.h	Limits and sizes of floating-point types
iccbutl.h	Low-level routines
limits.h	Limits and sizes of integral types
math.h	Mathematics
setjmp.h	Non-local jumps
stdarg.h	Variable arguments
stdbool.h	Adds support for the bool data type in C
stddef.h	Common definitions including size_t, NULL, ptrdiff_t, and offsetof
stdio.h	Input/output
stdlib.h	General utilities
string.h	String handling

Table 44: CLIB runtime environment header files

* The functions is*xxx*, toupper, and tolower declared in the header file ctype.h evaluate their argument more than once. This is not according to the ISO/ANSI standard.

AVR-specific library functions

This section lists the AVR–specific library functions declared in pgmspace.h that allow access to strings in flash memory. The _P functions allow access to strings in flash memory only. The _G functions use the __generic pointer instead, which means that

they allow access to both flash and data memory. Both the _P functions and the _G functions are available in both the IAR CLIB Library and the IAR DLIB Library.

memcmp_G

Syntax   int memcmp_G(const void *s1, const void __generic *s2, size_t n);

Description   Identical to memcmp except that s2 can be located in flash *or* data memory.

memcpy_G

Syntax   void * memcpy_G(void *s1, const void __generic *s2, size_t n);

Description   Identical to memcpy except that it copies string s2 in flash *or* data memory to the location that s1 points to in data memory.

memcpy_P

Syntax   void * memcpy_P(void *s1, PGM_P s2, size_t n);

Description   Identical to memcpy except that it copies string s2 in flash memory to the location that s1 points to in data memory.

printf_P

Syntax   int printf_P(PGM_P __format,…);

Description   Similar to printf except that the format string is in flash memory, not in data memory. For information about how to override default formatter, see *Formatters for printf*, page 113.

puts_G

Syntax   int puts_G(const char __generic *s);

Description   Identical to puts except that the string to be written can be in flash *or* data memory.

puts_P

Syntax   `int puts_P(PGM_P __s);`

Description   Identical to `puts` except that the string to be written is in flash memory, not in data memory.

scanf_P

Syntax   `int scanf_P(PGM_P __format,…);`

Description   Identical to `scanf` except that the format string is in flash memory, not in data memory. For information about how to override the default formatter, see *Formatters for scanf*, page 114.

sprintf_P

Syntax   `int sprintf_P(char *__s, PGM_P __format,…);`

Description   Identical to `sprintf` except that the format string is in flash memory, not in data memory.

sscanf_P

Syntax   `int sscanf_P(const char *__s, PGM_P __format,…);`

Description   Identical to `sscanf` except that the format string is in flash memory, not in data memory.

strcat_G

Syntax   `char *strcat_G(char *s1, const char __generic *s2);`

Description   Identical to `strcat` except that string `s2` can be in flash *or* data memory.

strcmp_G

Syntax   `int strcmp_G(const char *s1, const char __generic *s2);`

Description   Identical to `strcmp` except that string `s2` can be in flash *or* data memory.

strcmp_P

Syntax   `int strcmp_P(const char *s1, PGM_P s2);`

Description   Identical to `strcmp` except that string `s2` is in flash memory, not in data memory.

strcpy_G

Syntax   `char *strcpy_G(char *s1, const char __generic *s2);`

Description   Identical to `strcpy` except that the string `s2` being copied can be in flash *or* data memory.

strcpy_P

Syntax   `char * strcpy_P(char *s1, PGM_P s2);`

Description   Identical to `strcpy` except that the string `s2` being copied is in flash memory, not in data memory.

strerror_P

Syntax   `PGM_P strerror_P(int errnum);`

Description   Identical to `strerror` except that the string returned is in flash memory, not in data memory.

strlen_G

Syntax   `size_t strlen_G(const char __generic *s);`

Description   Identical to `strlen` except that the string being tested can be in flash *or* data memory.

strlen_P

Syntax   `size_t strlen_P(PGM_P s);`

Description   Identical to `strlen` except that the string being tested is in flash memory, not in data memory.

strncat_G

Syntax   `char *strncat_G(char *s1, const char __generic *s2, size_t n);`

Description   Identical to `strncmp` except that the string `s2` can be in flash *or* data memory.

strncmp_G

Syntax   `int strncmp_G(const char *s1, const char __generic *s2, size_t n);`

Description   Identical to `strncmp` except that the string `s2` can be in flash *or* data memory.

strncmp_P

Syntax   `int strncmp_P(const char *s1, PGM_P s2, size_t n);`

Description   Identical to `strncmp` except that the string `s2` is in flash memory, not in data memory.

strncpy_G

Syntax   `char *strncpy_G(char *s1, const char __generic *s2, size_t n);`

Description   Identical to `strncpy` except that the source string `s2` can be in flash *or* data memory.

strncpy_P

Syntax   `char * strncpy_P(char *s1, PGM_P s2, size_t n);`

Description   Identical to `strncpy` except that the source string `s2` is in flash memory, not in data memory.

Segment reference

- Summary of segments

- Descriptions of segments

For more information about placement of segments, see the chapter *Linking your application*.

Summary of segments

The table below lists the segments that are available in the compiler:

Segment	Description
CHECKSUM	Holds the checksum generated by the linker.
CODE	Holds the program code.
CSTACK	Holds the stack used by C or C++ programs.
DIFUNCT	Holds pointers to code, typically C++ constructors, that should be executed by the system startup code before main is called.
EEPROM_I	Holds static and global initialized variables that have the memory attribute __eeprom.
EEPROM_N	Holds __no_init static and global variables that have the memory attribute __eeprom.
FARCODE	Holds program code declared __farfunc.
FAR_C	Holds constant data that has the memory attribute __far.
FAR_F	Holds static and global variables that have the memory attribute __farflash.
FAR_HEAP	Holds the far heap.
FAR_I	Holds the static and global initialized variables that have the memory attribute __far.
FAR_ID	Holds initial values for static and global initialized variables in FAR_I.
FAR_N	Holds __no_init static and global variables that have the memory attribute __far.
FAR_Z	Holds zero-initialized static and global variables that have the memory attribute __far.
HEAP	Holds the heap used for dynamically allocated data using the CLIB library.

Table 45: Segment summary

Segment	Description
HUGE_C	Holds constant data that has the memory attribute `__huge`.
HUGE_F	Holds static and global `__hugeflash` variables.
HUGE_HEAP	Holds the heap used for dynamically allocated data in huge memory using the DLIB library.
HUGE_I	Holds static and global initialized variables that have the memory attribute `__huge`.
HUGE_ID	Holds initial values for static and global initialized variables in `HUGE_I`.
HUGE_N	Holds `__no_init` static and global variables that have the memory attribute `__huge`.
HUGE_Z	Holds zero-initialized static and global variables that have the memory attribute `__huge`.
INITTAB	Contains compiler-generated table entries that describe the segment initialization that will be performed at system startup.
INTVEC	Contains the reset and interrupt vectors.
NEAR_C	Holds constant data that has the memory attribute `__tiny` and `__near`.
NEAR_F	Holds static and global `__flash` variables.
NEAR_HEAP	Holds the heap used for dynamically allocated data in near memory using the DLIB library.
NEAR_I	Holds static and global initialized variables that have the memory attribute `__near`.
NEAR_ID	Holds initial values for static and global initialized variables in `NEAR_I`.
NEAR_N	Holds `__no_init` static and global variables that have the memory attribute `__near`.
NEAR_Z	Holds zero-initialized static and global variables that have the memory attribute `__near`.
RSTACK	Holds the internal return stack.
SWITCH	Holds switch tables for all functions.
TINY_F	Holds static and global `__tinyflash` variables.
TINY_HEAP	Holds the heap used for dynamically allocated data in tiny memory using the DLIB library.
TINY_I	Holds static and global initialized variables that have the memory attribute `__tiny`.
TINY_ID	Holds initial values for static and global variables in `TINY_I`.

Table 45: Segment summary (Continued)

Segment	Description
TINY_N	Holds __no_init static and global variables that have the memory attribute__tiny.
TINY_Z	Holds zero-initialized static and global variables that have the memory attribute __tiny.

Table 45: Segment summary (Continued)

Descriptions of segments

This section gives reference information about each segment.

The segments are placed in memory by the segment placement linker directives -Z and -P, for sequential and packed placement, respectively. Some segments cannot use packed placement, as their contents must be continuous. For information about these directives, see *Using the -Z command for sequential placement*, page 89 and *Using the -P command for packed placement*, page 89, respectively.

For each segment, the segment memory type is specified, which indicates in which type of memory the segment should be placed; see *Segment memory type*, page 72.

For information about how to define segments in the linker configuration file, see *Linking your application*, page 87.

For more information about the extended keywords mentioned here, see the chapter *Extended keywords*.

CHECKSUM

Description   Holds the checksum bytes generated by the linker. This segment also holds the __checksum symbol. Note that the size of this segment is affected by the linker option -J.

Segment memory type   CODE

Memory placement   This segment can be placed anywhere in ROM memory.

Access type   Read-only

CODE

Description	Holds __nearfunc program code, except the code for system initialization.
Segment memory type	CODE
Memory placement	Flash. This segment must be placed within the address range 0x0-0x01FFFE.
Access type	Read-only

CSTACK

Description	Holds the internal data stack.
Segment memory type	DATA
Memory placement	Data. The address range depends on the memory model:
	In the Tiny memory model, this segment must be placed within the address range 0x0-0xFF.
	In the Small memory model, this segment must be placed within the address range 0x0-0xFFFF.
	In the Large and Huge memory models, this segment must be placed within the address range 0x0-0xFFFFFF. In the Large and Huge memory models, the stack can be a maximum of 64 Kbytes, and CSTACK must not cross a 64-Kbyte boundary.
Access type	Read-write
See also	*The stack*, page 60 and *RSTACK*, page 388.

DIFUNCT

Description	Holds the dynamic initialization vector used by C++.
Segment memory type	CODE
Memory placement	This segment must be placed in the first 64 Kbytes of memory.
Access type	Read-only

EEPROM_I

Description	Holds `__eeprom` static and global initialized variables initialized by copying from the segment `EEPROM_ID` at application startup.
	This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the `-Z` directive must be used.
	This segment is not copied to EEPROM during system startup. Instead it is used for programming the EEPROM during the download of the code.
Segment memory type	`XDATA`
Memory placement	This segment must be placed in EEPROM. Use the command line option `--eeprom_size` to set the address range for this segment.
Access type	Read-only
See also	*--eeprom_size*, page 251

EEPROM_N

Description	Holds static and global `__no_init __eeprom` variables.
Segment memory type	`XDATA`
Memory placement	This segment must be placed in EEPROM. Use the command line option `--eeprom_size` to set the address range for this segment.
Access type	Read-only
See also	*--eeprom_size*, page 251

FARCODE

Description	Holds `__farfunc` program code. The `__farfunc` memory attribute is available when using the `-v5` and `-v6` options, in which case the `__farfunc` is implicitly used for all functions.
Segment memory type	`CODE`
Memory placement	This segment must be placed within the address range `0x0–0x7FFFFE` in flash memory.

Access type	Read-only

FAR_C

Description	Holds __far constant data. This can include constant variables, string and aggregate literals, etc.
	Note: This segment is located in external ROM. Systems without external ROM cannot use this segment.
Segment memory type	DATA
Memory placement	External ROM. This segment must be placed within the address range 0x0–0xFFFFFF.
Access type	Read-only

FAR_F

Description	Holds the static and global __farflash variables and aggregate initializers.
Segment memory type	CODE
Memory placement	Flash. This segment must be placed within the address range 0x0–0x7FFFFF.
Access type	Read-only

FAR_HEAP

Description	Holds the heap used for dynamically allocated data in far memory, in other words data allocated by far_malloc and far_free, and in C++, new and delete.
	Note: This segment is only used when you use the DLIB library.
Segment memory type	DATA
Memory placement	Data. This segment must be placed within the address range 0x0–0xFFFFFF.
Access type	Read-write
See also	*Setting up heap memory*, page 92 and *New and Delete operators*, page 181.

FAR_I

Description	Holds `__far` static and global initialized variables initialized by copying from the segment `FAR_ID` at application startup.
	This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the `-Z` directive must be used.
Segment memory type	`DATA`
Memory placement	Data. This segment must be placed within the address range `0x0-0xFFFFFF`.
Access type	Read-write

FAR_ID

Description	Holds initial values for `__far` static and global variables in the `FAR_I` segment. These values are copied from `FAR_ID` to `FAR_I` at application startup.
	This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the `-Z` directive must be used.
Segment memory type	`CODE`
Memory placement	Flash. This segment must be placed within the address range `0x0-0x7FFFFF`.
Access type	Read-only

FAR_N

Description	Holds static and global `__no_init __far` variables.
Segment memory type	`DATA`
Memory placement	Data. This segment must be placed within the address range `0x0-0xFFFFFF`.
Access type	Read-write

FAR_Z

Description	Holds zero-initialized `__far` static and global variables. The contents of this segment is declared by the system startup code.
	This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the `-Z` directive must be used.
Segment memory type	`DATA`
Memory placement	Data. This segment must be placed within the address range `0x0-0xFFFFFF`.
Access type	Read-write

HEAP

Description	Holds the heap used for dynamically allocated data, in other words data allocated by `malloc` and `free`.
	Note: This segment is only used when you use the CLIB library.
Segment memory type	`DATA`
Memory placement	Data. The address range depends on the memory model.
	In the Tiny memory model, this segment must be placed within the address range `0x0-0xFF`.
	In the Small memory model, this segment must be placed within the address range `0x0-0xFFFF`.
	In the Large and Huge memory models, this segment must be placed within the address range `0x0-0xFFFFFF`.
Access type	Read-write
See also	*Setting up heap memory*, page 92 and *New and Delete operators*, page 181.

HUGE_C

Description	Holds `__huge` constant data. This can include constant variables, string and aggregate literals, etc.

Note: This segment is located in external ROM. Systems without external ROM cannot use this segment.

Segment memory type	`DATA`
Memory placement	External ROM. This segment must be placed within the address range `0x0-0xFFFFFF`.
Access type	Read-only

HUGE_F

Description	Holds the static and global `__hugeflash` variables and aggregate initializers.
Segment memory type	`CODE`
Memory placement	Flash. This segment must be placed within the address range `0x0-0xFFFFFF`.
Access type	Read-only

HUGE_HEAP

Description	Holds the heap used for dynamically allocated data in huge memory, in other words data allocated by `huge_malloc` and `huge_free`, and in C++, `new` and `delete`.
	Note: This segment is only used when you use the DLIB library.
Segment memory type	`DATA`
Memory placement	Data. This segment must be placed within the address range `0x0-0xFFFFFF`.
Access type	Read-write
See also	*Setting up heap memory*, page 92 and *New and Delete operators*, page 181.

HUGE_I

Description	Holds `__huge` static and global initialized variables initialized by copying from the segment `HUGE_ID` at application startup.
	This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the `-Z` directive must be used.

When the -y compiler option is used, __huge constant data is located in this segment.

Segment memory type	DATA
Memory placement	Data. This segment must be placed within the address range 0x0–0xFFFFFF.
Access type	Read-write

HUGE_ID

Description	Holds initial values for __huge static and global variables in the HUGE_I segment. These values are copied from HUGE_ID to HUGE_I at application startup.
	This segment cannot be placed in memory by using the -P directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the -Z directive must be used.
Segment memory type	CODE
Memory placement	Flash. This segment must be placed within the address range 0x0–0x7FFFFF.
Access type	Read-only

HUGE_N

Description	Holds static and global __no_init __huge variables.
Segment memory type	DATA
Memory placement	Data. This segment must be placed within the address range 0x0–0xFFFFFF.
Access type	Read-write

HUGE_Z

Description	Holds zero-initialized __huge static and global variables. The contents of this segment is declared by the system startup code.
	This segment cannot be placed in memory by using the -P directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the -Z directive must be used.

Segment memory type	DATA
Memory placement	Data. This segment must be placed within the address range `0x0-0xFFFFFF`.
Access type	Read-write

INITTAB

Description	Holds compiler-generated table entries that describe the segment initialization which will be performed at system startup.
Segment memory type	CODE
Memory placement	Flash. This segment must be placed within the address range `0x0-0xFFFF` (`0x7FFFFF` if farflash is enabled).
Access type	Read-only

INTVEC

Description	Holds the interrupt vector table generated by the use of the `__interrupt` extended keyword in combination with the `#pragma vector` directive.
Segment memory type	CODE
Memory placement	This segment can be placed anywhere in memory.
Access type	Read-only

NEAR_C

Description	Holds `__tiny` and `__near` constant data. This can include constant variables, string and aggregate literals, etc. `TINY_I` is copied from this segment, because `TINY_C` is not a possible segment. **Note:** This segment is located in external ROM. Systems without external ROM cannot use this segment.
Segment memory type	DATA
Memory placement	External ROM. This segment must be placed within the address range `0x0-0xFFFF`.

Access type	Read-only

NEAR_F

Description	Holds the static and global `__flash` variables and aggregate initializers.
Segment memory type	CODE
Memory placement	Flash. This segment must be placed within the address range `0x0-0xFFFF`.
Access type	Read-only

NEAR_HEAP

Description	Holds the heap used for dynamically allocated data in near memory, in other words data allocated by `near_malloc` and `near_free`, and in C++, `new` and `delete`. **Note:** This segment is only used when you use the DLIB library.
Segment memory type	DATA
Memory placement	Data. This segment must be placed within the address range `0x0-0xFFFF`.
Access type	Read-write
See also	*Setting up heap memory*, page 92 and *New and Delete operators*, page 181.

NEAR_I

Description	Holds `__near` static and global initialized variables initialized by copying from the segment NEAR_ID at application startup. This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the `-Z` directive must be used. When the `-y` compiler option is used, NEAR_C data (`__near`) is located in this segment.
Segment memory type	DATA
Memory placement	Data. This segment must be placed within the address range `0x0-0xFFFF`.

Access type	Read-write

NEAR_ID

Description	Holds initial values for `__near` static and global variables in the `NEAR_I` segment. These values are copied from `NEAR_ID` to `NEAR_I` at application startup.
	This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the `-Z` directive must be used.
Segment memory type	`CODE`
Memory placement	Flash. This segment must be placed within the address range `0x0-0x7FFFFF`.
Access type	Read-only

NEAR_N

Description	Holds static and global `__no_init __near` variables.
Segment memory type	`DATA`
Memory placement	Data. This segment must be placed within the address range `0x0-0xFFFF`.
Access type	Read-write

NEAR_Z

Description	Holds zero-initialized `__near` static and global variables. The contents of this segment is declared by the system startup code.
	This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the `-Z` directive must be used.
Segment memory type	`DATA`
Memory placement	Data. This segment must be placed within the address range `0x0-0xFFFF`.
Access type	Read-write

RSTACK

Description	Holds the internal return stack.
Segment memory type	DATA
Memory placement	Data. This segment must be placed within the address range 0x0–0xFFFF.
Access type	Read-write
See also	*CSTACK*, page 378

SWITCH

Description	Holds switch tables for all functions.
	The SWITCH segment is for compiler internal use only and should always be defined. The segment allocates, if necessary, jump tables for C/C++ switch statements.
Segment memory type	CODE
Memory placement	Flash. This segment must be placed within the address range 0x0–0xFFFF. If the __farflash extended keyword and the --enhanced_core option are used, the segment must be placed within the range 0x0–0x7FFFFF. This segment must not cross a 64-Kbyte boundary.
Access type	Read-only

TINY_F

Description	Holds the static and global __tinyflash variables and aggregate initializers.
Segment memory type	CODE
Memory placement	Flash. This segment must be placed within the address range 0x0–0xFF.
Access type	Read-only

TINY_HEAP

Description	Holds the heap used for dynamically allocated data in tiny memory, in other words data allocated by `tiny_malloc` and `tiny_free`, and in C++, `new` and `delete`.
	Note: This segment is only used when you use the DLIB library.
Segment memory type	`DATA`
Memory placement	Data. This segment must be placed within the address range `0x0-0xFF`.
Access type	Read-write
See also	*Setting up heap memory*, page 92 and *New and Delete operators*, page 181.

TINY_I

Description	Holds `__tiny` static and global initialized variables initialized by copying from the segment `TINY_ID` at application startup.
	This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the `-Z` directive must be used.
	When the `-y` compiler option is used, `NEAR_C` data is located in this segment.
Segment memory type	`DATA`
Memory placement	Data. This segment must be placed within the address range `0x0-0xFF`.
Access type	Read-write

TINY_ID

Description	Holds initial values for `__tiny` static and global variables in the `TINY_I` segment. These values are copied from `TINY_ID` to `TINY_I` at application startup.
	This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the `-Z` directive must be used.
Segment memory type	`CODE`
Memory placement	Flash. This segment must be placed within the address range `0x0-0x7FFFFF`.

Access type	Read-only

TINY_N

Description	Holds static and global `__no_init` `__tiny` variables.
Segment memory type	`DATA`
Memory placement	Data. This segment must be placed within the address range `0x0-0xFF`.
Access type	Read-write

TINY_Z

Description	Holds zero-initialized `__tiny` static and global variables. The contents of this segment is declared by the system startup code.
	This segment cannot be placed in memory by using the `-P` directive for packed placement, because the contents must be continuous. Instead, when you define this segment in the linker configuration file, the `-Z` directive must be used.
Segment memory type	`DATA`
Memory placement	Data. This segment must be placed within the address range `0x0-0xFF`.
Access type	Read-write

The stack usage control file

- Overview

- Stack usage control directives

- Syntactic components

Before you read this chapter, see *Stack usage analysis*, page 79.

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

You can also use wildcards in function names. "#*" matches any sequence of characters, and "#?" matches a single character.

Stack usage control directives

This section gives detailed reference information about each stack usage control directive.

call graph root directive

Syntax   call graph root **[** *category* **]** : *func-spec* **[**, *func-spec*... **]**;

Parameters

    category   See *category*, page 395

    func-spec   See *func-spec*, page 395

Description   Specifies that the listed functions are call graph roots. You can optionally specify a call graph root category. Call graph roots are listed under their category in the *Stack Usage* chapter in the linker map file.

The linker will normally issue a warning for functions needed in the application that are not call graph roots and which do not appear to be called.

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

See also   *call_graph_root*, page 322.

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*`, `*stack*`)`	The stack depth of *stack* in the deepest call chain for any call graph root function in the category.
`totalstack(`*category*`, `*stack*`)`	The sum of the stack depths of the deepest call chains for each call graph root function in the category.
`size("`*SEGMENT*`")`	The size of the segment.

Example

```
check that maxstack("Program entry", CSTACK)
              + totalstack("interrupt", RSTACK)
              + 1K
          <= size("CSTACK");
```

In the example, `maxstack` is the deepest call chain of the program on the CSTACK stack plus the total stack usage on the RSTACK stack plus 1024.

See also   *Stack usage analysis*, page 79.

exclude directive

Syntax   exclude *func-spec* **[**, *func-spec*... **]**;

Parameters

func-spec   See *func-spec*, page 395

Description   Excludes the specified functions, and call trees originating with them, from stack usage
                          calculations.

Example   exclude MyFunc5, MyFunc6;

function directive

Syntax   **[** override **]** function **[** *category* **]** *func-spec : stack-size*
                          [, *call-info*... **]**;

Parameters

category   See *category*, page 395

func-spec   See *func-spec*, page 395

call-info   See *call-info*, page 396

stack-size   See *stack-size*, page 397

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 MyOtherFunc: (CSTACK 8, RSTACK 2);
                          function [interrupt] MyInterruptHandler: (CSTACK 44, RSTACK 4);

max recursion depth directive

Syntax   max recursion depth *func-spec* : *size*;

Parameters

func-spec   See *func-spec*, page 395

size   See *size*, page 397

Description   Specifies the maximum number of iterations through any of the cycles in the recursion
                nest of which the function is a member.

                A recursion nest is a set of cycles in the call graph where each cycle shares at least one
                node with another cycle in the nest.

                Stack usage analysis will base its result on the max recursion depth multiplied by the
                stack usage of the deepest cycle in the nest. If the nest is not entered on a point along
                one of the deepest cycles, no stack usage result will be calculated for such calls.

Example   `max recursion depth MyFunc12: 10;`

no calls from directive

Syntax   `no calls from` *module-spec* `to` *func-spec* `[,` *func-spec***...** `];`

Parameters

                func-spec   See *func-spec*, page 395

                module-spec   See *module-spec*, page 396

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.r90] to MyFunc13, MyFun14;`

possible calls directive

Syntax   `possible calls` *calling-func* `:` *called-func* `[ ,` *called-func***...** `];`

Parameters

                calling-func   See *func-spec*, page 395

                called-func   See *func-spec*, page 395

Description   Specifies an exhaustive list of possible destinations for all indirect calls in one function.
                   Use this for functions which are known to perform indirect calls and where you know
                   exactly which functions that might be called in this particular application. Consider
                   using the #pragma calls directive if the information about which functions that might
                   be called is available when compiling.

Example   possible calls MyFunc7: MyFunc8, MyFunc9;

                   When the function does not perform any calls, the list is empty:

                   possible calls MyFunc8: ;

See also   *calls*, page 322.

Syntactic components

This section describes the syntactical components that can be used by the stack usage
control directives.

category

Syntax   [*name*]

Description   A call graph root category. You can use any name you like. Categories are not
                    case-sensitive.

Example   category examples:

                    [interrupt]
                    [task]

func-spec

Syntax   **[** *?* **]** *name* **[** *module-spec* **]**

Description   Specifies the name of a symbol, and for module-local symbols, the name of the module
                    it is defined in. Normally, if *func-spec* does not match a symbol in the program, a
                    warning is emitted. Prefixing with ? suppresses this warning.

Example   *func-spec* examples:

                    xFun
                    MyFun [file.r90]

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.r90]
[file.r90(lib.a)]
["D:\C1\test\file.r90"]
```

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.r90
"file-1.r90"
```

call-info

Syntax
calls *func-spec* **[** , *func-spec*... **][** : *stack-size* **]**

Description
Specifies one or more called functions, and optionally, the stack size at the calls.

Example
    call-info examples:

```
calls MyFunc1 : stack 16
calls MyFunc2, MyFunc3, MyFunc4
```

stack-size

Syntax
    (CSTACK *size*, RSTACK *size*)

Description
    Specifies the size of a stack frame.

Example
    stack-size examples:

```
(CSTACK 16, RSTACK 4)
```

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
```

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 415. For a short overview of the differences between Standard C and C89, see *C language overview*, page 167.

The text in this chapter applies to the DLIB runtime environment. Because the CLIB runtime environment does not follow Standard C, its implementation-defined behavior is not documented. See also *The CLIB runtime environment*, page 141.

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 for the DLIB runtime environment, see *Customizing __low_level_init*, page 122.

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 runtime environment, the set of supported signals is the same as in Standard C. A raised signal will do nothing, unless the `signal` function is customized to fit the application.

Signal values for computational exceptions (7.14.1.1)

In the DLIB runtime environment, there are no implementation-defined values that correspond to a computational exception.

Signals at system startup (7.14.1.1)

In the DLIB runtime environment, there are no implementation-defined signals that are executed at system startup.

Environment names (7.20.4.5)

In the DLIB runtime environment, 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 DLIB runtime environment needs a multibyte character scanner to support a multibyte execution character set. See *Locale*, page 133.

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 282.

The rank of extended integer types (6.3.1.1)

There are no extended integer types.

Signals when converting to a signed integer type (6.3.1.3)

No signal is raised when an integer is converted to a signed integer type.

Signed bitwise operations (6.5)

Bitwise operations on signed integers work the same way as bitwise operations on unsigned integers; in other words, the sign-bit will be treated as any other bit, except for the operator >> which will behave as an arithmetic right shift.

J.3.6 FLOATING POINT

Accuracy of floating-point operations (5.2.4.2.2)

The accuracy of floating-point operations is unknown.

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 289.

ptrdiff_t (6.5.6)

For information about ptrdiff_t, see *ptrdiff_t*, page 289.

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 67.

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 284.

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 250.

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 284.

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 281.

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 291.

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 241.

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 228.

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 228.

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
```

```
building_runtime
```

```
can_instantiate
```

```
codeseg
```

```
cspy_support
```

```
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 99.

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 367.

Feraiseexcept raising floating-point exception (7.6.2.3)

For information about the `feraiseexcept` function raising floating-point exceptions, see *Floating-point environment*, page 286.

Strings passed to the setlocale function (7.11.1.1)

For information about strings passed to the `setlocale` function, see *Locale*, page 133.

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: The default implementation of `signal` does not perform anything. Use the template source code to implement application-specific signal handling. See *signal*, page 129 and *raise*, page 127, respectively.

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 *Briefly about input and output (I/O)*, page 100.

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 *Briefly about input and output (I/O)*, page 100.

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 *Briefly about input and output (I/O)*, page 100.

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 *__time32, __time64*, page 130.

Range and precision of time (7.23)

For information about range and precision, see *time.h*, page 368. The application must supply the actual implementation for the functions time and clock. See *__time32, __time64*, page 130 and *clock*, page 124, respectively.

clock() (7.23.2.1)

The application must supply an implementation of the clock function. See *clock*, page 124.

%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 *__time32, __time64*, page 130.

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 281.

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 281.

The value of the result of the sizeof operator (6.5.3.4)

See *Data representation*, page 281.

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 46: Message returned by strerror()—DLIB runtime environment

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 399. For a short overview of the differences between Standard C and C89, see *C language overview*, page 167.

Descriptions of implementation-defined behavior

The descriptions follow the same order as the ISO appendix. Each item covered includes references to the ISO chapter and section (in parenthesis) that explains the implementation-defined behavior.

TRANSLATION

Diagnostics (5.1.1.3)

Diagnostics are produced in the form:

```
filename,linenumber level[tag]: message
```

where `filename` is the name of the source file in which the error was encountered, `linenumber` is the line number at which the compiler detected the error, `level` is the level of seriousness of the message (remark, warning, error, or fatal error), `tag` is a unique tag that identifies the message, and `message` is an explanatory message, possibly several lines.

ENVIRONMENT

Arguments to main (5.1.2.2.2.1)

The function called at program startup is called `main`. No prototype was declared for `main`, and the only definition supported for `main` is:

```
int main(void)
```

To change this behavior for the DLIB runtime environment, see *Customizing __low_level_init*, page 122. To change this behavior for the CLIB runtime environment, see *Customizing system initialization*, page 147.

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 DLIB runtime environment needs a multibyte character scanner to support a multibyte execution character set. The CLIB runtime environment does not support multibyte characters.

See *Locale*, page 133.

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 DLIB runtime environment will support multibyte characters if you add a locale with multibyte support or a multibyte character scanner to the library. The CLIB runtime environment does not support multibyte characters.

See *Locale*, page 133.

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 282, for information about the ranges for the different integer types.

Demotion of integers (6.2.1.2)

Converting an integer to a shorter signed integer is made by truncation. If the value cannot be represented when converting an unsigned integer to a signed integer of equal length, the bit-pattern remains the same. In other words, a large enough value will be converted into a negative value.

Signed bitwise operations (6.3)

Bitwise operations on signed integers work the same way as bitwise operations on unsigned integers; in other words, the sign-bit will be treated as any other bit, except for the operator >> which will behave as an arithmetic right shift.

Sign of the remainder on integer division (6.3.5)

The sign of the remainder on integer division is the same as the sign of the dividend.

Negative valued signed right shifts (6.3.7)

The result of a right-shift of a negative-valued signed integral type preserves the sign-bit. For example, shifting `0xFF00` down one step yields `0xFF80`.

FLOATING POINT

Representation of floating-point values (6.1.2.5)

The representation and sets of the various floating-point numbers adheres to 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 285, 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 289, for information about size_t.

Conversion from/to pointers (6.3.4)

See *Casting*, page 289, for information about casting of data pointers and function pointers.

ptrdiff_t (6.3.6, 7.1.1)

See *ptrdiff_t*, page 289, 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 282, 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

building_runtime

can_instantiate

codeseg

cspy_support

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.

LIBRARY FUNCTIONS FOR THE IAR DLIB RUNTIME ENVIRONMENT

Note that some items in this list only apply when file descriptors are supported by the library configuration. 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: The default implementation of `signal` does not perform anything. Use the template source code to implement application-specific signal handling. See *signal*, page 129 and *raise*, page 127, respectively.

Terminating newline character (7.9.2)

`stdout` stream functions recognize either `newline` or `end of file (EOF)` as the terminating character for a line.

Blank lines (7.9.2)

Space characters written to the `stdout` stream immediately before a newline character are preserved. There is no way to read the line through the `stdin` stream that was written through the `stdout` stream.

Null characters appended to data written to binary streams (7.9.2)

No null characters are appended to data written to binary streams.

Files (7.9.3)

Whether the file position indicator of an append-mode stream is initially positioned at the beginning or the end of the file, depends on the application-specific implementation of the low-level file routines.

Whether a write operation on a text stream causes the associated file to be truncated beyond that point, depends on the application-specific implementation of the low-level file routines. See *Briefly about input and output (I/O)*, page 100.

The characteristics of the file buffering is that the implementation supports files that are unbuffered, line buffered, or fully buffered.

Whether a zero-length file actually exists depends on the application-specific implementation of the low-level file routines.

Rules for composing valid file names depends on the application-specific implementation of the low-level file routines.

Whether the same file can be simultaneously open multiple times depends on the application-specific implementation of the low-level file routines.

remove() (7.9.4.1)

The effect of a remove operation on an open file depends on the application-specific implementation of the low-level file routines. See *Briefly about input and output (I/O)*, page 100.

rename() (7.9.4.2)

The effect of renaming a file to an already existing filename depends on the application-specific implementation of the low-level file routines. See *Briefly about input and output (I/O)*, page 100.

%p in printf() (7.9.6.1)

The argument to a %p conversion specifier, print pointer, to printf() is treated as having the type void *. The value will be printed as a hexadecimal number, similar to using the %x conversion specifier.

%p in scanf() (7.9.6.2)

The %p conversion specifier, scan pointer, to scanf() reads a hexadecimal number and converts it into a value with the type void *.

Reading ranges in scanf() (7.9.6.2)

A – (dash) character is always treated as a range symbol.

File position errors (7.9.9.1, 7.9.9.4)

On file position errors, the functions fgetpos and ftell store EFPOS in errno.

Message generated by perror() (7.9.10.4)

The generated message is:

usersuppliedprefix:*errormessage*

Allocating zero bytes of memory (7.10.3)

The calloc(), malloc(), and realloc() functions accept zero as an argument. Memory will be allocated, a valid pointer to that memory is returned, and the memory block can be modified later by realloc.

Behavior of abort() (7.10.4.1)

The abort() function does not flush stream buffers, and it does not handle files, because this is an unsupported feature.

Behavior of exit() (7.10.4.3)

The argument passed to the `exit` function will be the return value returned by the `main` function to `cstartup`.

Environment (7.10.4.4)

The set of available environment names and the method for altering the environment list is described in *getenv*, page 125.

system() (7.10.4.5)

How the command processor works depends on how you have implemented the `system` function. See *system*, page 130.

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 47: Message returned by strerror()—DLIB runtime environment

The time zone (7.12.1)

The local time zone and daylight savings time implementation is described in *__time32, __time64*, page 130.

clock() (7.12.2.1)

From where the system clock starts counting depends on how you have implemented the `clock` function. See *clock*, page 124.

LIBRARY FUNCTIONS FOR THE CLIB RUNTIME ENVIRONMENT

NULL macro (7.1.6)

The `NULL` macro is defined to `(void *) 0`.

Diagnostic printed by the assert function (7.2)

The `assert()` function prints:

`Assertion failed:` *expression*`, file` *Filename*`, line` *linenumber*

when the parameter evaluates to zero.

Domain errors (7.5.1)

`HUGE_VAL`, the largest representable value in a double floating-point type, will be returned by the mathematic functions on domain errors.

Underflow of floating-point values sets errno to ERANGE (7.5.1)

The mathematics functions set the integer expression `errno` to `ERANGE` (a macro in `errno.h`) on underflow range errors.

fmod() functionality (7.5.6.4)

If the second argument to `fmod()` is zero, the function returns zero (it does not change the integer expression `errno`).

signal() (7.7.1.1)

The signal part of the library is not supported.

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)

There are no binary streams implemented.

Files (7.9.3)

There are no other streams than `stdin` and `stdout`. This means that a file system is not implemented.

remove() (7.9.4.1)

There are no other streams than `stdin` and `stdout`. This means that a file system is not implemented.

rename() (7.9.4.2)

There are no other streams than `stdin` and `stdout`. This means that a file system is not implemented.

%p in printf() (7.9.6.1)

The argument to a `%p` conversion specifier, print pointer, to `printf()` is treated as having the type `'char *'`. The value will be printed as a hexadecimal number, similar to using the `%x` conversion specifier.

%p in scanf() (7.9.6.2)

The `%p` conversion specifier, scan pointer, to `scanf()` reads a hexadecimal number and converts it into a value with the type `'void *'`.

Reading ranges in scanf() (7.9.6.2)

A – (dash) character is always treated explicitly as a – character.

File position errors (7.9.9.1, 7.9.9.4)

There are no other streams than `stdin` and `stdout`. This means that a file system is not implemented.

Message generated by perror() (7.9.10.4)

`perror()` is not supported.

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 `exit()` function does not return.

Environment (7.10.4.4)

Environments are not supported.

system() (7.10.4.5)

The system() function is not supported.

Message returned by strerror() (7.11.6.2)

The messages returned by strerror() depending on the argument are:

Argument	Message
EZERO	no error
EDOM	domain error
ERANGE	range error
<0 || >99	unknown error
all others	error No.xx

Table 48: Message returned by strerror()—CLIB runtime environment

The time zone (7.12.1)

The time zone function is not supported.

clock() (7.12.2.1)

The clock() function is not supported.

A

G

H

M

N

O

Q

R

U

V

Symbols

Numerics