

# **IAR Runtime Environment and Library**

User Guide

**Addendum to  
IAR C/C++ Compiler Reference Guide**

## **COPYRIGHT NOTICE**

© Copyright 1986–2004 IAR Systems. All rights reserved.

No part of this document may be reproduced without the prior written consent of IAR Systems. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license.

## **DISCLAIMER**

The information in this document is subject to change without notice and does not represent a commitment on any part of IAR Systems. While the information contained herein is assumed to be accurate, IAR Systems assumes no responsibility for any errors or omissions.

In no event shall IAR Systems, its employees, its contractors, or the authors of this document be liable for special, direct, indirect, or consequential damage, losses, costs, charges, claims, demands, claim for lost profits, fees, or expenses of any nature or kind.

## **TRADEMARKS**

IAR, IAR Embedded Workbench, IAR XLINK Linker, IAR XAR Library Builder, IAR XLIB Librarian, IAR MakeApp, and IAR PreQual are trademarks owned by IAR Systems. C-SPY is a trademark registered in Sweden by IAR Systems. IAR visualSTATE is a registered trademark owned by IAR Systems.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

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

## **EDITION NOTICE**

First edition: May 2004

Part number: CAEWDC-1

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

# Contents

Tables .....	vii
Preface .....	ix
<b>Who should read this guide</b> .....	ix
<b>How to use this guide</b> .....	ix
<b>Document conventions</b> .....	ix
Typographic conventions .....	ix
Overview .....	1
<b>IAR language overview</b> .....	1
<b>Getting started using the runtime environment</b> .....	2
Two sets of runtime libraries .....	2
Compiling and linking with the DLIB runtime library .....	3
The DLIB runtime environment .....	5
<b>Introduction to the runtime environment</b> .....	5
Runtime environment functionality .....	5
Library selection .....	6
Situations that require library building .....	7
Library configurations .....	7
Debug support in the runtime library .....	8
<b>Using a prebuilt library</b> .....	9
Customizing a prebuilt library without rebuilding .....	9
<b>Choosing formatting capabilities</b> .....	10
Choosing printf formatter .....	10
Choosing scanf formatter .....	11
<b>Overriding library modules</b> .....	12
<b>Building and using a customized library</b> .....	14
Setting up a library project .....	14
Modifying the library functionality .....	14
Using a customized library .....	15

<b>System startup and termination</b> .....	16
System startup .....	17
System termination .....	17
<b>Customizing system initialization</b> .....	18
__low_level_init .....	18
Modifying the cstartup file .....	18
<b>Standard streams for input and output</b> .....	18
Implementing low-level character input and output .....	19
<b>Configuration symbols for printf and scanf</b> .....	20
Customizing formatting capabilities .....	21
<b>File input and output</b> .....	21
<b>Locale</b> .....	22
Locale support in prebuilt libraries .....	22
Customizing the locale support .....	23
Changing locales at runtime .....	23
<b>Environment interaction</b> .....	24
<b>Signal and raise</b> .....	25
<b>Time</b> .....	25
<b>Strtod</b> .....	26
<b>Assert</b> .....	26
<b>The stack</b> .....	26
<b>The heap</b> .....	27
Heap segments in the DLIB runtime environment .....	27
Heap segments in the CLIB runtime environment .....	28
Heap size allocation in the IAR Embedded Workbench .....	28
Heap size allocation from the command line .....	28
Placement of heap segment .....	28
Heap size and standard I/O .....	28
<b>C-SPY Debugger runtime interface</b> .....	29
Low-level debugger runtime interface .....	29
The debugger terminal I/O window .....	30
<b>Implementation of cstartup</b> .....	30
Modules and segment parts .....	30

<b>Added C functionality</b> .....	32
stdint.h .....	32
stdbool.h .....	32
math.h .....	32
stdio.h .....	33
stdlib.h .....	33
printf, scanf and strtod .....	33
<b>The CLIB runtime environment</b> .....	35
<b>Runtime environment</b> .....	35
<b>Input and output</b> .....	36
Character-based I/O .....	36
Formatters used by printf and sprintf .....	37
Formatters used by scanf and sscanf .....	38
<b>System startup and termination</b> .....	39
System startup .....	39
System termination .....	39
<b>Overriding default library modules</b> .....	39
<b>Customizing system initialization</b> .....	40
<b>Implementation of cstartup</b> .....	40
<b>C-SPY runtime interface</b> .....	40
The debugger terminal I/O window .....	40
Termination .....	40
<b>Using C++</b> .....	41
<b>Overview</b> .....	41
Standard Embedded C++ .....	41
Extended Embedded C++ .....	42
Enabling C++ support .....	42
<b>Feature descriptions</b> .....	43
Using IAR-specific attributes on class members .....	43
Functions .....	46
New and Delete operators .....	46
Templates .....	47
Variants of casts .....	50

Mutable .....	50
Namespace .....	50
The STD namespace .....	50
Pointer to member functions .....	50
Using interrupts and C++ destructors .....	51
<b>Reference information</b> .....	<b>53</b>
<b>Descriptions of options</b> .....	<b>53</b>
<b>Descriptions of pragma directives</b> .....	<b>56</b>
<b>Implementation-defined behavior</b> .....	<b>56</b>
IAR DLIB Library functions .....	57
<b>Library functions</b> .....	<b>59</b>
<b>Introduction</b> .....	<b>59</b>
Header files .....	59
Library object files .....	60
Reentrancy .....	60
<b>IAR DLIB Library</b> .....	<b>60</b>
C header files .....	61
C++ header files .....	62
<b>IAR CLIB Library</b> .....	<b>64</b>
Library definitions summary .....	64
<b>Index</b> .....	<b>67</b>

# Tables

1: Typographic conventions used in this guide .....	ix
2: Command line used when compiling .....	3
3: Command line used when linking .....	4
4: Library configurations .....	7
5: Levels of debugging support in runtime libraries .....	8
6: Customizable items .....	9
7: Formatters for printf .....	11
8: Formatters for scanf .....	12
9: Descriptions of printf configuration symbols .....	20
10: Descriptions of scanf configuration symbols .....	21
11: Low-level I/O files .....	22
12: Functions with special meanings when linked with debug info .....	29
13: Traditional standard C header files—DLIB .....	61
14: Embedded C++ header files .....	62
15: Additional Embedded C++ header files—DLIB .....	62
16: Standard template library header files .....	63
17: New standard C header files—DLIB .....	63
18: IAR CLIB Library header files .....	64
19: Miscellaneous IAR CLIB Library header files .....	65





# Preface

Welcome to the IAR Runtime Environment and Library User Guide. The purpose of this guide is to provide you with detailed information that can help you to use the new features related to the runtime environment, the libraries, and the programming languages provided by the IAR C/C++ Compiler.

---

## Who should read this guide

You should read this guide if you plan to develop an application using the C or C++ language and need to get detailed information on how to use the runtime environment and the runtime library.

---

## How to use this guide

When you start using the IAR C/C++ Compiler, you should read this guide in combination with the *IAR C/C++ Compiler Reference Guide*. Note that the information in this guide replaces the corresponding information in the *IAR C/C++ Compiler Reference Guide*.

---

## Document conventions

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

### TYPOGRAPHIC CONVENTIONS

This guide uses the following typographic conventions:

Style	Used for
<code>computer</code>	Text that you enter or that appears on the screen.
<code>parameter</code>	A label representing the actual value you should enter as part of a command. Note that this style is also used for <i>cpuname</i> , <i>configfile</i> , <i>libraryfile</i> , and other labels representing your product, as well as for the numeric part of filename extensions— <i>xx</i> .
<code>[option]</code>	An optional part of a command.
<code>{a   b   c}</code>	Alternatives in a command.

*Table 1: Typographic conventions used in this guide*




Style	Used for
<b>bold</b>	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
<i>reference</i>	A cross-reference within this guide or to another guide.
	Identifies instructions specific to the IAR Embedded Workbench interface.
	Identifies instructions specific to the command line interface.
	Identifies helpful tips and programming hints.

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

# Overview

This chapter gives you an overview of the supported programming languages, followed by a short introduction about how to get started using the runtime environment.

---

## IAR language overview

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

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

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

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

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

---

## Getting started using the runtime environment

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

### TWO SETS OF RUNTIME LIBRARIES

There are two different sets of runtime libraries provided:

- The IAR DLIB Library, which supports ISO/ANSI C and C++. This library also supports floating-point numbers in IEEE 754 format and it can be configured to include different levels of support for locale, file descriptors, multibytes, et cetera.
- The IAR CLIB Library is a light-weight library, which is not fully compliant with ISO/ANSI C. Neither does it fully support floating-point numbers in IEEE 754 format or does it support Embedded C++. (This library is used by default).

To build code produced by any version of the compiler, you should use the runtime environment components it provides. It is not always possible to link object code produced using an older compiler version with components provided with a newer compiler version.

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

For detailed information about the runtime environments, see the chapters *The DLIB runtime environment* and *The CLIB runtime environment*, respectively.

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

### Migration from CLIB to DLIB

There are some considerations to have in mind if you want to migrate from the CLIB library, the legacy C library, to the modern DLIB C/C++ library:

- The CLIB `exp10()` function defined in `iccext.h` is not available in DLIB.
- The DLIB library uses the low-level I/O routines `__write` and `__read` instead of `putchar` and `getchar`.
- If the heap size in your old compiler version using CLIB was defined in a file named `heap.c`, you must now set the heap size either in the extended linker command file (`*.xcl`) or in the Embedded Workbench to use the DLIB library.

## COMPILING AND LINKING WITH THE DLIB RUNTIME LIBRARY

In earlier versions, the choice of runtime library did not have any impact on the compilation. This has changed in this version. Now you can configure the runtime library to contain the features that are needed by your application.

One example is input and output. An application may use the `fprintf` function for terminal I/O (`stdout`), but the application does not use file I/O functionality on file descriptors associated with the files. In this case the library can be configured so that code related to file I/O is removed but still provides terminal I/O functionality.

This configuration involves the library header files, for example `stdio.h`. This means that when you build your application, the same header file setup must be used as when the library was built. The library setup is specified in a *library configuration file*, which is a header file that defines the library functionality.



### Choosing a runtime library in the IAR Embedded Workbench

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

When building an application using the IAR Embedded Workbench, there are three library configuration alternatives to choose between: **Normal**, **Full**, and **Custom**. **Normal** and **Full** are prebuilt library configurations delivered with the product. **Custom** is used for your own libraries. See *Library configurations*, page 7, for more information.

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



### Choosing a runtime library from the command line

When building an application from the command line, you must use the same library configuration file as when the library was built. For the prebuilt libraries (`rxx`) there is a corresponding library configuration file (`h`), which has the same name as the library. The files are located in the `cpuname\lib` directory.

The command line used when compiling could look like this:

Command line used when compiling	Description
<code>-I\cpuname\inc</code>	Specifies the include paths
<code>-I\cpuname\inc\{clib dlib}</code>	Specifies the library-specific include path. Use <code>clib</code> or <code>dlib</code> depending on which library you are using

Table 2: Command line used when compiling

Command line used when compiling	Description
<code>-D_DLIB_CONFIG_FILE=</code> <code>C:\..\cpuname\lib\</code> <code>configfile.h</code>	Specifies the library configuration file (for the IAR DLIB Library only)

*Table 2: Command line used when compiling (Continued)*

In case you intend to build your own library version, use the default library configuration file `dlcpunameCustom.h`.

The command line used when linking could look like this:

Command line used when linking	Description
<code>-s __program_start</code>	Specifies the label where the application starts
<code>libraryfile.rxx</code>	Specifies the library object file

*Table 3: Command line used when linking*

For information about the prebuilt libraries and how they are configured, see the release notes provided with the IAR product installation.

## Setting library and runtime environment options

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

- The formatters used by the functions `printf`, `scanf`, and their variants, see *Choosing formatting capabilities*, page 10 (DLIB), and *Input and output*, page 36 (CLIB).
- The size of the stack and the heap, see *The stack*, page 26, and *The heap*, page 27, respectively.

# The DLIB runtime environment

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

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

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

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

---

## Introduction to the runtime environment

The runtime environment is the environment in which your application executes. The runtime environment depends on the target hardware, the software environment, and the application code. The IAR DLIB runtime environment can be used as is together with the IAR C-SPY Debugger. However, to be able to run the application on hardware, you must adapt the runtime environment.

This section gives an overview of:

- The runtime environment and its components
- Library selection.

### RUNTIME ENVIRONMENT FUNCTIONALITY

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

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

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

- Support for hardware features:
  - Direct access to low-level processor operations by means of *intrinsic* functions, such as functions for register handling
  - Peripheral unit registers and interrupt definitions in include files
  - Target-specific arithmetic support modules like hardware multipliers or floating-point coprocessors.
- Runtime environment support, that is, startup and exit code and low-level interface to some library functions.
- Special compiler support for some functions, for instance functions for floating-point arithmetics

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

For reference information about the library functions, see the online help system available from the **Help** menu.

## LIBRARY SELECTION

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

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

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

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

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



## SITUATIONS THAT REQUIRE LIBRARY BUILDING

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

You must build your own library when:

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

For information about how to build a customized library, see *Building and using a customized library*, page 14.

## LIBRARY CONFIGURATIONS

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

The following DLIB library configurations are available:

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

Table 4: Library configurations

In addition to these configurations, you can define your own configurations, which means that you must modify the configuration file. Note that the library configuration file describes how a library was built and thus cannot be changed unless you rebuild the library. For further information, see *Building and using a customized library*, page 14.

The prebuilt libraries are based on the default configurations. For a list of all runtime libraries, see the release notes provided with the IAR product installation. There is also a ready-made library project template that you can use if you want to rebuild the runtime library.

## DEBUG SUPPORT IN THE RUNTIME LIBRARY

You can make the library provide different levels of debugging support—basic, runtime, and I/O debugging.

The following table describes the different levels of debugging support:

Debugging support	Linker option in IAR Embedded Workbench	Linker command line option	Description
Basic debugging	Debug information for C-SPY	-Fubrof	Debug support for C-SPY without any runtime support
Runtime debugging	With runtime control modules	-r	The same as -Fubrof, but also includes debugger support for handling program abort, exit, and assertions.
I/O debugging	With I/O emulation modules	-rt	The same as -r, but also includes debugger support for I/O handling, which means that <code>stdin</code> and <code>stdout</code> are redirected to the C-SPY Terminal I/O window, and that it is possible to access files on the host computer during debugging.

Table 5: Levels of debugging support in runtime libraries

If you build your application project with the XLINK options **With runtime control modules** or **With I/O emulation modules**, certain functions in the library will be replaced by functions that communicate with the IAR C-SPY Debugger. For further information, see *C-SPY Debugger runtime interface*, page 29.



To choose linker option for debug support in the IAR Embedded Workbench, choose **Project>Options** and select the **Linker** category. On the **Output** page, select the appropriate **Format** option.

## Using a prebuilt library

The IAR C/C++ Compiler comes with a set of prebuilt libraries configured for different combinations of certain options. For information about available prebuilt libraries and how they are configured, see the release notes delivered with the IAR product installation.

Each library comes with a corresponding library configuration file. The library configuration file has the same base name as the library. You can find the library object files and the library configuration files in the subdirectory `cpuname\lib`.



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



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

- Specify which library object file to use on the XLINK command line, for instance:  
`d1cpuname.rxx`

You can find the library variants as object files in the directory `cpuname\lib`.

- Specify the include paths for the compiler and assembler:  
`-I cpuname\inc`
- Specify the library configuration file for the compiler:  
`-D_DLIB_CONFIG_FILE=C:\...\d1cpuname.h`

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

### CUSTOMIZING A PREBUILT LIBRARY WITHOUT REBUILDING

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

- Setting options for:
  - Formatters used by `printf` and `scanf`
  - The sizes of the heap and stack, see page 26.
- Overriding library modules with your own customized versions.

The following items can be customized without rebuilding:

Items that can be customized	Described on page
Formatters for <code>printf</code> and <code>scanf</code>	page 10
Startup and termination code	page 16

Table 6: Customizable items

Items that can be customized	Described on page
Low-level input and output	page 19
File input and output	page 21
Low-level environment functions	page 24
Low-level signal functions	page 25
Low-level time functions	page 25
Size of heaps, stacks, and segments	page 26

*Table 6: Customizable items (Continued)*

For a description about how to override library modules, see *Overriding library modules*, page 12.

---

## Choosing formatting capabilities

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

**Note:** If you rebuild the library, it is possible to optimize these functions even further, see *Configuration symbols for printf and scanf*, page 20.

### CHOOSING PRINTF FORMATTER

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

The following table summarizes the capabilities of the different formatters:

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

Table 7: Formatters for `printf`

\* Depends on which library configuration is used.

For information about how to fine-tune the formatting capabilities even further, see *Configuration symbols for `printf` and `scanf`*, page 20.



### Specifying the print formatter in the IAR Embedded Workbench

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



### Specifying `printf` formatter from the command line

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

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

## CHOOSING SCANF FORMATTER

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

The following table summarizes the capabilities of the different formatters:

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

Table 8: Formatters for `scanf`

\* Depends on which library configuration that is used.

For information about how to fine-tune the formatting capabilities even further, see *Configuration symbols for `printf` and `scanf`*, page 20.



### Specifying `scanf` formatter in the IAR Embedded Workbench

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



### Specifying `scanf` formatter from the command line

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

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

## Overriding library modules

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

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



### Overriding library modules using the IAR Embedded Workbench

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

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



### Overriding library modules from the command line

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

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

This creates a replacement object module file named `library_module.rxx`.

**Note:** The library configuration file and some other project options must be the same for `library_module` as for the rest of your code. For a list of necessary project options, see the release notes provided with the IAR product installation.

- 4 Add `library_module.rxx` to the XLINK command line.

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

Run XLINK to rebuild your application.

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

## Building and using a customized library

In some situations, see *Situations that require library building*, page 7, it is necessary to rebuild the library. In those cases you need to:

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

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

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

### SETTING UP A LIBRARY PROJECT

The IAR Embedded Workbench provides a library project template which can be used for customizing the runtime environment configuration. This library template has full library configuration, see Table 4, *Library configurations*, page 7.



In the IAR Embedded Workbench, modify the generic options in the created library project to suit your application.

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

### MODIFYING THE LIBRARY FUNCTIONALITY

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

The library functionality is determined by a set of *configuration symbols*. The default values of these symbols are defined in the `Dlib_defaults.h` file. This read-only file describes the configuration possibilities. In addition, your library has its own library configuration file `dlcpunameCustom.h`, which sets up that specific library with full library configuration. For more information, see Table 6, *Customizable items*, page 9.

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



## Modifying the library configuration file

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

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

## USING A CUSTOMIZED LIBRARY

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



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

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

## System startup and termination

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

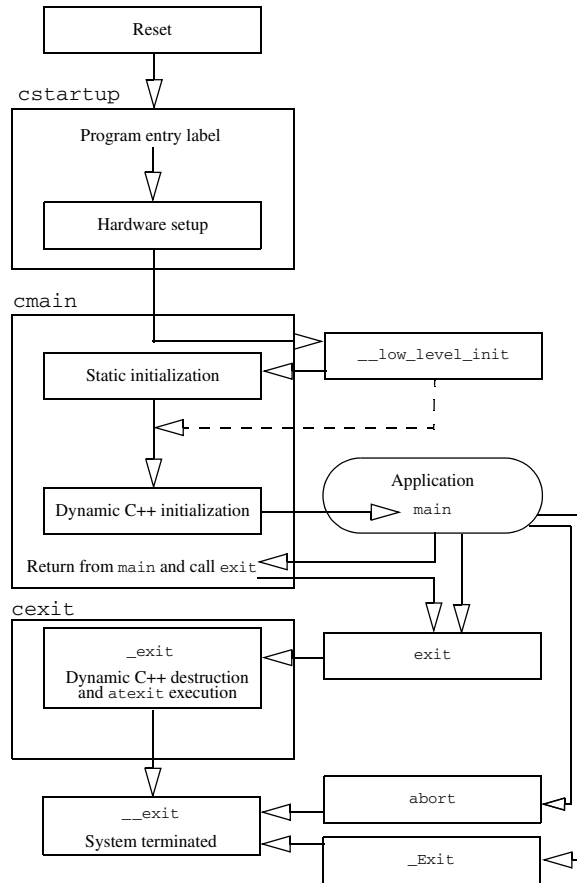


Figure 1: Startup and exit sequences

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

**Note:** Depending on your product installation, the functionality provided by `cmain.sxx` might be included in `cstartup.sxx` instead of being in a separate file.

## SYSTEM STARTUP

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

- When the cpu is reset it will jump to the program entry label `__program_start` in the `cstartup` module.
- The function `__low_level_init` is called, giving the application a chance to perform early initializations
- Static variables are initialized; this includes clearing zero-initialized memory and copying the ROM image of the RAM memory of the rest of the initialized variables depending on the return value of `__low_level_init`
- Static C++ objects are constructed
- The `main` function is called, which starts the application.

## SYSTEM TERMINATION

An application can terminate normally in two different ways:

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

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

The default `exit` function is written in C. It calls a small function `_exit` provided by the `cstartup` file.

The `_exit` function will perform the following operations:

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

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

## C-SPY interface to system termination

If your project is linked with support for runtime debugging, the normal `__exit` and `abort` functions are replaced with special ones. C-SPY will then recognize when those functions are called and can take appropriate actions to simulate program termination. For more information, see *C-SPY Debugger runtime interface*, page 29.

---

## Customizing system initialization

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

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

The code for handling system startup is located in the source files `cstartup.sxx` and `low_level_init.c`, located in the `cpuname\src` directory. If you intend to rebuild the library, the source files are available in the template library project, see *Building and using a customized library*, page 14.

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

### **\_\_LOW\_LEVEL\_INIT**

There is a skeleton low-level initialization file supplied with the product—the C source file `low_level_init.c`. The only limitation using a C source version is that static initialized variables cannot be used within the file, as variable initialization has not been performed at this point.

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

### **MODIFYING THE CSTARTUP FILE**

As noted earlier, you should not modify the `cstartup.sxx` file if a customized version of `__low_level_init` is enough for your needs. However, if you do need to modify the `cstartup.sxx` file, we recommend that you follow the general procedure for creating a modified copy of the file and adding it to your project, see *Overriding library modules*, page 12.

---

## Standard streams for input and output

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

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

## IMPLEMENTING LOW-LEVEL CHARACTER INPUT AND OUTPUT

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

If you intend to rebuild the library, the source files are available in the template library project, see *Building and using a customized library*, page 14. Note that customizing the low-level routines for input and output does not require you to rebuild the library.

**Note:** If you write your own variants of `__read` or `__write`, special considerations for the C-SPY runtime interface are needed, see *C-SPY Debugger runtime interface*, page 29.

### Example of using `__write` and `__read`

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

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

```

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

```

For information about the @ operator, see the *IAR C/C++ Compiler Reference Guide*.

---

## Configuration symbols for printf and scanf

When you set up your application project, you typically need to consider what `printf` and `scanf` formatting capabilities your application requires, see *Choosing formatting capabilities*, page 10.

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

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

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

<b>Printf configuration symbols</b>	<b>Includes support for</b>
<code>_DLIB_PRINTF_MULTIBYTE</code>	Multibyte characters
<code>_DLIB_PRINTF_LONG_LONG</code>	Long long (ll qualifier)
<code>_DLIB_PRINTF_SPECIFIER_FLOAT</code>	Floating-point numbers
<code>_DLIB_PRINTF_SPECIFIER_A</code>	Hexadecimal floats
<code>_DLIB_PRINTF_SPECIFIER_N</code>	Output count (%n)
<code>_DLIB_PRINTF_QUALIFIERS</code>	Qualifiers h, l, L, v, t, and z
<code>_DLIB_PRINTF_FLAGS</code>	Flags -, +, #, and 0
<code>_DLIB_PRINTF_WIDTH_AND_PRECISION</code>	Width and precision

Table 9: Descriptions of printf configuration symbols

Printf configuration symbols	Includes support for
<code>_DLIB_PRINTF_CHAR_BY_CHAR</code>	Output char by char or buffered

Table 9: Descriptions of printf configuration symbols (Continued)

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

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

Table 10: Descriptions of scanf configuration symbols

## CUSTOMIZING FORMATTING CAPABILITIES

To customize the formatting capabilities, you need to set up a library project, see *Building and using a customized library*, page 14. Define the configuration symbols according to your application requirements.

## File input and output

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

Note that file I/O capability in the library is only supported by libraries with full library configuration, see *Library configurations*, page 7. In other words, file I/O is supported when the configuration symbol `__DLIB_FILE_DESCRIPTOR` is enabled. If not enabled, functions taking a `FILE *` argument cannot be used.

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

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

Table 11: Low-level I/O files

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

**Note:** If you link your library with I/O debugging support, C-SPY variants of the low-level I/O functions will be linked for interaction with C-SPY. For more information, see *Debug support in the runtime library*, page 8.

## Locale

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

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

The DLIB library can be used in two major modes:

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

### LOCALE SUPPORT IN PREBUILT LIBRARIES

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

- All prebuilt libraries supports the C locale only
- All libraries with *full library configuration* have support for the locale interface. For prebuilt libraries with locale interface, it is by default only supported to switch multibyte encoding during runtime.



- Libraries with *normal library configuration* do not have support for the locale interface.

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

## CUSTOMIZING THE LOCALE SUPPORT

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

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

### Locale configuration symbols

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

If you want to customize the locale support, you simply define the locale configuration symbols required by your application. For more information, see *Building and using a customized library*, page 14.

**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 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 encoding:

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

---

## Environment interaction

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

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

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

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

```
key=value\0
```

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

For example:

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

If you need a more sophisticated environment variable handling, you should implement your own `getenv`, and possibly `putenv` function. This does not require that you rebuild the library. You can find source templates in the files `getenv.c` and `environ.c` in the `cpuname\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 12.

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

If you decide to rebuild the library, you can find source templates in the library project template. For further information, see *Building and using a customized library*, page 14.

**Note:** If you link your application with support for I/O debugging, the functions `getenv` and `system` will be replaced by C-SPY variants. For further information, see *Debug support in the runtime library*, page 8.

---

## Signal and raise

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

This does not require that you rebuild the library. You can find source templates in the files `Signal.c` and `Raise.c` in the `cpuname\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 12.

If you decide to rebuild the library, you can find source templates in the library project template. For further information, see *Building and using a customized library*, page 14.

---

## Time

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

This does not require that you rebuild the library. You can find source templates in the files `Clock.c` and `Time.c`, and `Getzone.c` in the `cpuname\src\lib` directory. For information about overriding default library modules, see *Overriding library modules*, page 12.

If you decide to rebuild the library, you can find source templates in the library project template. For further information, see *Building and using a customized library*, page 14.

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

**Note:** If you link your application with support for I/O debugging, the functions `clock` and `time` will be replaced by C-SPY variants that return the host clock and time respectively. For further information, see *C-SPY Debugger runtime interface*, page 29.

---

## Strtod

The function `strtod` does not accept hexadecimal floating-point strings in libraries with the normal library configuration. To make a library do so, you need to rebuild the library, see *Building and using a customized library*, page 14. Enable the configuration symbol `_DLIB_STRTOD_HEX_FLOAT` in the library configuration file.

---

## Assert

If you have linked your application with support for runtime debugging, C-SPY will be notified about failed asserts. If this is not the behavior you require, you must add the source file `xReportAssert.c` to your application project. Alternatively, you can rebuild the library. The `__ReportAssert` function generates the assert notification. You can find template code in the `cpuname\src` directory. For further information, see *Building and using a customized library*, page 14.

---

## The stack

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

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

Allocating a memory area for the stack is done differently when you use the command line interface compared to when you use the IAR Embedded Workbench IDE.



### Stack size allocation in the IAR Embedded Workbench

Select **Project>Options**. In the **General Options** category, click the **Stack/Heap** page.

Add the required stack size in the **Stack size** text box.



### Stack size allocation from the command line

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

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

```
\\-D_CSTACK_SIZE=size
```

Remove the comment characters and specify the appropriate size. Note that the size is written hexadecimally without the `0x` notation.



### Placement of stack segment

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

```
-Z (DATA) CSTACK+_CSTACK_SIZE=A000-FE1F
```

**Note:** This range is not the size of the stack, but the range of the available memory.

### Stack size considerations

The compiler uses the internal data stack, `CSTACK`, for a variety of user program operations, and the required stack size depends heavily on the details of these operations. If the given stack size is too small, the stack will normally overwrite the variable storage, which is likely to result in program failure. If the given stack size is too large, RAM will be wasted.

---

## The heap

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

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

- Linker segments used for the heap, which differs between the DLIB and the CLIB runtime environment
- Allocating the heap size, which differs depending on which build interface you are using
- Placing the heap segments in memory.

### HEAP SEGMENTS IN THE DLIB RUNTIME ENVIRONMENT

The compiler supports heaps in different memory types. To access a specific heap, use the appropriate memory attribute as a prefix to the standard functions `malloc`, `free`, `calloc`, and `realloc`. If you use any of the default functions it will be mapped to one of the memory-specific variants depending on your project settings, such as data model.

Each heap will reside in a segment with the name `_HEAP` prefixed by a memory attribute.

For more information about this, see the release notes provided with the IAR product installation.

## HEAP SEGMENTS IN THE CLIB RUNTIME ENVIRONMENT

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 ALLOCATION IN THE IAR EMBEDDED WORKBENCH

Select **Project>Options**. Options for setting the heap size are available in the **General Options** category.

Add the required heap size in the **Heap size** text box.



## HEAP SIZE ALLOCATION FROM THE COMMAND LINE

The size of the heap segment is defined in the linker command file.

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

```
-D_HEAP_SIZE=size
```

Remove the comment characters and specify the appropriate size.



## PLACEMENT OF HEAP SEGMENT

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

```
-Z (DATA) HEAP+_HEAP_SIZE=08000-08FFF
```

**Note:** This range is not the size of the heap, but the range of the available memory.



## HEAP SIZE AND STANDARD I/O

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

## C-SPY Debugger runtime interface

To include support for runtime and I/O debugging, you must link your application with the XLINK options **With runtime control modules** or **With I/O emulation modules**, see *Debug support in the runtime library*, page 8. In this case, C-SPY variants of the following library functions will be linked to the application:

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

Table 12: Functions with special meanings when linked with debug info

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

### LOW-LEVEL DEBUGGER RUNTIME INTERFACE

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

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

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

## THE DEBUGGER TERMINAL I/O WINDOW

To make the Terminal I/O window available, the application must be linked with support for I/O debugging, see *Debug support in the runtime library*, page 8. This means that when the functions `__read` or `__write` are called to perform I/O operations on the streams `stdin`, `stdout`, or `stderr`, data will be sent to or read from the C-SPY Terminal I/O window.

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

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

---

## Implementation of cstartup

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

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

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

## MODULES AND SEGMENT PARTS

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

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



When XLINK builds an application, it starts with a small number of modules that have either been declared using the `__root` keyword or have the program entry label. It then continues to include all modules that are referred from the already included modules. XLINK then discards unused segment parts.

### Segment parts, REQUIRE, and the falling-through trick

The `cstartup.sxx` file has been designed to use the mechanism described in the previous paragraph, so that as little as possible of unused code will be included in the linked application.

For example, every piece of code used for initializing one type of memory is stored in a segment part of its own. If a variable is stored in a certain memory type, the corresponding initialization code will be referenced by the code generated by the compiler, and included in your application. Should no variables of a certain type exist, the code is simply discarded.

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

The segment parts defined in the `cstartup.sxx` file are guaranteed to be placed immediately after each other. XLINK will not change the order of the segment parts or modules, because the segments are placed using the `-Z` option.

This lets the `cstartup.sxx` file specify code in subsequent segment parts and modules that are designed so that some of the parts may not be included by XLINK. The code simply falls through to the next piece of code not discarded by the linker. The following example shows this technique:

```

MODULE doSomething

RSEG MYSEG:CODE:NOROOT(1) // First segment part.
PUBLIC ?do_something
EXTERN ?end_of_test
REQUIRE ?end_of_test

?do_something: // This will be included if someone refers to
... // ?do_something. If this is included then
// the REQUIRE directive above ensures that
// the RETURN instruction below is included.

RSEG MYSEG:CODE:NOROOT(1) // Second segment part.
PUBLIC ?do_something_else

?do_something_else:
... // This will only be included in the linked
// application if someone outside this function

```

```

// refers to or requires ?do_something_else

RSEG MYSEG:CODE:NOROOT(1) // Third segment part.
PUBLIC ?end_of_test

?end_of_test:
    RETURN // This instruction differs depending on which
           // compiler you are using. The instruction is
           // included if ?do_something above
           // is included.

ENDMOD

```

---

## Added C functionality

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

The following include files provide this added functionality:

- `stdint.h`
- `stdbool.h`
- `math.h`
- `stdio.h`
- `stdlib.h`

### STDINT.H

This include file provides integer characteristics.

### STDBOOL.H

This include file makes the `bool` type available if the **Allow IAR extensions** (`-e`) option is used.

### MATH.H

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

## STDIO.H

In `stdio.h`, the following functions have been added from the C99 standard:

<code>vscanf</code>	Variants that have a <code>va_list</code> as argument.
<code>vfscanf</code>	
<code>vsscanf</code>	
<code>vsnprintf</code>	
<code>snprintf</code>	Same as <code>sprintf</code> , but writes to a size-limited array.

The following functions have been added to provide I/O functionality for libraries built without `FILE` support:

<code>__write_array</code>	Corresponds to <code>fwrite</code> on <code>stdout</code> .
<code>__ungetchar</code>	Corresponds to <code>ungetc</code> on <code>stdout</code> .
<code>__gets</code>	Corresponds to <code>fgets</code> on <code>stdin</code> .

## STDLIB.H

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

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

## PRINTF, SCANF AND STRTOD

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



# The CLIB runtime environment

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

The standard library uses a small set of low-level input and output routines for character-based I/O. This chapter describes how the low-level routines can be replaced by your own version.

The chapter also describes how you can choose printf and scanf formatters.

Finally, the chapter describes system initialization and termination. It presents how an application can control what happens before the start function main is called, and the method for how you can customize the initialization.

Note that 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 *Migration from CLIB to DLIB*, page 2.

---

## Runtime environment

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 reference information about the runtime libraries, see the online help system available from the **Help** menu.

The IAR C/C++ Compiler comes with a set of prebuilt libraries configured for different combinations of certain options. For information about available prebuilt libraries and how they are configured, see the release notes delivered with the IAR product installation. You can find the library object files and the library configuration files in the subdirectory `cpuname\lib`.



The IAR Embedded Workbench includes the correct runtime library based on the options you select. See the *IAR Embedded Workbench™ IDE User Guide* for additional information.



Specify which runtime library object file to use on the XLINK command line.

---

## 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 functions through which C performs all 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 the following 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 DEV_IO @ address;

int putchar(int outchar)
{
    DEV_IO = 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 12.

## FORMATTERS USED BY PRINTF AND SPRINTF

The `printf` and `sprintf` functions use a common formatter, called `_formatted_write`. The full version of `_formatted_write` is very large, and provides facilities not required in many embedded applications. To reduce the memory consumption, two smaller, alternative versions are also provided in the standard C library.

### `_medium_write`

The `_medium_write` formatter has the same functions as `_formatted_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 `_formatted_write`.

### `_small_write`

The `_small_write` formatter works in the same way as `_medium_write`, except that it supports only the `%i`, `%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 `_formatted_write`.



### Specifying the printf formatter in the IAR Embedded Workbench

- 1 Choose **Project>Options** and select the **General Options** category. Click the **Library options** tab.
- 2 Select the appropriate **Printf formatter** option, which can be either **Small**, **Medium**, or **Large**.



### Specifying the printf formatter from the command line

To use the `_small_write` or `_medium_write` formatter, add the corresponding line in the linker command file:

```
-e_small_write=_formatted_write
```

or

```
-e_medium_write=_formatted_write
```

To use the full version, remove the line.

## 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 may 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 12.

## FORMATTERS USED BY SCANF AND SSCANF

Similar to the `printf` and `sprintf` functions, `scanf` and `sscanf` use a common formatter, called `_formatted_read`. The full version of `_formatted_read` is very large, and provides facilities that are not required in many embedded applications. To reduce the memory consumption, an alternative smaller version—`_medium_read`—is also provided.

### `_medium_read`

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



### Specifying the scanf formatter in the IAR Embedded Workbench

- 1 Choose **Project>Options** and select the **General Options** category. Click the **Library options** tab.
- 2 Select the appropriate **Scanf formatter** option, which can be either **Medium** or **Large**.



### Specifying the read formatter from the command line

To use the `_medium_read` formatter, add the following line in the linker command file:

```
-e_medium_read=_formatted_read
```

To use the full version, remove the line.



---

## 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.sxx` and `low_level_init.c`, located in the `cpuname\src` directory.

### SYSTEM STARTUP

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

- The custom function `__low_level_init` is called, giving the application a chance to perform early initializations
- Static variables are initialized; this includes clearing zero-initialized memory and copying the ROM image of the RAM memory of the remaining initialized variables
- The `main` function is called, which starts the application.

Note that `cstartup` 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 ISO/ANSI C standard states that the two methods should be equivalent, the `cstartup` code calls the `exit` function if `main` returns. The parameter passed to the `exit` function is the return value of `main`. The default `exit` function is written in 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` in order 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 12 in the chapter *The DLIB runtime environment*.

---

## Customizing system initialization

For information about how to customize system initialization, see *Customizing system initialization*, page 18.

---

## Implementation of `cstartup`

For information about `cstartup` implementation, see *Implementation of `cstartup`*, page 30, in the chapter *The DLIB runtime environment*.

---

## C-SPY runtime interface

The low-level debugger 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 has typed some input and pressed 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 *IAR Embedded Workbench™ IDE User Guide*.

### TERMINATION

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

# Using C++

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

---

## Overview

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

### STANDARD EMBEDDED C++

The following C++ features are supported:

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

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

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

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

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

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

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

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

## EXTENDED EMBEDDED C++

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

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

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

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

## ENABLING C++ SUPPORT



In the IAR C/C++ Compiler, the default language is C. To be able to compile files written in Embedded C++, you must use the `--ec++` compiler option. See the *IAR C/C++ Compiler Reference Guide*. You must also use the IAR DLIB runtime library.

To take advantage of *Extended* Embedded C++ features in your source code, you must use the `--eec++` compiler option. See `--eec++`, page 53.



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

## Feature descriptions

When writing C++ source code for the IAR C/C++ Compiler, you need to know the effects of mixing C++ features—such as classes, and class members—with IAR language extensions, such as IAR-specific attributes.

In this guide, attributes that affect placement of data are referred to as `__data_mem1`, `__data_mem2`, and `__data_mem3`. Attributes that affect placement of functions are referred to as `__func_mem1` and `__func_mem2`. For information about which IAR-specific attributes that are supported by the IAR C/C++ Compiler you are using, see the *IAR C/C++ Compiler Reference Guide*.

### USING IAR-SPECIFIC ATTRIBUTES ON CLASS MEMBERS

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

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

#### Example

```
class A {
public:
    static __data_mem1 int i @ 60; //Located in data_mem1
                                   at address 60
    static __func_mem1 void f(); //Located in func_mem1 memory
    __func_mem1 void g();       //Located in func_mem1 memory
    virtual __func_mem1 void h(); //Located in func_mem1 memory
};
```

#### Class memory

The `this` pointer used for referring to 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 casted to a default data pointer.

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 __data_mem2 C {
public:
    void f();           // Has a this pointer of type
                       // C __data_mem2 *
    void f() const;   // Has a this pointer of type
                       // C __data_mem2 const *
    C();              // Has a this pointer pointing into data_mem2
                       // memory
    C(C const &);     // Takes a parameter of type
                       // C __data_mem2 const &
                       // (also true of generated copy constructor)

    int i;
};
C Ca;                 // Resides in data_mem2 memory instead of the
                       // default memory
C __data_mem1 Cb;    // Resides in data_mem1 memory, the 'this'
                       // pointer still points into data_mem2 memory
C __data_mem3 Cc;    // Not allowed, __data_mem3 pointer can't be
                       // implicitly casted into a __data_mem2
pointer
void h()
{
    C Cd;             // Resides on the stack
}
C * Cp;              // Creates a pointer to data_mem2 memory
C __data_mem1 * Cp;  // Creates a pointer to data_mem1
memory
```

**Note:** 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 __data_mem2 C;
```

Also note that class types associated with different class memories are not compatible types.

There is a built-in operator that returns the class memory type associated with a class, `__memory_of(class)`. For instance, `__memory_of(C)` returns `__data_mem2`.

Inheritance of a base class with class memory restricts the memory type associated with the subclass to be associated with a class memory that resides in the class memory associated with the base class. (A pointer to it can implicitly be casted to a pointer to the class memory associated with the base class.)

```
class __data_mem2 D : public C { // OK, same class memory
public:
    void g();
    int j;
};

class __data_mem1 E : public C { // OK, data_mem1 memory is
                                // inside data_mem2
public:
    void g() // Has a this pointer pointing into data_mem1
            // memory
    {
        f(); // Gets a this pointer into data_mem2 memory
    }
    int j;
};

class __data_mem3 F : public C { // Not OK, data_mem3 memory
                                // isn't inside data_mem3 memory
public:
    void g();
    int j;
};

class G : public C { // OK, will be associated with same class
                    // memory as C
public:
    void g();
    int j;
};
```

A new expression on the class will allocate memory in the heap residing in 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.

If a pointer to class memory cannot be implicitly casted into a default pointer type, no temporaries can be created for that class.

For more information about memory types, see the *IAR C/C++ Compiler Reference Guide*.

## FUNCTIONS

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

### Example

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

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

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

## NEW AND DELETE OPERATORS

A `new` expression can allocate a data type in any memory that has a heap with the following syntax:

```
new attribute datatype;
```

### Example

```
int __data_mem2 *p = new __data_mem2 int;
int __data_mem2 *q = new __data_mem1 int;
int __data_mem2 *r = new __data_mem2 int[10];
```

The `delete` expression must mention the memory in which the object was allocated, like:

```
delete __data_mem2 p;
delete __data_mem1 q;
delete[] __data_mem2 r;
```

Note that the `delete` of an allocated object must use the same memory as the `new` used.

```
delete __data_mem2 q; // Corrupts the heaps
```



To override a global or class-local `new` or `delete` operator for a specific memory type, use the following syntax:

```
void mem *operator new mem(mem-itype);
void operator delete(void mem *);
void mem *operator new[] mem(mem-itype);
void operator delete[] (void mem *);
```

where `mem` is a data memory attribute and `mem-itype` is the index type of the pointer that points into the memory type corresponding to `mem`.

Note that not all memory types have a heap implemented.

## TEMPLATES

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

### Templates and data memory attributes

For data memory attributes to work as expected in templates, two elements of the standard C++ template handling have been 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 data_mem2 is the memory type of the default
// pointer.
template<typename> class Z;
template<typename T> class Z<T *>;

Z<int __data_mem1 *> zn; // T = int __data_mem1
Z<int __data_mem2 *> zf; // T = int
Z<int *> zd; // T = int
Z<int __data_mem3 *> zh; // T = int __data_mem3
```

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

```
template<typename T> void fun(T *);

fun((int __data_mem1 *) 0); // T = int
fun((int      *) 0);      // T = int
fun((int __data_mem2 *) 0); // T = int
fun((int __data_mem3 *) 0); // T = int __data_mem3
```

Note that line 3 above gets a different result than the analogous situation with class template specializations.

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. However, for *large* and “other” memory types (memory that cannot be pointed to by a default pointer) it is possible. In order 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

```
#pragma basic_template_matching
template<typename T> void fun(T *);

fun((int __data_mem1 *) 0); // T = int __data_mem1
```

## 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 __sfr x; // Not possible to point at x

template<__sfr int &y>
void foo()
```

```

{
    y = 17;
}

void bar()
{
    foo<x>();
}

```

**Note:****The standard template library**

The STL (standard template library) delivered with the product is tailored for Extended EC++, as described in *Extended Embedded C++*, page 42.

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**

```

vector<int > d; // d placed in default memory, using
               // the default heap, uses default
               // pointers
vector<int __data_mem1> __data_mem1 x; // x placed in data_mem1
                                       // memory, heap allocation
                                       // from data_mem1, uses
                                       // pointers to data_mem1
                                       // memory
vector<int __data_mem3> __data_mem1 y; // y placed in data_mem1
                                       // memory, heap allocation
                                       // from data_mem3, uses
                                       // pointers to data_mem3
                                       // memory
vector<int __data_mem1> __data_mem3 z; // Illegal

```

Note 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.

Note that two containers that only differ by the data memory attribute they use cannot be assigned to each other.

**Example**

```
vector<int __data_mem1> x;
vector<int __data_mem3> y;

x = y; // Illegal
y = x; // Illegal
```

However, the templated assign member method will work:

```
x.assign(y.begin(), y.end());
y.assign(x.begin(), x.end());
```

**STL and the IAR C-SPY Debugger**

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

**VARIANTS OF CASTS**

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

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

**MUTABLE**

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

**NAMESPACE**

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

**THE STD NAMESPACE**

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

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

**POINTER TO MEMBER FUNCTIONS**

A pointer to a member function can only contain a default function pointer, or a function pointer that can implicitly be casted to a default function pointer. To use a pointer to a member function, make sure that all functions that should be pointed to reside in the default memory or a memory contained in the default memory. To read more about available pointer types, see the *IAR C/C++ Compiler Reference Guide*.

**Example**

```
class X{
    __func_mem1 void f();
};
void (__func_mem1 X::*pmf)(void) = &X::f;
```

**USING INTERRUPTS AND C++ DESTRUCTORS**

If interrupts are enabled and the interrupt functions use class objects that have destructors, there may be problems if the program exits either by using `exit` or by returning from `main`. If an interrupt occurs after an object has been destroyed, there is no guarantee that the program will work properly.

To avoid this, you must override the function `exit(int)`.

The standard implementation of this function (located in the file `exit.c`) looks like this:

```
extern void _exit(int arg);
void exit(int arg)
{
    _exit(arg);
}
```

`_exit(int)` is responsible for calling the destructors of global class objects before ending the program.

To avoid interrupts, place a call to the intrinsic function `__disable_interrupts()` before the call to `_exit()`.



# Reference information

This chapter gives detailed reference information about new features related to the runtime environment and the library. Note that other features might have been added after the writing of this guide, see the release notes provided with your IAR product installation.

---

## Descriptions of options

This section gives detailed reference information about new compiler options.

---

`--eec++`    `--eec++`

In the IAR C/C++ Compiler, the default language is C. If you take advantage of Extended Embedded C++ features like namespaces or the standard template library in your source code, you must use this option to change the language to Extended Embedded C++. See *Extended Embedded C++*, page 42.



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

---

`--error_limit`    `--error_limit=n`

Use the `--error_limit` option to specify the number of errors allowed before the compiler stops the compilation. By default, 100 errors are allowed. *n* must be a positive number; 0 indicates no limit.

---

`--no_tbaa`    `--no_tbaa`

Use `--no_tbaa` to disable type-based alias analysis. When this options is not used, the compiler is free to assume that objects are only accessed through the declared type or through `unsigned char`.



This option is related to the **Optimization** options in the **C/C++ Compiler** category in the IAR Embedded Workbench.

## Description of type-based alias analysis

Type-based alias analysis means that the compiler assumes that an assignment, for example via pointers, to an object of one type does not change the value of objects of other types, except for `char` objects. This is according to the ISO/ANSI standard.

A C or C++ application that conforms to the ISO/ANSI standard thus accesses an object only by a modifiable value that has one of the following types:

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

By default, the compiler does not assume that objects are only accessed through the declared type or through `unsigned char`. However, at optimization level **High** the type-based alias analysis transformation is used, which means that the optimizer will assume that the program is standards compliant and the rules above will be used for determining what objects may be affected when a pointer indirection is used in an assignment.

Consider the following example:

```
short s;
unsigned short us;
long l;
unsigned long ul;
float f;

unsigned short *usptr;
char *cptr;

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

void test(float *fptr, long *lptr)
{
    /* May affect: */
    *lptr = 0;      /* l, ul */
    *fptr = 1.0;   /* f, a */
    *usptr = 4711; /* s, us, a */
    *cptr = 17;    /* s, us, l, ul, f, usptr, cptr, a */
}
```



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

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

```
short f(short *sptr, long *lptr)
{
    short x = *sptr;
    *lptr = 0;
    return *sptr + x;
}
```

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

Consider the following example, which is not ISO/ANSI compliant:

```
short fail()
{
    union
    {
        short s[2];
        long l;
    } u;
    u.s[0] = 4711;

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

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

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

---

## Descriptions of pragma directives

This section gives detailed reference information about new pragma directives.

---

```
#pragma basic_template_matching #pragma basic_template_matching
```

Use this pragma directive in front of a template function declaration to make the function fully memory-aware, in the rare cases where this is useful. This template function will then match the template without any modifications. Read more about memory attributes and templates in *Feature descriptions*, page 43.

### Example

```
#pragma basic_template_matching
template<typename T> void fun(T *);

fun((int __data_mem1 *) 0); // T = int __data_mem1
```

---

## Implementation-defined behavior

This section lists changes related to implementation-defined behavior.

### Source and execution character sets (5.2.1)

The source character set is the set of legal characters that can appear in source files. The default source character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the source character set will be the host computer's default character set.

The execution character set is the set of legal characters that can appear in the execution environment. The default execution character set is the standard ASCII character set. However, if you use the command line option `--enable_multibytes`, the execution character set will be the host computer's default character set. The IAR DLIB Library needs a multibyte character scanner to support a multibyte execution character set. The IAR CLIB Library does not support multibyte characters.

See *Locale*, page 22.

### Converting multibyte characters (6.1.3.4)

The only locale supported—that is, the only locale supplied with the IAR C/C++ Compiler—is the 'C' locale. If you use the command line option `--enable_multibytes`, the IAR DLIB Library will support multibyte characters if you add a locale with multibyte support or a multibyte character scanner to the library. The IAR CLIB Library does not support multibyte characters.

See *Locale*, page 22.

## IAR DLIB LIBRARY FUNCTIONS

The information in this section is valid only if the runtime library configuration you have chosen supports file descriptors. See the chapter *The DLIB runtime environment* for more information about runtime library configurations.

### **signal() (7.7.1.1)**

The signal part of the library is not supported.

**Note:** Low-level interface functions exist in the library, but will not perform anything. Use the template source code to implement application-specific signal handling. See *Signal and raise*, page 25.

### **Files (7.9.3)**

Whether a write operation on a text stream causes the associated file to be truncated beyond that point, depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 21.

#### **remove() (7.9.4.1)**

The effect of a remove operation on an open file depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 21.

#### **rename() (7.9.4.2)**

The effect of renaming a file to an already existing filename depends on the application-specific implementation of the low-level file routines. See *File input and output*, page 21.

### **Environment (7.10.4.4)**

The set of available environment names and the method for altering the environment list is described in *Environment interaction*, page 24.

#### **system() (7.10.4.5)**

How the command processor works depends on how you have implemented the `system` function. See *Environment interaction*, page 24.

### **The time zone (7.12.1)**

The local time zone and daylight savings time implementation is described in *Time*, page 25.

### **clock() (7.12.2.1)**

From where the system clock starts counting depends on how you have implemented the `clock` function. See *Time*, page 25.

# Library functions

This chapter gives an introduction to the C and C++ library functions. It also lists the header files used for accessing library definitions.

---

## Introduction

The IAR C/C++ Compiler provides two different libraries:

- IAR DLIB Library is a complete ISO/ANSI C and C++ library. This library also supports floating-point numbers in IEEE 754 format and it can be configured to include different levels of support for locale, file descriptors, multibytes, et cetera.
- IAR CLIB Library is a light-weight library, which is not fully compliant with ISO/ANSI C. Neither does it fully support floating-point numbers in IEEE 754 format or does it support Embedded C++.

Note that different customization methods are normally needed for these two libraries. For additional information, 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 additional information about library functions, see *Implementation-defined behavior*, page 56.

## HEADER FILES

Your application program gains access to library definitions through header files, which is incorporated using the `#include` directive. The definitions are divided into a number of different header files, each covering a particular functional area, letting you include just those that are required.

It is essential to include the appropriate header file before making any reference to its definitions. Failure to do so can cause the call to fail during execution, or generate error or warning messages at compile time or link time.

## LIBRARY OBJECT FILES

Most of the library definitions can be used without modification, that is, directly from the library object files that are supplied with the product. For information about how to choose a runtime library, see *Getting started using the runtime environment*, page 2. The linker will include only those routines that are required—directly or indirectly—by your application.

## REENTRANCY

A function that can be simultaneously invoked in the main application and in any number of interrupts is *reentrant*. A library function that uses statically allocated data is therefore not reentrant. Most parts of the DLIB library are reentrant, but the following functions and parts are not reentrant:

<code>atexit</code>	Needs static data
heap functions	Need static data for memory allocation tables
<code>strerror</code>	Needs static data
<code>strtok</code>	Designed by ISO/ANSI standard to need static data
I/O	Every function that uses files in some way. This includes <code>printf</code> , <code>scanf</code> , <code>getchar</code> , and <code>putchar</code> . The functions <code>sprintf</code> and <code>sscanf</code> are not included.

In addition, some functions share the same storage for `errno`. These functions are not reentrant, since an `errno` value resulting from one of these functions can be destroyed by a subsequent use of the function before it has been read. Among these functions are:

```
exp, exp10, ldexp, log, log10, pow, sqrt, acos, asin, atan2,
cosh, sinh, strtod, strtol, strtoul
```

Remedies for this are:

- Do not use non-reentrant functions in interrupt service routines
- Guard calls to a non-reentrant function by a mutex, or a secure region, etc.

---

## IAR DLIB Library

The IAR DLIB Library provides most of the important C and C++ library definitions that apply to embedded systems. These are of the following types:

- Adherence to a free-standing implementation of the ISO/ANSI standard for the programming language C. For additional information, see *Implementation-defined behavior*, page 56.
- Standard C library definitions, for user programs.

- Embedded C++ library definitions, for user programs.
- `CSTARTUP`, the module containing the start-up code. It is described in the *The DLIB runtime environment* chapter in this guide.
- Runtime support libraries; for example, low-level floating-point routines.
- Intrinsic functions, allowing low-level use of device-specific features. See *IAR C/C++ Compiler Reference Guide* for more information.

## C HEADER FILES

This section lists the header files specific to the DLIB library C definitions. Header files may additionally contain target-specific definitions; these are documented in the *IAR C/C++ Compiler Reference Guide*.

The following table lists the C header files:

Header file	Usage
<code>assert.h</code>	Enforcing assertions when functions execute
<code>ctype.h</code>	Classifying characters
<code>errno.h</code>	Testing error codes reported by library functions
<code>float.h</code>	Testing floating-point type properties
<code>iso646.h</code>	Using Amendment 1— <code>iso646.h</code> standard header
<code>limits.h</code>	Testing integer type properties
<code>locale.h</code>	Adapting to different cultural conventions
<code>math.h</code>	Computing common mathematical functions
<code>setjmp.h</code>	Executing non-local goto statements
<code>signal.h</code>	Controlling various exceptional conditions
<code>stdarg.h</code>	Accessing a varying number of arguments
<code>stdbool.h</code>	Adds support for the <code>bool</code> data type in C.
<code>stddef.h</code>	Defining several useful types and macros
<code>stdio.h</code>	Performing input and output
<code>stdlib.h</code>	Performing a variety of operations
<code>string.h</code>	Manipulating several kinds of strings
<code>time.h</code>	Converting between various time and date formats
<code>wchar.h</code>	Support for wide characters
<code>wctype.h</code>	Classifying wide characters

Table 13: Traditional standard C header files—DLIB

## C++ HEADER FILES

This section lists the C++ header files.

### Embedded C++

The following table lists the Embedded C++ header files:

Header file	Usage
<code>complex</code>	Defining a class that supports complex arithmetic
<code>exception</code>	Defining several functions that control exception handling
<code>fstream</code>	Defining several I/O streams classes that manipulate external files
<code>iomanip</code>	Declaring several I/O streams manipulators that take an argument
<code>ios</code>	Defining the class that serves as the base for many I/O streams classes
<code>iosfwd</code>	Declaring several I/O streams classes before they are necessarily defined
<code>iostream</code>	Declaring the I/O streams objects that manipulate the standard streams
<code>istream</code>	Defining the class that performs extractions
<code>new</code>	Declaring several functions that allocate and free storage
<code>ostream</code>	Defining the class that performs insertions
<code>sstream</code>	Defining several I/O streams classes that manipulate string containers
<code>stdexcept</code>	Defining several classes useful for reporting exceptions
<code>streambuf</code>	Defining classes that buffer I/O streams operations
<code>string</code>	Defining a class that implements a string container
<code>stringstream</code>	Defining several I/O streams classes that manipulate in-memory character sequences

Table 14: Embedded C++ header files

The following table lists additional C++ header files:

Header file	Usage
<code>fstream.h</code>	Defining several I/O stream classes that manipulate external files
<code>iomanip.h</code>	Declaring several I/O streams manipulators that take an argument
<code>iostream.h</code>	Declaring the I/O streams objects that manipulate the standard streams
<code>new.h</code>	Declaring several functions that allocate and free storage

Table 15: Additional Embedded C++ header files—DLIB



## Extended Embedded C++ standard template library

The following table lists the Extended Embedded C++ standard template library (STL) header files:

Header file	Description
<code>algorithm</code>	Defines several common operations on sequences
<code>deque</code>	A deque sequence container
<code>functional</code>	Defines several function objects
<code>hash_map</code>	A map associative container, based on a hash algorithm
<code>hash_set</code>	A set associative container, based on a hash algorithm
<code>iterator</code>	Defines common iterators, and operations on iterators
<code>list</code>	A doubly-linked list sequence container
<code>map</code>	A map associative container
<code>memory</code>	Defines facilities for managing memory
<code>numeric</code>	Performs generalized numeric operations on sequences
<code>queue</code>	A queue sequence container
<code>set</code>	A set associative container
<code>slist</code>	A singly-linked list sequence container
<code>stack</code>	A stack sequence container
<code>utility</code>	Defines several utility components
<code>vector</code>	A vector sequence container

Table 16: Standard template library header files

## Using standard C libraries in C++

The C++ library works in conjunction with 15 of the header files from the standard C library, sometimes with small alterations. The header files come in two forms—new and traditional—for example, `cassert` and `assert.h`.

The following table shows the new header files:

Header file	Usage
<code>cassert</code>	Enforcing assertions when functions execute
<code>cctype</code>	Classifying characters
<code>cerrno</code>	Testing error codes reported by library functions
<code>cfloat</code>	Testing floating-point type properties
<code>climits</code>	Testing integer type properties

Table 17: New standard C header files—DLIB

Header file	Usage
<code>locale</code>	Adapting to different cultural conventions
<code>cmath</code>	Computing common mathematical functions
<code>csetjmp</code>	Executing non-local goto statements
<code>csignal</code>	Controlling various exceptional conditions
<code>cstdarg</code>	Accessing a varying number of arguments
<code>cstddef</code>	Defining several useful types and macros
<code>stdio</code>	Performing input and output
<code>stdlib</code>	Performing a variety of operations
<code>cstring</code>	Manipulating several kinds of strings
<code>ctime</code>	Converting between various time and date formats

Table 17: New standard C header files—DLIB (Continued)

## IAR CLIB Library

The IAR CLIB Library provides most of the important C 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.
- `CSTARTUP`, the single program module containing the start-up code. It is described in the *The CLIB runtime environment* chapter in this guide.
- Runtime support libraries; for example low-level floating-point routines.
- Intrinsic functions, allowing low-level use of device-specific features. See the *IAR C/C++ Compiler Reference Guide* for more information.

### LIBRARY DEFINITIONS SUMMARY

This section lists the header files. Header files may additionally contain target-specific definitions.

Header file	Description
<code>assert.h</code>	Assertions
<code>ctype.h*</code>	Character handling
<code>iccbutl.h</code>	Low-level routines
<code>math.h</code>	Mathematics
<code>setjmp.h</code>	Non-local jumps
<code>stdarg.h</code>	Variable arguments

Table 18: IAR CLIB Library header files

Header file	Description
<code>stdio.h</code>	Input/output
<code>stdlib.h</code>	General utilities
<code>string.h</code>	String handling

*Table 18: IAR CLIB Library header files (Continued)*

**\* The functions `isxxx`, `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.**

The following table shows header files that do not contain any functions, but specify various definitions and data types:

Header file	Description
<code>errno.h</code>	Error return values
<code>float.h</code>	Limits and sizes of floating-point types
<code>limits.h</code>	Limits and sizes of integral types
<code>stdbool.h</code>	Adds support for the <code>bool</code> data type in C
<code>stddef.h</code>	Common definitions including <code>size_t</code> , <code>NULL</code> , <code>ptrdiff_t</code> , and <code>offsetof</code>

*Table 19: Miscellaneous IAR CLIB Library header files*



**A**

algorithm (STL header file) . . . . . 63

applications

- initializing . . . . . 17, 39
- terminating . . . . . 17, 39

assembler directives

- ENDMOD . . . . . 30
- MODULE . . . . . 30
- REQUIRE . . . . . 31
- RSEG . . . . . 30

assembler labels

- ?C\_EXIT . . . . . 40
- ?C\_GETCHAR . . . . . 40
- ?C\_PUTCHAR . . . . . 40

assert.h (library header file) . . . . . 61, 64

assumptions (programming experience) . . . . . ix

**B**

basic\_template\_matching (pragma directive) . . . . . 48, 56

bool (data type)

- adding support for in CLIB . . . . . 65
- adding support for in DLIB . . . . . 61
- making available in C . . . . . 32

bubble sort algorithm, adding support for . . . . . 33

**C**

C header files . . . . . 61

cassert (library header file) . . . . . 63

cast operators

- in Extended EC++ . . . . . 42
- missing from Embedded C++ . . . . . 42

cctype (library header file) . . . . . 63

cerrno (library header file) . . . . . 63

cfloat (library header file) . . . . . 63

class memory (extended EC++) . . . . . 43

class template

- partial specialization matching (extended EC++) . . . . . 47

classes . . . . . 43

CLIB . . . . . 2, 64

- documentation . . . . . 59
- header files . . . . . 59
- library object files . . . . . 60

climits (library header file) . . . . . 63

locale (library header file) . . . . . 64

\_\_close (library function) . . . . . 22

cmath (library header file) . . . . . 64

code, excluding when linking . . . . . 31

compiler options

- ec++. . . . . 53
- error\_limit . . . . . 53
- no\_tbaa . . . . . 53

complex numbers, supported in Embedded C++ . . . . . 42

complex (library header file) . . . . . 62

computer style, typographic convention . . . . . ix

configuration symbols, in library configuration files . . . . . 14

const\_cast() (cast operator) . . . . . 42

conventions, typographic . . . . . ix

copyright notice . . . . . ii

csetjmp (library header file) . . . . . 64

csignal (library header file) . . . . . 64

CSTACK (segment) . . . . . 26

cstartup . . . . . 39

- implementation . . . . . 40

cstartup, customizing . . . . . 18

cstartup, implementation . . . . . 30

cstdlib (library header file) . . . . . 64

cstdlib (library header file) . . . . . 64

cstdio (library header file) . . . . . 64

cstdlib (library header file) . . . . . 64

cstring (library header file) . . . . . 64

ctime (library header file) . . . . . 64

ctype.h (library header file) . . . . . 61, 64

customization, \_\_low\_level\_init . . . . . 18

C++

- features excluded from EC++ . . . . . 41

*See also Embedded C++ and Extended Embedded C++ terminology* . . . . . ix

C++ header files . . . . . 62

C-SPY

- low-level interface (CLIB) . . . . . 40
- low-level interface (DLIB) . . . . . 29
- STL container support . . . . . 50

?C\_EXIT (assembler label) . . . . . 40

?C\_GETCHAR (assembler label) . . . . . 40

?C\_PUTCHAR (assembler label) . . . . . 40

C99 standard, added functionality from . . . . . 32

## D

data, excluding when linking . . . . . 31

date (library function), configuring support for . . . . . 25

delete operator (extended EC++) . . . . . 46

deque (STL header file) . . . . . 63

destructors and interrupts, using . . . . . 51

disclaimer . . . . . ii

DLIB . . . . . 2, 60

- documentation . . . . . 59
- using . . . . . 3

document conventions . . . . . ix

dynamic initialization . . . . . 16, 36–39

## E

EC++ header files . . . . . 62

--eec++ (compiler option) . . . . . 53

Embedded C++ . . . . . 41

- differences from C++ . . . . . 41
- language extensions . . . . . 41
- overview . . . . . 41

ENDMOD (assembler directive) . . . . . 30

environment, runtime . . . . . 35

errno.h (library header file) . . . . . 61, 65

exception handling, missing from Embedded C++ . . . . . 41

exception (library header file) . . . . . 62

export keyword, missing from Extended EC++ . . . . . 47

Extended Embedded C++ . . . . . 42

- enabling . . . . . 53
- standard template library (STL) . . . . . 63

## F

float.h (library header file) . . . . . 61, 65

\_formatted\_write (library function) . . . . . 10, 37

fstream (library header file) . . . . . 62

fstream.h (library header file) . . . . . 62

function template parameter deduction (extended EC++) . . . . . 48

functional (STL header file) . . . . . 63

functions, reentrancy (DLIB) . . . . . 60

## G

getchar (library function) . . . . . 36

getenv (library function), configuring support for . . . . . 24

getzone (library function), configuring support for . . . . . 25

guidelines, reading . . . . . ix

## H

hash\_map (STL header file) . . . . . 63

hash\_set (STL header file) . . . . . 63

header files

- assert.h . . . . . 64
- C . . . . . 61
- CLIB . . . . . 59
- ctype.h . . . . . 64
- C++ . . . . . 62
- EC++ . . . . . 62
- errno.h . . . . . 65
- float.h . . . . . 65
- iccbutl.h . . . . . 64
- limits.h . . . . . 65
- math.h . . . . . 64
- setjmp.h . . . . . 64

stdarg.h	64
stdbool.h	61, 65
stddef.h	65
stdio.h	65
stdlib.h	65
STL	63
string.h	65
heap	
changing default size (cl)	28
changing default size (IDE)	28
size	26–28
HEAP (segment)	28

**I**

iccbutl.h (library header file)	64
implementation	
cstartup	30
inheritance, in Embedded C++	41
initialization	
dynamic	16, 36–39
integer characteristics, adding	32
interrupts and EC++ destructors, using	51
iomanip (library header file)	62
iomanip.h (library header file)	62
ios (library header file)	62
iosfwd (library header file)	62
iostream (library header file)	62
iostream.h (library header file)	62
ISO/ANSI C	2, 59
C++ features excluded from EC++	41
iso646.h (library header file)	61
istream (library header file)	62
iterator (STL header file)	63

**L**

language extensions	
Embedded C++	41

libraries	
runtime	35
standard template library	63
library configuration file, modifying	15
library documentation	59
library features, missing from Embedded C++	42
library functions	59
getchar	36
printf	
choosing formatter in DLIB	10
printf, choosing formatter in CLIB	37
putchar	36
remove	22
rename	22
scanf	
choosing formatter in DLIB	11
scanf, choosing formatter in CLIB	38
sprintf	
choosing formatter in DLIB	10
sprintf, choosing formatter in CLIB	37
sscanf, choosing formatter in CLIB	38
summary	61, 64
__close	22
__lseek	22
__open	22
__read	22
__write	22
library object files, CLIB	60
limits.h (library header file)	61, 65
list (STL header file)	63
locale.h (library header file)	61
__low_level_init, customizing	18
__lseek (library function)	22

**M**

map (STL header file)	63
math.h (library header file)	32, 61, 64
__medium_write (library function)	37

memory management, type-safe	41
memory (STL header file)	63
MODULE (assembler directive)	30
modules, assembler	30
multiple inheritance, missing from Embedded C++	41
mutable attribute, in Extended EC++	50

## N

namespace support	
in Extended EC++	42, 50
missing from Embedded C++	42
new operator (extended EC++)	46
new (library header file)	62
new.h (library header file)	62
NULL	65
numeric (STL header file)	63

## O

offsetof	65
__open (library function)	22
optimization	
type-based alias analysis	54
--no_tbaa	53
ostream (library header file)	62

## P

parameters	
typographic convention	ix
polymorphism, in Embedded C++	41
pragma directives	
basic_template_matching	48, 56
prerequisites (programming experience)	ix
printf (library function)	
choosing formatter in CLIB	37
choosing formatter in DLIB	10
programming experience, required	ix

ptrdiff_t (integer type)	65
putchar (library function)	36

## Q

queue (STL header file)	63
-------------------------	----

## R

raise (library function), configuring support for	25
__read (library function)	22
read formatter, selecting	12, 38
reading guidelines	ix
reentrancy (DLIB)	60
reinterpret_cast() (cast operator)	42
remove (library function)	22
rename (library function)	22
REQUIRE (assembler directive)	31
RSEG (assembler directive)	30
rti support, missing from STL	42
runtime environment	35
runtime libraries	35
introduction	59
runtime type information, missing from Embedded C++	41

## S

scanf (library function)	
choosing formatter in CLIB	38
choosing formatter in DLIB	11
segment parts, unused	31
segments	
CSTACK	26
HEAP	28
set (STL header file)	63
setjmp.h (library header file)	61, 64
signal (library function), configuring support for	25
signal.h (library header file)	61
size_t (integer type)	65



slist (STL header file) . . . . . 63  
 \_small\_write (library function) . . . . . 37  
 sprintf (library function)  
     choosing formatter in CLIB . . . . . 37  
     choosing formatter in DLIB . . . . . 10  
 sscanf (library function)  
     choosing formatter in CLIB . . . . . 38  
     choosing formatter in DLIB . . . . . 11  
 ostream (library header file) . . . . . 62  
 stack  
     changing default size (from command line) . . . . . 26  
     changing default size (in Embedded Workbench) . . . . . 26  
     size . . . . . 27  
 stack (STL header file) . . . . . 63  
 standard template library (STL)  
     in Extended EC++ . . . . . 42, 49, 63  
     missing from Embedded C++ . . . . . 42  
 startup, system . . . . . 17, 39  
 static\_cast() (cast operator) . . . . . 42  
 std namespace, missing from EC++  
     and Extended EC++ . . . . . 50  
 stdarg.h (library header file) . . . . . 61, 64  
 stdbool.h (library header file) . . . . . 32, 61, 65  
 stddef.h (library header file) . . . . . 61, 65  
 stderr . . . . . 22  
 stdexcept (library header file) . . . . . 62  
 stdin . . . . . 22  
 stdint.h (library header file) . . . . . 32  
 stdio.h (library header file) . . . . . 33, 61, 65  
 stdlib.h (library header file) . . . . . 33, 61, 65  
 stdout . . . . . 22  
 streambuf (library header file) . . . . . 62  
 streams, supported in Embedded C++ . . . . . 42  
 string (library header file) . . . . . 62  
 strings, supported in Embedded C++ . . . . . 42  
 string.h (library header file) . . . . . 61, 65  
 strstream (library header file) . . . . . 62  
 strtod (library function), configuring support for . . . . . 26  
 system startup . . . . . 17, 39  
 system termination . . . . . 17, 39

system (library function), configuring support for . . . . . 24

## T

template support

    in Extended EC++ . . . . . 42, 47  
     missing from Embedded C++ . . . . . 41  
 Terminal I/O window, in C-SPY . . . . . 40  
 termination, system . . . . . 17, 39  
 terminology . . . . . ix  
 this pointer, referring to a class object (extended EC++) . . . . . 43  
 time (library function), configuring support for . . . . . 25  
 time.h (library header file) . . . . . 61  
 Type-based alias analysis, optimization . . . . . 54  
 type-safe memory management . . . . . 41  
 typographic conventions . . . . . ix

## U

utility (STL header file) . . . . . 63

## V

vector (STL header file) . . . . . 63

## W

wchar.h (library header file) . . . . . 61  
 wctype.h (library header file) . . . . . 61  
 \_write (library function) . . . . . 22  
 write formatter, selecting . . . . . 37–38

## Symbols

--eec++ (compiler option) . . . . . 53  
 --error\_limit (compiler option) . . . . . 53  
 --no\_tbaa (compiler option) . . . . . 53  
 ?C\_EXIT (assembler label) . . . . . 40  
 ?C\_GETCHAR (assembler label) . . . . . 40

?C_PUTCHAR (assembler label) . . . . .	40
__close (library function) . . . . .	22
__low_level_init, customizing . . . . .	18
__lseek (library function) . . . . .	22
__open (library function) . . . . .	22
__read (library function) . . . . .	22
__write (library function) . . . . .	22
_formatted_write (library function) . . . . .	10, 37
_medium_write (library function) . . . . .	37
_small_write (library function) . . . . .	37