



**IAR Embedded
Workbench**

Safety Guide

COPYRIGHT NOTICE

© 2012–2018 IAR Systems AB.

No part of this document may be reproduced without the prior written consent of IAR Systems AB. 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 Systems, IAR Embedded Workbench, Embedded Trust, IAR Connect, C-SPY, C-RUN, C-STAT, IAR Visual STATE, IAR KickStart Kit, I-jet, I-jet Trace, I-scope, IAR Academy, IAR, and the logotype of IAR Systems are trademarks or registered trademarks owned by IAR Systems AB.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

Adobe and Acrobat Reader are registered trademarks of Adobe Systems Incorporated.

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

EDITION NOTICE

Sixth edition: November 2018

Part number: SafetyEW-6

Internal reference: INIT

Contents

Using a build toolchain in functional safety projects	5
Introduction	5
The scope of tool usage and assumptions of use	7
System and environment considerations	7
Language standards compliance	7
IAR language extensions	9
MISRA C standards and the MISRA C checker	10
Hazard and operability analysis	11
Device-specific support files	14
Compatibility between different versions of the same toolchain	15
Compatibility with other toolchains	15
General guidelines on MCU self-check strategies	16
Installation, commissioning, operation, and maintenance ...	17
Setting up the build environment	18
Debug mode, release mode, and build configurations	18
Build options	19
Stack depth considerations	20
Linker configuration	21
Add-on analysis tools	23
Implementation and coding considerations	24
Optimization modes	24
Integral type selection	24
Floating-point arithmetic	25
Functions	25
Global symbols	26
Const and volatile	26
Pointers	27
The C/C++ standard libraries	27
The C standard library	27

Using a build toolchain in functional safety projects

This guide contains information that is relevant when you use IAR Embedded Workbench® for creating high-integrity software that has functional safety requirements. The guide is a complement to the IAR Embedded Workbench user guides.

I Introduction

The purpose of this guide is to highlight issues that you should consider when you use the build toolchain in IAR Embedded Workbench® for projects with functional safety requirements or more general high-integrity requirements. The focus of this guide will be on different topics covered by the international standards IEC 61508 edition 2.0 and ISO 26262. However, the guide applies to any development project where functional safety or high integrity are part of the requirements.

For each topic, IAR Systems provides various notes of advice that are relevant for the build toolchain—such advice is marked with **Advice**. This guide encourages you to consider the relevance of each listed advice, and to discuss the implications related to your application with project stakeholders, and when applicable, the appointed functional safety assessor.

Note: Some notes of advice only apply to a specific microcontroller or to a specific feature set that might not be available in the toolchain you are using. If so, this will be mentioned.

Note: The demands of the mentioned standards are seldom absolute requirements—almost any practice, decision, or deviation can be justified as long as there is valid and sufficient rationale for it. Furthermore, the decisions and activities proposed by the standards differ depending on the integrity level of the project.

The guide assumes a basic understanding of the requirements and proposed best practices brought forward by a relevant safety standard with regards to implementation languages, implementation and test practices, and build toolchains.

References to sections and paragraphs of IEC 61508 are given for further background information.

The information in this guide is valid for the following IAR Embedded Workbench products and associated instruction sets:

IAR Embedded Workbench for *	Version
Arm	6.40.1 and later
AVR	6.11 and later
AVR32	4.10 and later
S08	1.20 and later
HCS12	3.20 and later
CF	1.23 and later
CR16C	3.10 and later
RL78	1.20 and later
RX	2.40 and later
RH850	1.10 and later
M16C	3.50 and later
78K	4.71 and later
H8	2.20 and later
M32C	2.30 and later
R32C	1.31 and later
V850	3.81 and later
SH	2.20 and later
STM8	1.30 and later
430	4.50 and later
8051	8.10 and later

Table 1: IAR Embedded Workbench products

*** The information in this guide might also be useful for other IAR Embedded Workbench products.**

Note: In this guide, if your product package includes the XLINK Linker, any reference to the *IAR C/C++ Development Guide* refers to the *IAR C/C++ Compiler Reference Guide* and the *IAR Linker and Library Tools Reference Guide*. If your product package includes the ILINK Linker, any reference to the *IAR C/C++ Development Guide* refers only to that guide.

1.1 THE SCOPE OF TOOL USAGE AND ASSUMPTIONS OF USE

The IAR Embedded Workbench is a toolchain for writing, translating, debugging, and deploying applications written in C/C++ or assembler language where the target architecture typically is a microcontroller.

The build chain consists of the following components with their respective use cases:

- *The compiler:* Translates C and C++ programs to object code. The compiler can produce diagnostic messages and list files to help track programming errors. The compiler can also check conformance to MISRA C:2004 or MISRA C:1998 rules, and can for some target architectures produce meta information used by the linker to compute worst-case stack usage.
- *The assembler:* Translates assembler language input to object code. The assembler produces diagnostic messages and list files to help track programming errors.
- *The linker:* Accepts files in object format and resolves outstanding references to function calls, variables, and assembler level jumps. As output, the linker can produce a fully resolved binary image that can be downloaded to the target system, a fully resolved binary image that contains debug information which can be used in a debugger to debug the application, or library files that can be linked to other object files to produce a fully resolved binary image.
- Depending on the IAR Embedded Workbench product you are using, you might be able to enable the optional add-on tools C-STAT for static analysis and C-RUN for runtime error checking. These add-on tools are not strictly part of the build chain, although they can be used as an integral part of your quality assurance.

2 System and environment considerations

This section covers topics that you should consider early in your project setup.

2.1 LANGUAGE STANDARDS COMPLIANCE

The build toolchain that is part of IAR Embedded Workbench supports C and C++ programming languages, including some dialects. For information about the supported languages and their dialects, see the IAR language overview in the *IAR C/C++ Development Guide*.

This guide will focus on advice applicable for the C language. The advice is generally also applicable for C++, but if you have chosen C++ for your project it is especially important you consider the advice given in the safety standards that govern your project on how object-oriented features can be used safely. See for example, *IEC 61508-7, Annex G*.

Before you start a project with functional safety requirements, consider these issues:

- If you have decided to use C/C++, safety standards generally advise you to use plain C with a suitable language subset, or C++ with a suitable language subset. The build toolchain supports both C99 (Standard C) and C89. See *IEC 61508-7, table C.1* and *IEC 61508-3, section 7.4.4* for advice on language selection.

Advice 2.1-1: Make an informed decision on what language, language dialect, and language subset you will use.

Advice 2.1-2: Consider available tools that enforce the use of suitable language subsets, for example, MISRA C rule checkers.

- If you have decided to use Standard C, safety standards generally advise you to use Standard C without any extensions. However, it can be useful from an implementation and code quality perspective to use various language extensions, for example, hardware-specific keywords or intrinsic functions.

Advice 2.1-3: Consider isolating the use of language extensions to specific modules to maximize the portability of the code base and simplify the use of compiler-independent code analysis tools. Such isolation also facilitates proper testing of the code that depends on the language extensions.

- The C standard library provided with IAR Embedded Workbench contains functionality mandated by Standard C. The library is delivered by a third-party vendor and adapted by IAR Systems to suit typical resource-constrained microcontrollers. Different standards for software development with functional safety requirements put different constraints on the use of such pre-existing code and functionality. For certain functionality, such as I/O and threading, the library depends on a low-level interface that you must implement yourself. For more information about using DLIB in a functional safety project, see *The C/C++ standard libraries*, page 27. For more information about the DLIB runtime environment, see the *IAR C/C++ Development Guide*.

Advice 2.1-4: Consider how pre-existing code should be treated and tested.

Advice 2.1-5: Consider how to implement and test the parts of the low-level interface that your application needs.

- The supplied libraries also contain low-level code to handle system startup and exit, as well as runtime routines needed by compiled code, for example, floating-point arithmetic. The IAR Systems implementation adheres to a freestanding implementation of Standard C/C++, which among other things means that you can control the code for your application's system startup and exit.

Advice 2.1-6: Consider not using the default code for system startup and exit, but instead use your own versions to fully control the behavior of your application.

Advice 2.1-7: If you decide to use your own versions of startup and exit code, consider how this code should be handled and tested. For more information, read about system startup and termination in the *IAR C/C++ Development Guide*.

Advice 2.1-8: Read about implementation-defined behavior for Standard C in the *IAR C/C++ Development Guide*.

2.2 IAR LANGUAGE EXTENSIONS

The IAR compiler supports many different language extensions that can be divided into two categories:

- Extensions for embedded system programming
- Relaxations of Standard C rules.

The first set of extensions covers additions to the language that make it easier to deal with the specific hardware you are using. This category includes extensions like type and object attributes for memory placement and memory area selection, alignment control, interrupt routine definitions, relaxation of permitted types in `enum` and bit field definitions, intrinsic functions to use special features of the target CPU, etc.

The second category consists of relaxations of some restrictive Standard C rules. This relaxation of rules is similar to how many other C compilers behave by default. Extensions include, for example, conversions between `void` pointers and other pointer types in some situations, assignment and pointer difference between types that are interchangeable but not identical, single value initializations of arrays, structs, etc.

Safety standards generally advise that you should not rely upon language extensions, implementation-defined behavior, or undefined behavior. However, it might be difficult to develop high-quality embedded systems without relying on tool-specific functionality, for example, to access memory-mapped peripheral devices, and to create interrupt handlers. Another example is accessing special features of the CPU that do not map naturally to the C/C++ language without using special compiler-provided *intrinsic* functions.

Advice 2.2-1: For development of software with functional safety requirements, there can be a trade-off between using a well-tested language extension and implementing the same functionality by other means. When you can choose between using a language extension or implementing similar functionality in your own source code, discuss with your assessor and other stakeholders on how to proceed. When deciding, consider factors like maintainability, portability, usability, and readability.

Advice 2.2-2: For an overview of the different levels of available language extensions, read about using C and C++ in the *IAR C/C++ Development Guide*.

Advice 2.2-3: In your coding standard, include advice about how to use language extensions.

Advice 2.2-4: Partition source files and header files to isolate the use of language extensions as much as possible. This might include creating a formal hardware abstraction layer.

Advice 2.2-5: The build chain does not by default enable generation of *remarks*—the least severe type of diagnostic messages. Enabling remarks can give you additional hints on language constructs that are non-standard, non-portable, or might produce unintended results.

If you need to implement a piece of functionality directly in assembler language, see the *IAR Assembler Reference Guide* for a description of the accepted assembler syntax for your product.

Advice 2.2-6: Consider that assembler syntax is often vendor-specific, which means that assembler code written for a specific toolchain is seldom directly portable to another toolchain for the same target architecture.

Advice 2.2-7: Due to the lack of portability and the risk of introducing errors that are hard to find in the interaction between the high-level C/C++ source code and the assembler code, consider carefully if there are alternatives to using assembler language. Read about mixing C and assembler language in the *IAR C/C++ Development Guide*.

2.3 MISRA C STANDARDS AND THE MISRA C CHECKER

Safety standards generally require you to adopt a coding standard to deal with hazardous constructs in the programming language you select. For C, it is recommended that your coding standard incorporates the MISRA C rules, or another set of rules with similar intent.

Note: The current revision of the MISRA guidelines is MISRA C:2012, which is currently not supported by the built-in MISRA C checker in the compiler. However, the C-STAT static analysis add-on tool supports MISRA C:2012, as well as MISRA C:2004 and MISRA C++:2008.

Advice 2.3-1: If you decide on any of the MISRA C standards, the MISRA C guidelines recommend using at least one MISRA C checker to get the best possible analysis quality.

Advice 2.3-2: If you are using the full version of IAR Embedded Workbench, consider setting up the built-in MISRA C checker for the previous MISRA C:2004 or MISRA C:1998 standards to be part of the build process. In this case, you should also consider using another MISRA C checker—because the MISRA C checker in IAR Embedded Workbench is based on the same technology as the IAR C/C++ Compiler and has the same view of the source code as the compiler, it is important to rule out common modes of error by using more than one checker.

Advice 2.3-3: Get a copy of the relevant MISRA C standard to review the rationale for each rule. Aim to adhere to all MISRA C rules in your project and make deviations a rare exception.

Advice 2.3-4: If you decide to ignore certain rules globally, consider configuring the MISRA checker to ignore those rules to minimize irrelevant messages. For example, both C-STAT and the built-in MISRA checkers can be configured to ignore specific rules globally and for individual files.

2.4 HAZARD AND OPERABILITY ANALYSIS

Safety standards generally require you to perform a *hazard and operability analysis* (HAZOP analysis) of potential tool failures. See *IEC 61508-3, section 7.4.4.5* and *IEC 61508-7, section C.6.2*.

In the context of the build toolchain, a HAZOP analysis can be interpreted as an analysis of possible failure modes.

Advice 2.4-1a: Consider the following possible failures—the different types of diagnostic messages—when setting up the tools environment, and when planning the test and validation activities:

- Remarks

Using C/C++ in a non-standard-compliant way will generate a remark—minor remarks are only produced when the `--strict` compiler option is used. Other sources for remarks are source constructs that potentially might cause problems in the application. To enable remarks, use the `--remarks` compiler option.

- Warning messages

Warning messages generally indicate that the toolchain has identified potentially erroneous or ambiguous behavior in your source code, in the compile stage or link process.

Advice 2.4-1: Strive for clean builds with no warning messages.

Advice 2.4-2: Consider having guidelines on when and how it is acceptable to ignore tool warnings.

Advice 2.4-3: Consider enabling the proper build options to treat all warnings as errors, at least for release builds and integration builds.

Advice 2.4-4: Document and sign off any particular warning that must be ignored to accomplish the desired effect.

Note: The absence of warnings does not indicate that your code is free from potential erroneous or ambiguous constructs. To avoid such constructs, IEC 61508 recommends the use of a well-defined language subset, like the MISRA C:2012 or MISRA C:2004 rules.

- Error messages

The toolchain will not produce code for modules that produce error messages. Therefore, error messages are favorable from a functional safety perspective as they cannot be ignored.

- Option settings not treated correctly

Advice 2.4-5: Consider verifying that no options set explicitly in the IAR Embedded Workbench IDE are ignored by the build tool. To verify this, make sure that each option you have set is listed as part of the active command line in the build tool list files.

Advice 2.4-6: Examine any implicitly set options to make sure that their effects are in line with the project goals. For example, when some optimization settings are enabled, a set of dependent functionality will be assumed. That functionality can be turned off by explicit options.

For example, for a High optimization level, a set of transformations will be implicitly enabled and must be explicitly disabled by a specific option. The `--no_cse` option is an example of such an option.

Advice 2.4-7: In the IAR Embedded Workbench IDE, the **Options** dialog box verifies that incompatible options cannot be active at the same time. For options that you specify directly on the command line, the toolchain will warn about conflicting settings. For some options, however, it is impossible for the tool to know whether dependent options are allowed or not. For example, if you select the CPU core explicitly instead of specifying a specific device, there is a risk that you enable core-specific features that are not available on the device you are using. Review your option settings to make sure the correct functionality is achieved.

- Internal errors

Internal errors are indications from the toolchain that it has detected an internal inconsistency that prevents the tool from generating the correct output. An internal error can be the result of unexpected erroneous input to the tool, or a malfunctioning of the tool in response to correctly formed input.

Advice 2.4-8: If you receive an internal error from the build toolchain, you should report it as a bug to your IAR Systems support contact or your reseller. Inspect the input that caused the internal error to determine if the input is supposed to be valid according to language standards and other constraining factors, or if the input is erroneous.

- Silent errors

A silent error is a tool error that is not reported by any kind of diagnostic message, and therefore has the potential to silently produce malfunctioning code. This type of

error is of course the most problematic kind of tool error, and as such should be given some extra consideration during test and validation planning.

Advice 2.4-9: Consider employing these strategies to prevent and catch this kind of issue:

- Regularly access updated information from the tool vendor about newly found and fixed issues in the toolchain. This information is available in release notes for new versions. For demanding projects, consider approaching your tool vendor for a special support agreement to get the most up-to-date information on known issues.
- Create different build configurations that differ in, for example, compiler optimization settings, linker configuration, and debugger setup. Each build configuration is then subjected to the same set of tests the build configuration used for release builds, and any discrepancies are analyzed. A side effect of this kind of testing is that erroneous source code that happens to work with one set of build options can be found. As an example, consider a piece of code that does not use the `volatile` keyword correctly—this code might execute tests correctly for low compiler optimization levels, but sometimes break on higher optimization levels.

Advice 2.4-10: As a minimum, consider a matrix of configurations exercising combinations of the following properties:

- Maximum and minimum optimization levels.
- For a CPU architecture that can accept different instruction sets, consider adding at least one test configuration for the instruction set that will not be used for release builds.
- If the C standard library is used, configure the runtime library in different ways to validate that application behavior is correct for the chosen configurations.
- If your application uses floating-point arithmetic and the chosen CPU contains an FPU unit, consider disabling the FPU support in one test configuration.
- Different test coverage metrics collected on unit tests and integration tests help direct test efforts to achieve the confidence level needed for correct functionality. Employ coverage-directed testing to validate correct behavior of the implemented functionality, and also efficiently discover potential tool errors.

Note: Some safety standards require 100% code coverage for the selected coverage metric. See for example, *IEC 61508-3, table B.2* and *IEC 61508-7, section C.5.8*.

- Strive for code that does not depend on any implementation-defined behavior, or any undefined behavior as defined by the language standard. Furthermore, strive

for code that is clean with respect to the MISRA C rule set or the selected coding standard.

2.5 DEVICE-SPECIFIC SUPPORT FILES

The product package contains support files for various devices. Depending on the product you are using, the number of supported devices can range from less than a hundred up to several thousand.

Files needed to support a specific target device are usually made up of these types:

- *I/O definition header file*: Defines mnemonic names for peripheral units and memory mapped peripheral device registers, so they can be accessed from the code with names resembling the names in the user documentation.
- *Device definition file for the C-SPY debugger*: Defines, for example, device-specific memory regions that give the debugger the possibility to check memory accesses and other functionality.
- *Linker configuration file*: Instructs the linker on how to place code and data segments for a specific device or device family.
- *Flash loader support*: Device-specific or device family-specific debugger plugin that downloads code to flash.
- *C-SPY macro file*: A general macro file that is sometimes used for setting up specific functionality of the debugger or device prior to execution of code.

All such support files are to be considered as convenience files. The toolchain can be used without support files, and you can always supply the needed information or work around the need for it. The build toolchain does not depend on the support files, and you are encouraged to create, for example, your own linker configuration file.

Note: Often, the device manufacturer supplies these files with minimal file validation. However, errors in support files are easily detected as they usually make the hardware behave incorrectly, or make it not work at all.

Advice 2.5-1: Make sure you have a clear picture of the support file dependencies for your project. Discuss with project stake holders and, if applicable, your assessor on how to treat these files and whether special measures are needed to test their functionality, besides integration testing.

Advice 2.5-2: If you are planning on using a new device that is not covered by any support files in your product package, you can often retrieve the required support files from a newer, certified, or non-certified version of IAR Embedded Workbench for your target CPU, or in certain cases directly from your device manufacturer. For reasons of reliability and accumulated confidence from use, however, consider using a well-known device that is already on the market.

2.6 COMPATIBILITY BETWEEN DIFFERENT VERSIONS OF THE SAME TOOLCHAIN

A major product update, for example from version 5 to version 6, of an IAR toolchain is not usually backward compatible. Therefore, you must recompile or reassemble all source code of an application using the new version.

All IAR Systems tools display their version number as part of the sign-on message or, for example, in list files when executed from the command line.

Advice 2.6-1: Before you update an IAR toolchain, make sure that any third-party libraries used are available in a version that is compatible with the new IAR toolchain version.

A minor update of an IAR toolchain, for example from version 6.1 to version 6.2, is usually backward compatible. However, problems found and corrected in supported public Application Binary Interfaces (ABI) can make the toolchain backward incompatible.

Advice 2.6-2: Before updating to a new version, read the release notes for the IAR toolchain to determine whether there are any backward incompatibilities.

Note: Any updates to a certified version of IAR Embedded Workbench are always backward compatible.

2.7 COMPATIBILITY WITH OTHER TOOLCHAINS

As a rule, an IAR toolchain is not fully compatible with other vendors' toolchains for the same microcontroller. Depending on which object format and which application binary interfaces your product uses, the IAR toolchain can be interoperable with toolchains from other vendors. Normally, this means that source code written for the IAR toolchain:

- Can use global variables and functions in the linked-in code from other toolchains
- Might fail to link, or link but fail at runtime, if the code from another toolchain uses C library functionality, or if that code needs compiler-intrinsic support.

Advice 2.7-1: If you are using an IAR toolchain other than IAR Embedded Workbench for Arm, read the *IAR C/C++ Development Guide* for information about compatibility with toolchains from other vendors.

Note: The following ABI information only applies to product packages that include the ILINK Linker.

Advice 2.7-2: If you are using a supported standardized ABI, for example, the Arm AEABI or the RX C ABI, read about ABI compliance in the *IAR C/C++ Development Guide* for information about whether code produced by third-party

tools can be linked. If so, you can also find information about how to link third-party tools, and with what limitations.

Basically, an ABI defines:

- The C library interface
- The compiler intrinsic library—support functions needed for the compiler
- The C++ runtime interface—support for exceptions, RTTI, new/delete, etc, but not for the C++ library interface itself.

The ABI makes it possible to use any third-party code—if compiled in ABI mode—in an application, and it is treated as any other code in the application.

Advice 2.7-3: If you import third-party code, make sure that it is translated with the correct ABI mode enabled, and that the complete application is translated with the ABI mode enabled.

2.8 GENERAL GUIDELINES ON MCU SELF-CHECK STRATEGIES

Safety standards advise or mandate the use of various self-checking strategies for runtime integrity of the hardware. For example, consider the requirements of the IEC 60730 standard for white goods appliances. For some classes of appliances, this standard poses detailed requirements for stack pointer checking, the use of memory checking strategies, and a watchdog timer for various purposes. These tests are focused on detecting hardware errors. However, similar strategies can be used for testing various aspects of the software integrity. Most CPU vendors that aim for this appliance area have created libraries to assist their customers with this testing. This means that even if you are not working under IEC 60730 requirements, you can use or modify selected tests, and take advantage of existing libraries to enhance your software testability.

Advice 2.8-1: Familiarize yourself with the self-check requirements in the IEC 60730 or other application standard, even if you have no specific self-check requirements for your project.

Advice 2.8-2: Review and use vendor libraries to implement the selected tests, making sure that you have a clear understanding of what a selected library does and does not test for. If applicable, discuss with your functional safety assessor how to best use library functionality.

Advice 2.8-3: Follow the advice on stack depth checking in *Stack depth considerations*, page 20.

Advice 2.8-4: Implement and run appropriate tests also for projects that do not explicitly mandate their use.

3 Installation, commissioning, operation, and maintenance

For information about how to install, invoke, use, and maintain IAR Embedded Workbench and its build toolchain, see the *Installation and Licensing Quick Reference*—available in the product box—and the *Licensing Guide*.

For a project with functional safety requirements, consider the following issues:

Advice 3-1: Never install a new toolchain, or an updated version of a toolchain, on top of an existing toolchain, unless there are specific reasons for doing so.

Advice 3-2: Never install IAR Embedded Workbench where it shares the `common` directory with another IAR Embedded Workbench, unless there are specific reasons for doing so—the two different versions would share a common IDE and debugger components. Installing IAR Embedded Workbench so that it shares the `common` directory with another version means that the existing installation could potentially be updated with newer (or older) common components from another product.

However, if two installations of IAR Embedded Workbench *do* share the `common` directory, there are mechanisms to ensure module consistency. By means of runtime model attributes, the linker ensures that modules that are linked into an application are compatible.

Note: The IAR Embedded Workbench installation wizard suggests an installation path. If the new installation is incompatible with a previous installation, the suggested path is not the same as for the previous installation.

Note: If your project involves more than one CPU and the CPUs use different IAR toolchains, it can be useful to let the toolchains share the `common` directory. In that case, consider implementing a versioning policy to avoid accidental updates of common components.

Advice 3-3: If you create automated builds that are not based on the IAR Embedded Workbench *workspace* and *project* files, consider making all references to the build tools absolute and not dependent on, for example, environment-provided path variables or similar.

Advice 3-4: Consider archiving the entire build environment as a snapshot before you update any tools in the toolchain. The archiving can be done in any way, as long as you can recreate and use a previous build environment at a later stage.

Advice 3-5: The typical workflow and build process in IAR Embedded Workbench is similar to corresponding processes in other integrated development environments. However, it is likely that the setup, configuration, customization, and adoption of the toolchain have different degrees of freedom compared to other toolchains. Read and familiarize yourself with the build process described in the *IAR C/C++ Development Guide*.

You should keep up to date with any newly identified issues in the toolchain. If you have a valid support and update agreement for functional safety, IAR Systems will periodically and automatically send you information on issues that are relevant for development under functional safety requirements. You should review and assess this information for potential impact on your source code.

If you identify an error or problematic condition in the toolchain, report this to IAR Systems, who will process the information and inform you whether the issue is a bug or not. In the case of a confirmed bug, you will receive potential workarounds and a tracking ID for the issue that maps to information in release notes and other communication. If the issue has consequences for functional safety development, the issue and ID will be included in periodic information from IAR Systems.

For information on how to submit bug reports, see Technical Support at www.iar.com.

4 Setting up the build environment

This section covers topics related to designing your build environment.

4.1 DEBUG MODE, RELEASE MODE, AND BUILD CONFIGURATIONS

IAR Embedded Workbench supports the concept of *build configurations* where you can easily specify different ways to build and debug an application. One build configuration can differ from another build configuration by option settings, the set of defined preprocessor symbols, or the set of files included in the build. When you create a project, IAR Embedded Workbench automatically creates two build configurations—*Debug* and *Release*. There are several practical reasons for creating different build configurations. For more information about how to create and use build configurations, and the differences between the two default configurations, see the *IDE Project Management and Building Guide*.

Note: For IAR Systems compilers, adding debug information does not impact the performance or size of the executable image. No actual instructions or data storage are introduced when debug information is added to the link image. The only information that is added is meta information used by the debugger to cross-reference instructions, stacks, registers, and labels to the source code.

Advice 4.1-1: As a minimum, consider the following build configuration scenarios as part of your test and validation cycle:

- Build configurations with different optimization goals

Using different optimization goals can help you track errors caused by source code that depends on undefined behavior or implementation-defined behavior

according to the language standard. This strategy can also help in identifying performance issues with the code.

- Clean release build configuration

Early in your project cycle, you should define and create a build configuration for release builds. This build configuration should be set up to be as close as possible to the needs of the final production environment, including optimization levels and, if applicable, production-ready output file formats. This build configuration should be used as the basis for all testing throughout the project. Unless there are specific reasons otherwise, this build configuration output file is what should be delivered to external stakeholders outside the software development team.

The build configuration should never be changed unless production needs dictate a change. If you need a build configuration based on the release configuration with changes for debugging purposes, the changed configuration should be cloned from the release configuration.

Note: If you enable debug support in the compiler, meta information will be added to the object file. The debugger can use this meta information to display high-level information when running the application. This information does not impact how the application behaves at runtime. However, when you set options for the runtime library and the C/C++ standard library, and then choose to enable debug support for certain low-level I/O functionality that enables functionality like file operations on the host computer, etc, such debug support will have an impact on the performance and responsiveness of your application.

Advice 4.1-2: Dynamic runtime analysis tools like C-RUN can help pinpoint potential error sources and direct test efforts. Add one or more build configurations to set up the build environment for the analysis tool. For more information about C-RUN, see *Add-on analysis tools*, page 23.

Note: Analysis tools that operate on a running application, typically instrument the code being tested by adding code to detect interesting situations and keep track of statistics, which impact the performance characteristics of the code. Therefore, if tests are run on instrumented code, you should make sure that the non-instrumented application passes all relevant tests.

4.2 BUILD OPTIONS

Advice 4.2-1: When you set up a new build project and workspace, make sure you only include relevant settings for the particular build configuration. Consider reviewing these examples:

- Preprocessor symbols used for directing conditional compilations should be reviewed to verify, for example, that there are no irrelevant preprocessor defines specified for the release build configuration.

- The setup of your CPU, data, code models, etc—if applicable in the toolchain you are using. The settings should not be too permissive or restrictive.
- For certain devices, you can use different memory byte orders—little and big-endian. Make sure you have selected the correct byte order options for your target architecture, and that the selection also matches any third-party code you use in your project.
- Linker and standard library settings should be reviewed to verify that C standard library I/O is handled correctly.
- Library settings should be reviewed to verify, for example, that the library supports the application’s needs, or that it does not contain superfluous features, etc.
- Settings for the programming language should be reviewed to verify, for example, the sign of the `char` type, the level of ANSI compliance, enabled extensions like intrinsic functions, etc.
- If your project needs different build options for different source files, consider collecting files with similar settings in a dedicated source code group in your IAR Embedded Workbench project, to avoid accidentally changing or leaving settings in place for files where the project-global settings are overridden.
- If you are using the add-on tool C-RUN to analyze the behavior of your code at runtime, you must set certain compiler and linker options. These options are not usually set for final production code. For more information, see *Advice 4.1-2* and *Add-on analysis tools*, page 23.

Advice 4.2-2: Read about compiler and linker options in the *IAR C/C++ Development Guide*. Read about assembler options in the *IAR Assembler Reference Guide*.

4.3 STACK DEPTH CONSIDERATIONS

Advice 4.3-1: Consider these issues:

- Keep stack allocation to a minimum.
- Allocate resources statically in an early stage of your project. In that way, it is possible to already know at link time exactly how much memory these resources will claim. However, for small data items with small scope and transient lifetime, referencing these static objects can potentially consume more registers and increase stack depth. Examples of such objects are loop counters and similar objects used for short-lived or temporary data.
- Pay attention to worst-case length of function call chains and recursion chains, and remember that interrupts increase stack usage.
- Consider implementing your own procedure, possibly tool-assisted, to assess stack depth for your application.

Advice 4.3-2: Consider writing a *magic number* or pattern at startup to each RAM memory location that is not used for stack, dynamic heap memory, or instructions. This pattern can be checked periodically during runtime by a dedicated stack check routine. Depending on the exact requirement specified by the safety standards, this check can either be enabled only for testing purposes, or be part of the final release build. When you manually view the memory section in a debugger memory window, a careful selection of the fill pattern can also assist you in identifying stack overflows and buffer overruns.

4.3.1 Stack usage analysis

Depending on which product you are using, IAR Embedded Workbench incorporates stack usage analysis that can assist you in computing the worst case stack depth for your application code. The advantages of using the stack usage analysis integrated with the compiler and linker toolchain is that the stack analysis operates on the same information for stack usage and layout as the compiler and linker. For more information about stack usage analysis, see the *IAR C/C++ Development Guide*.

Advice 4.3.1-1: Consider using stack usage analysis if it is available in your product.

4.3.2 Tracking stack usage using the debugger

The **Stack** window in the C-SPY debugger can optionally track stack usage. For more information, see the *C-SPY® Debugging Guide*.

When your application is first loaded, and upon each reset, the memory for the stack area is filled with the dedicated byte value `0xCD` before the application starts executing. Whenever execution stops, the stack memory is searched from the end of the stack until a byte with a value different from `0xCD` is found, which is assumed to be how far the stack has been used. Although this is a reasonably reliable way to track stack usage, there is no guarantee that a stack overflow is detected. For example, a stack *can* grow incorrectly outside its bounds, and even modify memory outside the stack area, without actually modifying any of the bytes near the stack range. Likewise, your application might modify memory within the stack area by mistake.

Advice 4.3.2-1: Consider using the stack usage tracking functionality in C-SPY.

Advice 4.3.2-2: If you use a product that supports the C-RUN add-on tool for runtime error checking, consider setting up one or more test configurations that use C-RUN to detect various errors, including conversion errors and out-of-bounds issues.

4.4 LINKER CONFIGURATION

Safety standards generally advise that applications operating under functional safety requirements should not use dynamically allocated objects on the heap or the stack. See for example, *IEC 61508-3, table B.1*.

Note: The standard makes a distinction between *objects* and *variables*.

Advice 4.4-1: Familiarize yourself with the configuration possibilities of the linker.

Advice 4.4-2: If you must use dynamically allocated objects on the heap or the stack, consider this:

- Familiarize yourself with heap setup for projects that do not use dynamic memory allocation so that heap memory is not accidentally allocated—thus impacting on available memory space.
- Avoid placing a large amount of data on the stack. Plan for this during the architecture and implementation phases.

Advice 4.4-3: Make sure the linker generates map files, and that you review the map file generated for your release build configuration regularly.

For information about how to set up the linker configuration file, including information about how to set up and allocate memory for the runtime stack and for the dynamic memory allocation on the heap, see the *IAR C/C++ Development Guide*.

4.4.1 Heap memory allocation

C/C++ constructs that use heap memory are:

- The C heap interface: `malloc`, `calloc`, `realloc`, and `free`.
- Variable length arrays (VLA)—the actual storage for such arrays is allocated on the heap.
- Certain C library functionality, such as:
 - File I/O
 - Locale
 - `printf` and `scanf` for `wchar_t` formatters
- The C++ heap interface: `new` and `delete`
- C++ exceptions
- The C++ standard template library (STL)
- Most of the C++ library.

If the heap is used dynamically, there is a risk that it will be increasingly fragmented. This means that it will become increasingly difficult to allocate a large amount of heap memory. Eventually, the application will run out of heap memory, even though the amount of free bytes on the heap is sufficient.

Advice 4.4.1-1: To reduce the effects of this problem, consider these strategies:

- Use a tool that checks that any used heap memory is freed when it is no longer needed.

- Only use the heap locally, for example, by making a function allocate the heap memory it needs and when finished, make the function free the same amount of memory.
- Allocate the heap memory at the startup of the application and never free the heap.

Advice 4.4.1-2: To make sure that the heap is not used from the application by accident, change the setup in the linker configuration so that no RAM is allocated for the heap. Any use of the heap will then return or throw an `Out of memory` error.

4.5 ADD-ON ANALYSIS TOOLS

Some IAR Embedded Workbench products have add-on tools for static and dynamic/runtime error checking.

- *C-STAT*: a tool for static analysis of C/C++ source code, supports MISRA C:2012, MISRA C:2004, and MISRA C++:2008, as well as numerous coding rules from CWE and CERT.
- *C-RUN*: a tool for runtime error checking, can detect various runtime errors, like conversion errors, shift errors, pointer out-of-bounds issues etc.

Note: C-RUN is currently only available for IAR Embedded Workbench for Arm and RX.

For more information about C-STAT and C-RUN, see the *C-STAT® Static Analysis Guide*, and the *C-SPY® Debugging Guide for Arm*, or *C-SPY® Debugging Guide for RX* respectively.

Advice 4.5-1: If you decide to use one or both of these tools, consider the following:

- C-RUN and C-STAT are add-on tools and are not certified against any functional safety standard. This does not imply that the tools cannot be used in projects with safety requirements, on the contrary, they can help you find programming errors and issues at an early stage. Discuss with your assessor how to incorporate C-RUN and C-STAT into your development cycle.
- C-RUN and C-STAT are fully integrated into the IAR Embedded Workbench IDE and can be used in the daily development workflow. This means that the tools accept the same language dialects as the build chain, and therefore configuration of language compliance, and configuration for adaptation to the build environment, are not needed.
- Messages reported by the analysis tools should be reviewed to determine whether they make sense. Note that errors reported by C-RUN are reported from running code, which means that the code under test is likely to contain code constructs that break one or more language rules.

5 Implementation and coding considerations

This section covers topics related to the implementation and coding of your application.

5.1 OPTIMIZATION MODES

5.1.1 Optimization modes overview

The compiler performs many transformations on your application to generate the best possible code. Examples of such transformations are storing values in registers instead of in memory, removing superfluous code, reordering computations in a more efficient order, and replacing arithmetic operations with more efficient operations.

Note: The more optimization that is applied to the source code, the less the resulting machine code's structure will resemble the structure of the original source code. This means that at high optimization levels, it is difficult to map the produced machine code to the corresponding source code statements, and it is difficult to preserve the values of variables during debugging.

Advice 5.1.1-1: Sometimes it is not possible to use any compiler optimizations at all, either for code or for parts of the code, because of detailed requirements on the traceability of functional safety requirements down to object code. Discuss the feasibility of using optimizations with your functional safety assessor and other project stake holders.

5.1.2 Facilitating compiler transformations

You should be aware that some source code constructs can inhibit compiler optimizations. Such constructs can often also be overly complex, difficult to understand and maintain, and seldom give any clear benefits in terms of code size or execution speed.

Advice 5.1.2-1: Review the texts about compiler transformations in the *IAR C/C++ Development Guide*.

Advice 5.1.2-2: Group together functions that have timing constraints into separate modules, and use speed optimizations. Use size optimizations on the rest of the application.

5.2 INTEGRAL TYPE SELECTION

In some situations, the rules for integer types and their conversion can lead to confusing behavior. Several operators in Standard C automatically convert operands from one type to another. This includes the rules about integer promotion, implicit conversion, and common arithmetic conversions. For more information, see chapter 6.3 *Conversions* in the C99 standard.

Advice 5.2-1: Look out for assignments or conditionals (test expressions) involving types with different size, and logical operations, especially bit negation.

Advice 5.2-2: Use a well-defined language subset, like the MISRA C:2012 or MISRA C:2004 rules, to avoid such constructs.

For information about integer types, arithmetic, and how to avoid common mistakes, see the *IAR C/C++ Development Guide*.

5.3 FLOATING-POINT ARITHMETIC

The build toolchain complies with the IEEE 754 standard for representing floating-point numbers. Support for subnormal numbers and exception flags varies with the CPU and the availability of an FPU unit.

Finite representation of arbitrary decimal numbers will generally lead to rounding and truncation errors.

Advice 5.3-1: If floating-point calculations have any relevance for the safety functions of your application, consider analyzing the validity and precision of the calculations.

Note: Some safety standards have additional rules and guidelines related to floating-point arithmetic. See *IEC 61508-7, section C.2.13*.

For more information about floating-point representation, see the *IAR C/C++ Development Guide*.

5.4 FUNCTIONS

Standard C supports two different styles for function declarations—the Kernighan & Ritchie style and the prototyped style.

Note: The K&R style is only supported for compatibility reasons.

Advice 5.4-1: Always use the prototyped style when declaring and defining functions. The compiler does not check parameter types for correctness when a function declared using K&R style is called.

Advice 5.4-2: Always declare public functions in a header file that is visible from all source files that define or use the function. This ensures that the declaration is always synchronized with the definition, and that all callers call the function with type-correct parameters.

Advice 5.4-3: Use the **Require prototypes** compiler option (`--require_prototypes`) to make the compiler check that no K&R style function declarations/definitions are used, and that a prototype declaration is present before the definition of each public function.

5.5 GLOBAL SYMBOLS

The safest way to use global symbols, such as functions and variables, is to make sure that the symbol is known to the compiler before the symbol is used. You achieve this by declaring the symbol in a header file, and including that header file in any source file that uses that symbol.

Advice 5.5-1: Make sure there is only one declaration of a symbol in the source code, and the declaration is placed in a header file. Any module that uses the symbol must include that header file.

Advice 5.5-2: Make sure all symbols have distinct names.

5.6 CONST AND VOLATILE

The semantics of the `volatile` type qualifier—as specified in Standard C—prohibits the compiler from removing accesses to a `volatile` object, and reordering accesses to `volatile` objects. A `volatile` object is always placed in read-write memory, even if the object is `const` declared.

Advice 5.6-1: Make sure you understand all the implications of the `volatile` keyword:

- Read about `volatile` in the *IAR C/C++ Development Guide* for information about how the specific IAR toolchain you are using treats `volatile` objects.
- Declaring an object `volatile` informs the compiler that it must not optimize away accesses and order of accesses to the object.
- Declaring an object `volatile` is *not* a guarantee that the object will be thread-safe.
- Declaring an object `volatile` is *not* a guarantee that accesses to the object will be atomic. It can be true for certain operations of certain object types and sizes, but if atomicity is a required property for object access, it must be ensured by other means.
- Declaring an object `volatile` without guarding object accesses from different execution contexts with some locking mechanism, might make your timing-dependent source code work in one specific compiler version using specific optimization settings. Changing the compiler version, the optimization settings, or making minor changes to the source code, might lead to erroneous behavior, unless simultaneous access to the object from different execution contexts is properly guarded against.

For more information about using `volatile`, see the *IAR C/C++ Development Guide*.

The `const` type qualifier only means that write accesses to the object are forbidden. Therefore, non-constant objects and constant objects will be placed in memory that is addressed in exactly the same way.

Advice 5.6-2: For information about how the specific IAR toolchain you are using treats `const` objects, read about `const` in the *IAR C/C++ Development Guide*.

Note: In some IAR toolchains, `const` objects are placed in RAM. In this case, the toolchain provides, for example, a specific memory attribute that can be used for placing such objects in ROM.

5.7 POINTERS

Data pointers to objects with different type qualifiers—`const` and `volatile`—point to the same memory and use the same access method. If the CPU supports different areas of data memory, the IAR toolchain will probably support one data pointer type for each different data memory. One of the data pointer types will be used by default.

Advice 5.7-1: If the compiler you are using supports several data pointer types, use the data model that is most suitable for your application. Read about data models in the *IAR C/C++ Development Guide*.

Generally, function pointers are not compatible with data pointers, and saving function pointers in `void *` types might not work. If the CPU supports several areas of code memory, the IAR toolchain will probably support one function pointer type for each code memory area. One of the function pointer types will be used by default.

Advice 5.7-2: If the compiler you are using supports several function pointers, use the code model that is most suitable for your application. Read about code models in the *IAR C/C++ Development Guide*.

6 The C/C++ standard libraries

This section discusses the use of the C/C++ standard library in projects with functional safety requirements. The C/C++ standard libraries contain functionality that is not part of the core language, but which is still standardized. This includes, for example, I/O functionality, string manipulation, and mathematical functions in the C standard library, and the parameterized algorithmic functions in the C++ standard library. The following discussion is restricted to the C standard library, but the same general principles also apply for the C++ standard library. For a brief discussion of the C standard library in a safety context, see *Language standards compliance*, page 7.

Advice 6.0-1: See the *IAR C/C++ Development Guide* for detailed information on the C/C++ standard library in IAR Embedded Workbench. For information about the standard library functions, see the online help system.

6.1 THE C STANDARD LIBRARY

In a functional safety context, the C standard library is most often considered as pre-existing software, see for example, *IEC 61508-3, section 7.4* and *IEC 61508-7*,

section C.2.10. This means that you must take special measures when using functionality provided by the library. Generally, you must evaluate all of the available information on design, verification, testing, and usage of the pre-existing software element. As a result of the evaluation, measures can be devised to handle specific issues. In the following text, a number of issues are listed that can help you make informed decisions about library usage.

6.1.1 Design

- The C standard library provided with IAR Embedded Workbench is licensed from Dinkumware—the world-leading supplier of standard-conformant C/C++ standard libraries. Their libraries are used by many tool vendors, including other compiler and build chain vendors, and organizations that prefer to use a well-known library implementation. Dinkumware actively participates in the standardization work for the C/C++ standard libraries, and also actively works together with test suite providers and parser companies to make sure that their interpretation of the standard is correct and in harmony with the rest of the industry.
- The C standard library is designed to be fully compliant with the ISO/IEC 9899:1999 standard, also known as C99 or Standard C. The library is then customized by IAR Systems to be as efficient as possible on different target hardware platforms. This also includes, for example, the possibility to select downscaled support for I/O.
- The library is delivered as a set of pre-built object libraries, with each specific library built to match, for example, a chosen support level for I/O, a specific calling convention, a specific instruction subset, etc., and combinations of such features and choices. For example, for Arm-based microcontrollers this might include the possibility to call the library from both a 16-bit instruction set and a 32-bit instruction set.
- For a majority of IAR Embedded Workbench product flavors, the full library source code is included in the product package, which makes code review of the full library or selected parts possible.
- The book *The Standard C Library* (Prentice Hall, 1992) by P. J. Plauger, the founder of Dinkumware, describes the design of the library.

Advice 6.1.1-1: Assess the need for, and potential gain from, using library functionality. Consider incorporating a restricted set of allowed functions into the coding standard, and strictly forbidding the use of other functions.

Advice 6.1.1-2: Assess the available design information and identify possible gaps in the information with respect to the information required in your project. Note that implementation complexity varies greatly between different functions in the library. Perform code reviews on the parts of the library that you would benefit from using and cross-reference to the standard for the C standard library.

Advice 6.1.1-3: Note that the library is C99-compliant, even in cases where the compiler is instructed to be C89-compliant. This means that some functions will have a slightly different or extended functionality compared to the C89 standard.

Advice 6.1.1-4: Some low-level library functions like `memcpy` might, under certain circumstances, be treated as compiler intrinsic functions, therefore making them part of the compiler. See *IAR language extensions*, page 9 for a discussion of language extension usage. See the *IAR C/C++ Development Guide* for more information on when and how that can happen.

6.1.2 Verification and test

Dinkumware thoroughly tests the library before delivery to their customers, using a combination of in-house conformance tests and stress tests, as well as third-party test suites from Perennial and Plum Hall. The test suites are cross-referenced to the standard to make sure that all functionality is verified and validated. IAR Systems then replicates the tests on the modified library, using internal Dinkumware tests and additional stress and regression tests, and tests the library in real-world applications. See the *Dinkumware statement* (from the product release notes) for a statement from Dinkumware on design, test, and verification.

Advice 6.1.2-1: Assess and review publicly available information from Dinkumware, Perennial, and Plum Hall on their test suites.

6.1.3 Usage

Advice 6.1.3-1: If your assessment of the library indicates that you can use certain parts of the library, consider the following items:

- *Defensive programming:* In a library context this includes techniques for sanity and invariant checking of data that passes the API boundary between the application and the library.
- *Unit tests:* By implementing project-specific unit tests of selected library functionality, project-specific confidence can be increased. If full library source code is available, code reviews can be used to design in-depth tests that incorporate knowledge of implementation choices.
- *Overriding existing functionality:* It might in certain circumstances be beneficial to replace a certain part of the library with a targeted implementation, for example, to restrict the input domain of some specific function or similar. Note that this requires a thorough understanding of the inner workings of the library.
- *Re-building the full library:* It is possible to configure and rebuild the library to better suit the intended usage. Note that this requires a thorough understanding of the inner workings of the library, and is only recommended for handling special requirements.

Advice 6.1.3-2: Make sure that you understand the implications of library thread safety, and library usage and configuration. For more information, read about managing a multithreaded environment in the *IAR C/C++ Development Guide*.