



Quantum[®]Leaps
innovating embedded systems

MISRA

Application Note

QP/C++[™] MISRA-C++:2008

Compliance Matrix

Document Revision F
December 2014
(Covers QP/C++ 5.4.0)

MISRA and MISRA C are registered trademarks of MIRA Ltd, held on behalf of the MISRA Consortium.



Table of Contents

1 Introduction	1
1.1 About MISRA-C++:2008	1
1.2 About QP™	1
2 Checking MISRA-C++ Compliance with PC-Lint/FlexeLint	2
2.1 Structure of PC-Lint Options for QP/C++	2
2.2 The QP:: Namespace	7
2.3 QS Software Tracing and the Spy (Q_SPY) Configuration	7
2.4 Checking MISRA Compliance of a QP/C++ Source Code	7
2.5 Checking MISRA Compliance of a QP/C++ Application Code	8
3 MISRA-C++:2008 Compliance Matrix	9
3.1 Unnecessary constructs	10
3.2 Storage	10
3.3 Runtime failures	11
3.4 Arithmetic	11
3.5 Language	11
3.6 Character sets	12
3.7 Trigraph sequence	12
3.8 Alternative tokens	12
3.9 Comments	12
3.10 Identifiers	13
3.11 Literals	13
3.12 Declarations and Definitions	13
3.13 One Definition Rule	14
3.14 Declarative regions and scope	14
3.15 Name lookup	14
3.16 Types	14
3.17 Integer promotions	15
3.18 Pointer conversions	15
3.19 Expressions	15
3.20 Postfix expressions	17
3.21 Unary expressions	18
3.22 Shift operators	18
3.23 Logical AND operator	18
3.24 Assignment operators	18
3.25 Comma operator	18
3.26 Constant expressions	19
3.27 Expression statement	19
3.28 Compound statement	19
3.29 Selection statements	20
3.30 Iteration statements	20
3.31 Jump statements	21
3.32 Specifiers	21
3.33 Enumeration declarations	21
3.34 Namespaces	22
3.35 The asm declaration	22
3.36 Linkage specifications	22
3.37 Declarations — General	23
3.38 Meaning of declarations	23
3.39 Function definitions	23
3.40 Declarators — Initializers	23
3.41 Member functions	24
3.42 Unions	24
3.43 Bit-fields(1)	24
3.44 Multiple base classes(1)	24
3.45 Member name lookup	25
3.46 Virtual Functions(1)	25
3.47 Constructors	25

3.48 Copying class objects.....	25
3.49 Template declarations(1).....	26
3.50 Name resolution(1).....	26
3.51 Template instantiation and specialization(1).....	26
3.52 Function template specialization(1).....	26
3.53 Exception handling — General(1).....	27
3.54 Throwing an exception.....	27
3.55 Handling an exception.....	27
3.56 Exception specification.....	28
3.57 Exception handling — Special functions.....	28
3.58 Processing directives — General.....	28
3.59 Conditional inclusion.....	29
3.60 Source file inclusion.....	29
3.61 Macro replacement.....	29
3.62 Pragma directive.....	29
3.63 Library introduction — General.....	30
3.64 Language support library — General(1).....	30
3.65 Language support library — Implementation properties.....	30
3.66 Language support library — Dynamic memory management.....	30
3.67 Language support library — Other runtime support.....	31
3.68 Diagnostics library — Error numbers.....	31
3.69 Input/Output library — General.....	31
4 Beyond MISRA: Compliance with Additional Rules and Standards.....	32
4.1 Strong Type Checking.....	32
4.2 Quantum Leaps C/C++ Coding Standard.....	32
5 Deviation Procedures for QP/C++ Source Code.....	33
5.1 Rule 0-1-1(req) and 0-1-2(req).....	33
5.2 Rule 0-1-11(req).....	33
5.3 Rule 3-9-3(req).....	33
5.4 Rule 5-0-15(req).....	33
5.5 Rule 5-0-18(req).....	33
5.6 Rule 5-2-4(req).....	34
5.7 Rule 5-2-5(req).....	34
5.8 Rule 5-2-7(req).....	34
5.9 Rule 5-2-9(adv).....	34
5.10 Rule 5-2-10(adv).....	34
5.11 Rule 7-1-1(req).....	35
5.12 Rule 8-5-1(req).....	35
5.13 Rule 9-5-1(req).....	35
5.14 Rule 16-0-4(req), 16-2-1(req), and 16-2-2(req).....	35
5.15 Rule 16-2-1(req) and 16-2-2(req).....	35
5.16 Rule 16-3-2(adv).....	36
6 Deviation Procedures for Application-Level Code.....	37
6.1 Rule 5-2-6(req).....	37
6.2 Rule 5-2-7(req).....	37
6.3 Rule 5-2-8(req).....	38
6.4 Rule 6-4-3(req), 6-4-5(req), and 6-4-6(req).....	38
7 Summary	40
8 Related Documents and References.....	41
9 Contact Information.....	42

1 Introduction

This Application Note describes the compliance of the QP/C++™ state machine framework version **5.1.0** or higher and the application code based on this framework with the Motor Industry Software Reliability Association (MISRA) Guidelines for the use of the C++ Language in Critical Systems [MISRA-C++:2008]. This Application Note is designed to be applied to production code in safety-related embedded systems.

1.1 About MISRA-C++:2008

MISRA, the Motor Industry Software Reliability Association (www.misra.org.uk), is a collaboration between vehicle manufacturers, component suppliers, and engineering consultancies, which seeks to promote best practices in developing **safety-related electronic systems** in road vehicles and other embedded systems.

Since its original publication in 1998 [MISRA-C:1998], the original MISRA-C guidelines have gained an unprecedented level of acceptance and use not only in the automotive industry, but in all kinds of embedded systems around the world. Following this initial success, in 2008 MISRA published a set of rules for using C++ in safety-critical embedded systems, known as the MISRA-C++:2008.

Due to the numerous idiosyncrasies, pitfalls, and undefined behavior of the standard C++ language, most experts agree that the full, unconstrained language should **not** be used for programming safety-critical systems. Consequently, the main objective of the MISRA-C++ guidelines was to define and promote a **safer subset** of the C++ language suitable for safety-related embedded systems. The [MISRA-C++:2008] guidelines define this language subset by means of 229 rules that restrict the use of the known problematic aspects of the language. For each of the rules the MISRA-C++ guidelines provide justification and examples.

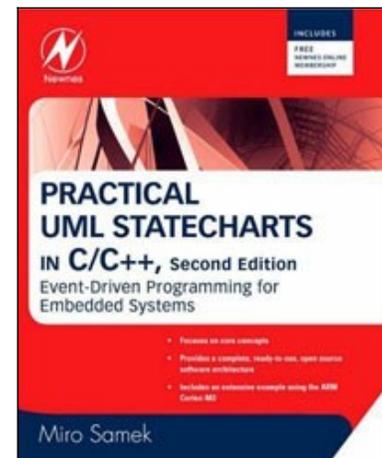


NOTE: MISRA and MISRA C are registered trademarks of [MIRA Ltd.](http://www.misra.org.uk), held on behalf of the [MISRA Consortium](http://www.misra.org.uk).

1.2 About QP™

QP/C++™ is a lightweight, open source, active object framework that enables software developers to build well-structured embedded applications as systems of concurrently executing active objects, each embedding a hierarchical state machine (UML statechart). QP has been described in great detail in the book *Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems* [PSiCC2 08] (Newnes, 2008).

The use of a tested framework, such as QP/C++, addresses the growing concern over the robustness of the **design**, not just the coding aspects of **safety-critical software**. Please refer to state-machine.com/qp for more information..



2 Checking MISRA-C++ Compliance with PC-Lint/FlexeLint

The [MISRA-C++:2008] guidelines place great emphasis on the use of **static code analysts tools** to check compliance with the MISRA-C++ language subset. In fact, the automatic enforcement of as many rules as possible is mandated by MISRA-C++:2008 required rule 0-3-1.

NOTE: The completely automatic enforcement of 100% of the MISRA-C++ rules is not possible and was never intended. Some of the rules are only enforceable by manual review of the code or checking the compiler/linker tools by other means.

This Application Note uses **PC-Lint/FlexeLint** version **9.00k** from Gimpel Software [www.Gimpel.com], which is a static analysis tool for C and C++ with one of the longest track records and best value of the money in the industry. PC-Lint has been supporting checks for the MISRA-C guidelines since early 2001, and MISRA-C++ since 2009. The company is committed to provide ongoing and increasing support for these guidelines (see [PC-Lint 08]).

The primary way of activating MISRA checking for MISRA-C++:2008 guidelines in PC/Lint is via the option file [PC-Lint/MISRA-C++:2008]:

`au-misra-cpp.lnt`

This file contains the options to activate and annotate PC-Lint messages dealing with MISRA-C++:2008. PC-Lint can report deviations from several MISRA C++ rules with messages 1960 and 1963. Additional rules, are covered in other messages, the details of which you can find listed in the `au-misra-cpp.lnt` file.

NOTE: The `au-misra-cpp.lnt` configuration file is also the best overall **documentation** on MISRA-C++:2008 coverage, including not just which rules are covered, but also **how** they are checked and what messages are produced.

2.1 Structure of PC-Lint Options for QP/C++

PC-Lint has several places where it reads its currently valid options:

- From special Lint option files (usually called `*.lnt`)
- From the command line
- From within the special lint-comments in the source code modules (**not** recommended)

The QP/C++ source code and example application code has been “linted” only by means of the first alternative (option files) with possibility of adding options via command line. The third alternative—lint comments—is **not** used and Quantum Leaps does not recommend this alternative.

NOTE: The QP/C++ source code is completely **free** of lint comments, which are viewed as a contamination of the source code.

The structure of the PC-Lint option files used for “linting” QP/C++ follows exactly the Gimpel Software guidelines for configuring PC-Lint (See Section 3.2 “Configuration” in the PC-Lint/FlexeLint Manual). The design and grouping of the lint options also reflects the fact that static code analysis of a software **framework**, such as QP/C++, has really two major aspects. First, the source code of the framework itself has to be analyzed. But even more important and helpful to the users of the framework is providing the infrastructure to effectively analyze the **application-level** code based on the framework. With this in mind, [Listing 1](#) shows that the PC-Lint options for static analysis of QP/C++ are divided into two groups, located

in directories `qpcpp\include\` and `qpcpp\ports\lint\`. These two groups are for analyzing QP/C++ applications and QP/C++ source code, respectively.

Listing 1: PC-Lint options for “linting” QP/C++ applications (`qpcpp\include\`) and “lining” QP/C++ source code itself (`qpcpp\ports\lint\`).

```
%QPCPP%\           - QP/C++ Root Directory (environment variable QPCPP)
|
+-include\         - QP/C++ platform-independent includes
| +-au-ds.lnt      - Dan Saks recommendations
| +-au-misra-cpp.lnt - Main PC-Lint MISRA-C++:2008 compliance options
| +-lib-qpcpp.lnt  - PC-Lint options for QP/C++ applications
| +-std.lnt        - Standard PC-Lint settings recommended by Quantum Leaps
| +-qassert.h      - QP/C++ header file
| +-qep.h          - QP/C++ header file
| +-. . .         - . . .
| |
| +-ports\         - QP/C++ ports directory
| | +-lint\        - QP/C++ “port” to PC-Lint
| | | +-lin.bat    - Batch file to invoke PC-Lint to run analysis of QP/C++ code
| | | +-options.lnt - PC/Lint options for “linting” QP/C++ source code
| | | +-lint_qep.txt - PC/Lint output for the QEP component of QP/C++
| | | +-lint_qf.txt - PC/Lint output for the QF component of QP/C++
| | | +-lint_qk.txt - PC/Lint output for the QK component of QP/C++
| | | +-lint_qs.txt - PC/Lint output for the QS component of QP/C++
| | | +-qep_port.h - QEP component “port” to a generic ANSI C++ compiler
| | | +-qf_port.h  - QF component “port” to a generic ANSI C++ compiler
| | | +-qk_port.h  - QK component “port” to a generic ANSI C++ compiler
| | | +-qs_port.h  - QS component “port” to a generic ANSI C++ compiler
| | | +-stdint.h   - Standard exact-width integers for an ANSI C++ compiler
| |
| +-examples\     - QP/C++ examples directory
| | +-arm-cortex\ - QP/C++ examples for ARM Cortex
| | | +-qk\        - QP/C++ examples for ARM Cortex with QK kernel
| | | | +-gnu\     - QP/C++ examples with GNU compiler
| | | | | +-dpp-qk-ev-lm3s811-lint\ - Lint-compatible example for GNU
| | | | | +-iar\   - QP/C++ examples with IAR compiler
| | | | | +-dpp-qk-ev-lm3s811-lint\ - Lint-compatible example for IAR
```

NOTE: This Application Note assumes that the baseline distribution of the QP/C++ framework has been downloaded and installed and that the environment variable `QPCPP` has been defined to point to the QP/C++ installation directory.

As shown in Listing 1, the directory `%QPCPP%\include\`, contains the PC-Lint options for “linting” the **application** code along with all platform-independent QP/C++ header files required by the applications. This collocation of lint options with header files simplifies “linting”, because specifying just `-i%QPCPP%\include\include` directory to PC-Lint accomplishes both inclusion of QP/C++ header files and PC-Lint options.

Note that the `%QPCPP%\include\` directory contains **all** PC-Lint option files used in “linting” the code, including the standard MISRA-C++:2008 `au-misra-cpp.lnt` option file as well as Dan Saks' recommendations `au-ds.lnt`, which are copied from the PC-Lint distribution. This design freezes the lint options for which the compliance has been checked.

2.1.1 The `std.lnt` option file

According to the Gimpel Software PC-Lint configuration guidelines, the file `%QPC%\include\std.lnt` file, shown in Listing 2, contains the top-level options, which Quantum Leaps recommends for all projects. These options include the formatting of the PC-Lint messages and making two passes to perform better cross-module analysis. However, the most important option is `-restore_at_end`, which has the effect of surrounding each source file with options `-save` and `-restore`. This precaution prevents options from “bleeding” from one file to another.

Listing 2: Top-level option file `std.lnt`

```
// output: a single line, file info always, use full path names
-hF1
+ffn
-"format=%(\q%f\q %l %C%) %t %n: %m"

-width(0,0)      // do not break lines
+flm            // make sure no foreign includes change the format

-zero(99)       // don't stop make because of warnings
-passes(2)      // make two passes (for better error messages)
-restore_at_end // don't let -e<nn> options bleed to other files
-summary()      // produce a summary of all produced messages
```

2.1.2 The `lib-qpcpp.lnt` option file

The most important file for “linting” QP/C++ applications is the `lib-qpcpp.lnt` option file. This file handles all deviations from the MISRA-C++:2008 rules, which might arise at the **application-level** code from the use of the QP/C++ framework. In other words, the `lib-qpcpp.lnt` option file allows completely clean “linting” of the application-level code, as long as the application code does not deviate from any of the MISRA-C++:2008 rules.

At the same time, the `lib-qpcpp.lnt` option file has been very carefully designed **not** to suppress any MISRA-C++:2008 rule checking outside the very specific context of the QP/C++ API. In other words, the `lib-qpcpp.lnt` option file still supports **100% of the MISRA-C++:2008 rule checks** that PC-Lint is capable of performing.

For example, for reasons explained in Section 5.12, QP/C++ extensively uses function-like macros, which deviates from the MISRA-C++:2008 advisory rule 19.7 and which PC-Lint checks with the warning 961. However, instead of suppressing this warning globally (with the `-e961` directive), the `lib-qpcpp.lnt` option file suppresses warning 961 **only** for the specific QP function-like macros that are visible to the application level. So specifically, the `lib-qpcpp.lnt` file contains directives `-estring(961, Q_TRAN, Q_SPUER, ...)`, which suppresses the warning only for the specified macros, but does not disable checking of any other macros in the application-level code.

Listing 3: file `lib-qpcpp.lnt`

```
// general (event.h)
-estring(1960,      // 11-0-1(req) non-private data member
  sig,
  poolId_,
  refCtr_)
```



```
-estring(1927,          // 8-5-1(req) not in the ctor initializer
  *QEvt::poolId_,
  *QEvt::refCtr_)
-estring(1923,          // 16-2-2(req) could become const variable
  Q_EVT_CTOR,
  Q_SIGNAL_SIZE,
  QP_VERSION)
-emacro(923, Q_UINT2PTR_CAST) // 5-2-7, 5-2-8 cast from int to pointer
-estring(1960,          // 16-0-4(req) function-like macro
  Q_DIM,
  Q_UINT2PTR_CAST)
-esyms(1790,           // Base class has no virtual functions
  *QEvt)
-estring(1712, QEvt) // default constructor not defined
-estring(1401,        // member not initialized in the ctor
  *QEvt::poolId_,
  *QEvt::refCtr_)
-emacro(866, Q_DIM) // Unusual use of 'SYM' in argument to sizeof
-estring(970, bool) // Use of 'bool' outside of a typedef

// Assertions
-estring(1960,        // 16-0-4 function-like macro
  Q_ASSERT,
  . . .
  Q_DEFINE_THIS_MODULE)
. . .
-function(exit, Q_onAssert) // give Q_onAssert() the semantics of "exit"

// QEP
-emacro(929,          // 5-2-7(req) cast pointer to pointer
  Q_STATE_CAST,
  Q_EVENT_CAST)
-emacro(1939,        // 5-2-2, Down cast detected
  Q_EVT_CAST,
  Q_NEW,
  Q_NEW_X)
-estring(1960, *QAttr) // 9-5-1 declaration of union
-esyms(1960,        // 16-0-4 function-like macro
  Q_ROM_BYTE,
  Q_TRAN,
  Q_SUPER,
  Q_STATE_CAST,
  Q_EVENT_CAST)
-esyms(1712,        // default constructor not defined
  QFsm,
  QHsm)

// QF
-emacro(929, Q_NEW) // 5-2-7 cast from pointer to pointer
-esyms(1927,        // 8-5-1 Symbol not in the ctor initializer list
  *QActive::m_prio,
  *QActive::m_eQueue,
  *QActive::m_osObject,
  *QActive::m_thread)
-estring(1923,        // 16-2-2 could become const variable
```



```
    QF_MAX_ACTIVE,
    . . .)
-esyml(1960,           // 16-0-4 function-like macro
    QF_QS_CRIT_EXIT,
    . . .)
-estring(1963, Q_NEW) // 16-3-2(adv) '#/##' used in macro
-esyml(1790,           // Base class has no virtual functions
    *QHsm,
    *QFsm,
    *QActive)
-esyml(1401,           // member not initialized by constructor
    *QActive::m_prio,
    *QActive::m_eQueue,
    *QActive::m_osObject,
    *QActive::m_thread)
-esyml(1712,           // default constructor not defined for class
    *QActive,
    *QTimeEvt)
-esyml(641, *Signals) // Converting enum '*Signals' to 'int'

// QK
-esyml(1960,           // 16-0-4 function-like macro
    QK_ISR_ENTRY,
    . . .)

// QS
-emacro(506, QS_*)     // 0-1-1 constant value boolean
-emacro(774, QS_*)     // 0-1-1 'if' always True
-emacro(930, QS_*)     // 5-2-7 cast from enum to unsigned char
-emacro(923,           // 5-2-7, 5-2-8 cast from pointer to int
    QS_OBJ_,
    QS_FUN_)
-estring(1923,         // 16-2-2 macro could become const variable
    QS_TIME_SIZE,
    . . .)
-esyml(1960,           // 16-0-4 function-like macro
    . . .)
-esyml(1923,           // 16-2-2 could become const variable
    QS_INIT,
    QS_EXIT,
    . . .)
-esyml(1963,           // 16-3-2, '#/##' operator used in macro
    QS_SIG_DICTIONARY,
    . . .)
-emacro(717, QS_*)     // do ... while(0);
-emacro(725, QS_END*) // Expected positive indentation
-esyml(641, *QSType)   // Converting enum '*QSType' to 'int'

// Miscellaneous
-estring(793,6)        // ANSI/ISO limit of 6 significant chars exceeded
-e546                  // Suspicious use of &
```

NOTE: Any changes to the PC-Lint option files (e.g., as part of upgrading PC-Lint) must be done with **caution** and must be always followed by regression analysis of all source code.

2.2 The QP : : Namespace

The MISRA-C++:2008 group of required rules 7-3-x (“Namespaces”) is concerned with avoiding name conflicts by applying C++ namespaces. To comply with these rules, QP/C++ source code must be configured to place all declarations and definitions in the `QP : : namespace`. Starting with QP/C++ version 4.5.02 QP/C++ applies the `QP : : namespace` by defining default.

Please note also that compliance with the rules 7-3-x requires also the **application-level code** to use a specific namespace as opposed to contaminate the global namespace. The application-level code can either extend the `QP : : namespace`, or it can use its own distinct namespace. Either way, all QP elements must be explicitly qualified with the `QP : : namespace` prefix, as the “using namespace QP” directive is not allowed by the required rules 7-3-4, 7-3-5, and 7-3-6.

NOTE: To comply with MISRA-C++:2008, the applications must explicitly qualify all QP/C++ elements with the `QP : : namespace` prefix. The MISRA-compliant application examples (see [Listing 1](#)) show how to use the `QP : : namespace` prefix.

2.3 QS Software Tracing and the Q_SPY Configuration

As described in Chapter 11 of the book “Practical UML Statecharts in C/C++” [PSiCC2], all components of the QP/C++ framework contain software tracing instrumentation (called Quantum Spy, or QS). This instrumentation code is inactive in the Debug and Release build configurations, but becomes active in the **Spy** configuration.

In the context of MISRA-C++ compliance it is important to note that, by the nature of software tracing, the QS code embedded in the QP/C++ framework contributes disproportionately to the total number of deviations from the MISRA-C++ rules, both in the QP/C++ source code and in the application-level code. However, these deviations occur only in the Spy build configuration, which is **not** the code shipped within a product.

NOTE: Many of the deviations from the MISRA-C++:2008 rules reported in the upcoming MISRA Compliance Matrix do **not** pertain to the production code.

2.4 Checking MISRA Compliance of a QP/C++ Source Code

The directory `%QPCPP%\ports\lint\` (see [Listing 1](#)) contains the `lin.bat` batch file for “linting” the QP/C++ source code. The `lin.bat` batch file invokes PC-Lint and directs it to generate the diagnostic to specific output files. As shown in [Listing 1](#), the lint output is collected into four text files `lint_qep.txt`, `lint_qf.txt`, `lint_qk.txt`, and `lint_qs.txt`, for QEP, QF, QK, and QS components of the QP/C++ framework, respectively.

NOTE: In order to execute the `lin.bat` file on your system, you might need to adjust the symbol `PC_LINT_DIR` at the top of the batch file, to the PC-Lint installation directory on **your** computer.

The `lin.bat` batch file invoked without any command-line options checks the code in the default configuration corresponding to Run or Debug build of a project. But the `lin.bat` batch can also be invoked with the option `-dQ_SPY` to check the QP/C++ code in the QS configuration with software tracing.

NOTE: By the nature of software tracing, the `Q_SPY` configuration transgresses many more MISRA-C++:2008 rules than the standard configuration. However, the `Q_SPY` configuration is never used for

production code, so the MISRA-C++ compliance of the QP/C++ framework should **not** be judged by the deviations that happen only in the `Q_SPY` configuration.

According to the PC-Lint guidelines, the `lin.bat` uses two option files: the `std.lnt` option file discussed before and the `options.lnt` option file that covers all deviations from the MISRA-C++ rules in the QP/C++ source code. Section 3 (MISRA compliance matrix) cross-references all these deviations, while Section 5 (deviation procedures) describes the reasons for deviations in those, very specific contexts.

2.5 Checking MISRA Compliance of a QP/C++ Application Code

The QP/C++ baseline code (for versions QP/C++ 4.4.00 and higher) contains two examples of MISRA-C++ compliance checking with PC/Lint:

- The DPP example for the EK-LM3S811 Cortex-M3 board with the IAR ARM compiler, located in the directory `qpcpp\examples\arm-cortex\qk\iar\dpp-qk-ev-lm3s811-lint\`; and
- The DPP example for the EK-LM3S811 Cortex-M3 board with the GNU ARM compiler, located in the directory `qpcpp\examples\arm-cortex\qk\gnu\dpp-qk-ev-lm3s811-lint\`.

The PC-Lint analysis is very simple and requires invoking the `lin.bat` file from the `lint\` subdirectory in each of the application folders.

NOTE: In order to execute the `lin.bat` file on your system, you might need to adjust the symbol `PC_LINT_DIR` at the top of the batch file, to the PC-Lint installation directory on **your** computer. You

The `lint\` subdirectory in each of the application folders contains also the `options.lnt` with the PC-Lint options specific to linting the application. This file specifies the include directory for the specific embedded compiler used to compile the application, and you most likely need to adjust it for your system.

Running PC-Lint on embedded projects (such as the DPP example for ARM Cortex-M) requires option files for the specific compilers (`co-iar-arm.lnt` file for IAR ARM and `co-gnu-arm.lnt` file GNU ARM, respectively). **These option files are provided in the Qtools collection.** The location of the **Qtools** directory in your system is specified in the `options.lnt` file, and you most likely need to adjust it for your system.

NOTE: The **Qtools** collection is available for a separate download from <http://www.state-machine.com/downloads/index.php#QTools>. Quantum Leaps is committed to keep adding more and more PC-Lint option files for various embedded C/C++ cross-compilers in the **Qtools** collection.

3 MISRA-C++:2008 Compliance Matrix

As recommended in Section 4.3.1 of the [MISRA-C++:2008] guidelines, this section presents the compliance matrix, which lists each MISRA-C++:2008 rule and indicates the compliance status and how the rule has been checked. The meaning of the compliance matrix columns is as follows:

1. **Rule No.** column lists the MISRA-C++:2008 rule number followed by the rule classification in parentheses (**req**) for required rule and (**adv**) for advisory rule.
2. **PC-Lint** column lists whether a rule is checked by PC-Lint/au-misra-cpp.lnt. The checked rules are marked with a check-mark (☑). Empty status (☐), also clearly marked by the **yellow background**, means that the rule is **not** checked by PC-Lint and requires a **manual review**.

NOTE: The ability of PC-Lint to check a MISRA-C++:2008 rule is determined by means of two sources (1) the Gimpel Software matrix [PC-Lint-MISRA-C++:2008] and (2) the test against the actual code. When in doubt, the rules are marked as **not-checked** by PC-Lint.

3. **QP/C++** column lists the compliance status of the QP/C++ source code. Letters **A** or **M** in this column mean that the QP/C++ framework source code complies with the rule, whereas A means that the rule has been checked automatically (via PC-Lint), and M means that the rule has been verified manually. A number in this column (clearly marked by the **orange background**) indicates a **deviation** from the rule. The number is the subsection number within the section [Deviation Procedures for QP/C++ Source Code](#), which describes in detail the nature and particular circumstances of the deviation.
4. **QP/C++ app.** column lists the deviations of the QP/C++ **application-level** code imposed by the QP/C++ framework. No entry in this column indicates that QP/C++ imposes or no deviations, meaning that the application-level code can be made compliant with the rule. However, for some rules (clearly marked by the **red background** in this column) the design and/or the implementation of the QP/C++ framework imposes a deviation from the rule, in which case the column lists the subsection number within the section [Deviation Procedures for Application-Level Code](#). Finally, cases that the QP/C++ **might** impose a deviation, but a **workaround** exists, are clearly marked with the **blue background** in this column.
5. **Rule Headline** column contains a short description of the rule, as published in Appendix A of the [MISRA-C++:2008] guidelines.

NOTE: The use of the MISRA-C++:2008 rule headlines in this document has been licensed to Quantum Leaps by [MIRA Ltd.](#)

3.1 Unnecessary constructs

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
0-1-1(req)	☑	5.1 ⁽¹⁾	5.1 ⁽¹⁾	A project shall not contain unreachable code.
0-1-2(req)	☑	5.1 ⁽¹⁾	5.1 ⁽¹⁾	A project shall not contain infeasible paths.
0-1-3(req)	☑	A		A project shall not contain unused variables.
0-1-4(req)	☑	A		A project shall not contain non-volatile POD variables having only one use
0-1-5(req)	☑	A		A project shall not contain unused type declarations
0-1-6(req)	☑	A		A project shall not contain instances of non-volatile variables being given values that are never subsequently used.
0-1-7(req)	☑	A		The value returned by a function having a non-void return type that is not an overloaded operator shall always be used
0-1-8(req)	☑	A		All functions with void return type shall have external side-effects
0-1-9(req)	☑	A		There shall be no dead code
0-1-10(req)	☑	A		Every defined function shall be called at least once
0-1-11(req)	☑	5.2		There shall be no unused parameters (named or unnamed) in non-virtual functions.
0-1-12(req)	☑	A		There shall be no unused parameters (named or unnamed) in a virtual function and all the functions that override it.

⁽¹⁾ QP/C++ deviates from this rule only in the QS macros in the Q_SPY configuration

3.2 Storage

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
0-2-1(req)	☑	A		An object shall not be assigned to an overlapping object

3.3 Runtime failures

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
0-3-1(doc)	<input checked="" type="checkbox"/>	A ⁽¹⁾	(1)	Minimization of run-time failures shall be ensured by the use of at least one of: (a) static analysis tools/techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults.
0-3-2(req)	<input checked="" type="checkbox"/>	A ⁽²⁾	(2)	If a function generates error information, then that error information shall be tested.

⁽¹⁾ The QP/C++ framework uses static analysis tool (PC-Lint) and explicit coding of checks (assertions). For compliance with this rule, application-level code needs to use static analysis tool as well.

⁽²⁾ The QP/C++ framework uses assertions except of error returns.

3.4 Arithmetic

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
0-4-1(doc)	<input type="checkbox"/>	M ⁽¹⁾		Use of scaled-integer of fixed-point arithmetic shall be documented.
0-4-2(doc)	<input type="checkbox"/>	M ⁽¹⁾		Use of floating-point arithmetic shall be documented.
0-4-3(doc)	<input type="checkbox"/>	M ⁽¹⁾		Floating-point implementations shall comply with a defined floating-point standard.

⁽¹⁾ QP/C++ does not use scaled-integers or floating-point arithmetic

3.5 Language

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
1-0-1(req)	<input checked="" type="checkbox"/>	A		All code shall conform to ISO/IEC 14882:2003 "The C++ Standard Incorporating Technical Corrigendum 1".
1-0-2(doc)	<input type="checkbox"/>	M ⁽¹⁾		Multiple compilers shall only be used if they have a common, defined interface.
1-0-3(doc)	<input type="checkbox"/>	M ⁽²⁾		The implementation of integer division in the chosen compiler shall be determined and documented.

⁽¹⁾ Selected QP/C++ ports might require assembly modules. In all such cases a documented interface standard, such as APCS (ARM Procedure Calling Standard) are observed.

⁽²⁾ QP/C++ does not use integer division or modulo operation anywhere in the code. (QP/C++ does not use even integer multiplication.)

3.6 Character sets

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
2-2-1(doc)	<input type="checkbox"/>	M ⁽¹⁾		The character set and the corresponding encoding shall be documented.

⁽¹⁾ QP/C++ source code uses only ASCII character set

3.7 Trigraph sequence

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
2-3-1(req)	<input checked="" type="checkbox"/>	A		Trigraphs shall not be used.

3.8 Alternative tokens

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
2-5-1(adv)	<input checked="" type="checkbox"/>	A		Digraphs should not be used.

3.9 Comments

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
2-7-1(req)	<input checked="" type="checkbox"/>	A ⁽¹⁾		The character sequence /* shall not be used within a C-style comment
2-7-2(req)	<input type="checkbox"/>	M		Sections of code shall not be “commented out” using C-style comments
2-7-3(adv)	<input type="checkbox"/>	M		Sections of code shall not be “commented out” using C++ comments.

⁽¹⁾ QP/C++ source code does not use the C-style comments, except in a few comments for multi-line assertions, where some compilers generated errors when C++ comments were used.

3.10 Identifiers

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
2-10-1(req)	<input type="checkbox"/> ⁽¹⁾	A		Different identifiers shall be typographically unambiguous.
2-10-2(req)	<input checked="" type="checkbox"/>	A		Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.
2-10-3(req)	<input type="checkbox"/>	M		A <code>typedef</code> name (including qualification, if any) shall be a unique identifier.
2-10-4(req)	<input type="checkbox"/>	M		A class, union, or <code>enum</code> name (including qualification, if any) shall be unique identifier.
2-10-5(adv)	<input type="checkbox"/>	M		The identifier name of a non-member object or function with static storage duration should not be reused.
2-10-6(req)	<input checked="" type="checkbox"/>	A		If an identifier refers to a type, it shall not also refer to an object or a function in the same scope.

⁽¹⁾ The PC-Lint matrix indicates “partial” checking of this rule

3.11 Literals

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
2-13-1(req)	<input checked="" type="checkbox"/>	A		Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.
2-13-2(req)	<input checked="" type="checkbox"/>	A		Octal constants (other than zero) and octal escape sequences (other than <code>'\0'</code>) shall not be used.
2-13-3(req)	<input checked="" type="checkbox"/>	A		A “ <code>U</code> ” suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.
2-13-4(req)	<input checked="" type="checkbox"/>	A		Literal suffixes shall be upper case.
2-13-5(req)	<input checked="" type="checkbox"/>	A		Narrow and wide string literals shall not be concatenated.

3.12 Declarations and Definitions

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
3-1-1(req)	<input checked="" type="checkbox"/>	A		It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.
3-1-2(req)	<input checked="" type="checkbox"/>	A		Functions shall not be declared at block scope.
3-1-3(req)	<input checked="" type="checkbox"/>	A		When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.

3.13 One Definition Rule

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
3-2-1(req)	<input checked="" type="checkbox"/>	A		All declarations of an object or function shall have compatible types
3-2-2(req)	<input checked="" type="checkbox"/>	A		The One Definition Rule shall not be violated
3-2-3(req)	<input checked="" type="checkbox"/>	A		A type, object or function that is used in multiple translation units shall be declared in one and only one file.
3-2-4(req)	<input checked="" type="checkbox"/>	A		An identifier with external linkage shall have exactly one definition.

3.14 Declarative regions and scope

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
3-3-1(req)	<input checked="" type="checkbox"/>	A		Objects and functions with external linkage shall be declared in a header file
3-3-2(req)	<input checked="" type="checkbox"/>	A		If a function has internal linkage then all re-declarations shall include the static storage class specifier.

3.15 Name lookup

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
3-4-1(req)	<input type="checkbox"/>	M		An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.

3.16 Types

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
3-9-1(req)	<input type="checkbox"/>	M		The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.
3-9-2(adv)	<input checked="" type="checkbox"/>	A ⁽¹⁾		<code>typedefs</code> that indicate size and signedness should be used in place of the basic numerical types.
3-9-3(req)	<input checked="" type="checkbox"/>	5.3 ⁽²⁾		The underlying bit representations of floating-point values shall not be used.

⁽¹⁾ QP/C++ uses the standard exact-width integer types `stdint.h` (WG14/N843 C99, Section 7.18.1.1)

⁽²⁾ QP/C++ deviates from this rule only in QS software tracing implementation for binary output of floating-point values.

3.17 Integer promotions

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
4-5-1(req)	<input checked="" type="checkbox"/>	A		Expressions with type <code>bool</code> shall not be used as operands to built-in operators other than the assignment operator <code>=</code> , the logical operators <code>&&</code> , <code> </code> , <code>!</code> , the equality operators <code>==</code> and <code>!=</code> , the unary <code>&</code> operator, and the conditional operator.
4-5-2(req)	<input type="checkbox"/>	M		Expressions with type <code>enum</code> shall not be used as operands to built-in operators other than the subscript operator <code>[]</code> , the assignment operator <code>=</code> , the equality operators <code>==</code> and <code>!=</code> , the unary <code>&</code> operator, and the relational operators <code><</code> , <code><=</code> , <code>></code> , <code>>=</code> .
4-5-3(req)	<input checked="" type="checkbox"/>	A		Expressions with type (plain) <code>char</code> and <code>wchar_t</code> shall not be used as operands to built-in operators other than the assignment operator <code>=</code> , the equality operators <code>==</code> and <code>!=</code> , and the unary <code>&</code> operator.

3.18 Pointer conversions

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
4-10-1(req)	<input type="checkbox"/>	M ⁽¹⁾		NULL shall not be used as an integer value.
4-10-2(req)	<input checked="" type="checkbox"/>	A		Literal zero (0) shall not be used as the null-pointer-constant.

⁽¹⁾ QP/C++ source code does not use the NULL literal anywhere in the code

⁽¹⁾ QP/C++ source code uses the strongly-typed, specifically cast pointer literals, e.g., `static_cast<QEvt const *>(0)`;

3.19 Expressions

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
5-0-1(req)	<input checked="" type="checkbox"/>	A		The value of an expression shall be the same under any order of evaluation that the standard permits.
5-0-2(adv)	<input checked="" type="checkbox"/>	A		Limited dependence should be placed on C++ operator precedence rules in expressions.
5-0-3(req)	<input checked="" type="checkbox"/>	A		A <i>cvalue</i> expression shall not be implicitly converted to a different underlying type.
5-0-4(req)	<input checked="" type="checkbox"/>	A		An implicit integral conversion shall not change the signedness of the underlying type.
5-0-5(req)	<input checked="" type="checkbox"/>	A		There shall be no implicit floating-integral conversions.
5-0-6(req)	<input checked="" type="checkbox"/>	A		An implicit integral or floating-point conversion shall not reduce the size of the underlying type.
5-0-7(req)	<input checked="" type="checkbox"/>	A		There shall be no explicit floating-integral conversions of a <i>cvalue</i> expression.
5-0-8(req)	<input checked="" type="checkbox"/>	A		An explicit integral or floating-point conversion shall not increase the

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
				size of the underlying type of a <i>cvalue</i> expression.
5-0-9(req)	<input checked="" type="checkbox"/>	A		An explicit integral conversion shall not change the signedness of the underlying type of a <i>cvalue</i> expression.
5-0-10(req)	<input checked="" type="checkbox"/>	A		If the bitwise operators <code>~</code> and <code><<</code> are applied to an operand with an underlying type of <code>unsigned char</code> or <code>unsigned short</code> , the result shall be immediately cast to the underlying type of the operand.
5-0-11(req)	<input checked="" type="checkbox"/>	A		The plain <code>char</code> type shall only be used for the storage and use of character values.
5-0-12(req)	<input checked="" type="checkbox"/>	A ⁽¹⁾		<code>signed char</code> and <code>unsigned char</code> type shall only be used for the storage and use of numeric values.
5-0-13(req)	<input checked="" type="checkbox"/>	A		The condition of an <code>if</code> -statement and the condition of an iteration-statement shall have type <code>bool</code> .
5-0-14(req)	<input checked="" type="checkbox"/>	A		The first operand of a conditional-operator shall have type <code>bool</code> .
5-0-15(req)	<input checked="" type="checkbox"/>	5.4		Array indexing shall be the only form of pointer arithmetic.
5-0-16(req)	<input checked="" type="checkbox"/>	A		A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.
5-0-17(req)	<input checked="" type="checkbox"/>	A		Subtraction between pointers shall only be applied to pointers that address elements of the same array.
5-0-18(req)	<input checked="" type="checkbox"/>	5.5		<code>></code> , <code>>=</code> , <code><</code> , <code><=</code> shall not be applied to objects of pointer type, except where they point to the same array.
5-0-19(req)	<input checked="" type="checkbox"/>	A		The declaration of objects shall contain no more than two levels of pointer indirection.
5-0-20(req)	<input type="checkbox"/>	M		Non-constant operands to a binary bitwise operator shall have the same underlying type.
5-0-21(req)	<input checked="" type="checkbox"/>	A		Bitwise operators shall only be applied to operands of unsigned underlying type.

⁽¹⁾ QP/C++ source code uses special type `char_t` for all (single-byte) strings and string constants.

3.20 Postfix expressions

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
5-2-1(req)	<input checked="" type="checkbox"/>	A		Each operand of a logical && or shall be a postfix-expression.
5-2-2(req)	<input checked="" type="checkbox"/> ⁽¹⁾	A	⁽¹⁾	A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of <code>dynamic_cast</code> .
5-2-3(adv)	<input type="checkbox"/>	M		Casts from a base class to a derived class should not be performed on polymorphic types.
5-2-4(req)	<input checked="" type="checkbox"/>	5.6 ⁽²⁾		C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used.
5-2-5(req)	<input checked="" type="checkbox"/>	5.7		A cast shall not remove any <code>const</code> or <code>volatile</code> qualification from the type of a pointer or reference.
5-2-6(req)	<input checked="" type="checkbox"/>	A	6.1	A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.
5-2-7(req)	<input checked="" type="checkbox"/>	5.8	6.2	An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.
5-2-8(adv)	<input checked="" type="checkbox"/>	A	5.6 ⁽²⁾	An object with integer type or pointer to <code>void</code> type shall not be converted to an object with pointer type.
5-2-9(adv)	<input checked="" type="checkbox"/>	5.9		A cast should not convert a pointer type to an integral type.
5-2-10(adv)	<input checked="" type="checkbox"/>	5.10		The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.
5-2-11(req)	<input checked="" type="checkbox"/>	A		The comma operator, && operator and the operator shall not be overloaded.
5-2-12(req)	<input checked="" type="checkbox"/>	A		An identifier with array type passed as a function argument shall not decay to a pointer.

⁽¹⁾ PC-Lint reports any down-cast as violation of the rule 5-2-2. The QP/C++ source code provides macro `Q_EVENT_CAST()` for performing downcasting of events from the `QEvt` base class to the specific user-defined derived class.

⁽²⁾ QP/C++ source code uses predominantly C++ style casts. The C-style casts are used only in code that is likely to be used in C modules, such as QP/C++ assertions (`qassert.h` header file).

⁽³⁾ QP/C++ provides macro `Q_UNIT2PTR_CAST()`, which could be used in the QP/C++ **applications** to access specific hardware registers.

3.21 Unary expressions

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
5-3-1(req)	<input checked="" type="checkbox"/>	A		Each operand of the ! operator, the logical && or the logical operators shall have type <code>bool</code> .
5-3-2(req)	<input checked="" type="checkbox"/>	A		The unary minus operator shall not be applied to an expression whose underlying type is unsigned.
5-3-3(req)	<input checked="" type="checkbox"/>	A		The unary & operator shall not be overloaded.
5-3-4(req)	<input checked="" type="checkbox"/>	A		Evaluation of the operand to the <code>sizeof</code> operator shall not contain side effects.

3.22 Shift operators

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
5-8-1(req)	<input checked="" type="checkbox"/>	A		The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.

3.23 Logical AND operator

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
5-14-1(req)	<input checked="" type="checkbox"/>	A		The right hand operand of a logical && or operator shall not contain side effects.

3.24 Assignment operators

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
5-17-1(req)	<input type="checkbox"/>	M ⁽¹⁾		The semantic equivalence between a binary operator and its assignment operator form shall be preserved.

⁽²⁾ QP/C++ does not overload any operators.

3.25 Comma operator

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
5-18-1(req)	<input checked="" type="checkbox"/>	A		The comma operator shall not be used.

3.26 Constant expressions

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
5-19-1(adv)	<input checked="" type="checkbox"/>	A		Evaluation of constant unsigned integer expressions should not lead to wrap-around.

3.27 Expression statement

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
6-2-1(req)	<input checked="" type="checkbox"/>	A		Assignment operators shall not be used in sub-expressions.
6-2-2(req)	<input checked="" type="checkbox"/>	A		Floating-point expressions shall not be directly or indirectly tested for equality or inequality.
6-2-3(req)	<input checked="" type="checkbox"/>	A		Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character.

3.28 Compound statement

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
6-3-1(req)	<input checked="" type="checkbox"/>	A		The statement forming the body of a <code>switch</code> , <code>while</code> , <code>do ... while</code> or <code>for</code> statement shall be a compound statement.

3.29 Selection statements

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
6-4-1(req)	<input checked="" type="checkbox"/>	A		An <code>if (condition)</code> construct shall be followed by a compound statement. The <code>else</code> keyword shall be followed by either a compound statement, or another <code>if</code> statement.
6-4-2(req)	<input checked="" type="checkbox"/>	A		All <code>if ... else if</code> constructs shall be terminated with an <code>else</code> clause.
6-4-3(req)	<input type="checkbox"/>	M	6.4 ⁽¹⁾	A <code>switch</code> statement shall be a well-formed <code>switch</code> statement.
6-4-4(req)	<input type="checkbox"/>	M		A <code>switch-label</code> shall only be used when the most closely-enclosing compound statement is the body of a <code>switch</code> statement.
6-4-5(req)	<input checked="" type="checkbox"/>	A	6.4 ⁽¹⁾	An unconditional <code>throw</code> or <code>break</code> statement shall terminate every non-empty <code>switch-clause</code> .
6-4-6(req)	<input checked="" type="checkbox"/>	A	6.4 ⁽¹⁾	The final clause of a <code>switch</code> statement shall be the <code>default-clause</code> .
6-4-7(req)	<input checked="" type="checkbox"/>	A		The condition of a <code>switch</code> statement shall not have <code>bool</code> type.
6-4-8(req)	<input checked="" type="checkbox"/>	A		Every <code>switch</code> statement shall have at least one <code>case-clause</code> .

⁽¹⁾The published examples of state machine implementations with QP/C++ deviate from MISRA-C++ rules 6-4-3, 6-4-5, and 6-4-6. However, a fully compliant implementation is also possible and is described in Section 6.4.

3.30 Iteration statements

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
6-5-1(req)	<input type="checkbox"/>	M		A <code>for</code> loop shall contain a single loop-counter which shall not have floating type.
6-5-2(req)	<input type="checkbox"/>	M		If loop-counter is not modified by <code>--</code> or <code>++</code> , then, within condition, the loop-counter shall only be used as an operand to <code><=</code> , <code><</code> , <code>></code> or <code>>=</code> .
6-5-3(req)	<input checked="" type="checkbox"/>	A		The loop-counter shall be modified within condition statement.
6-5-4(req)	<input type="checkbox"/>	M		The loop-counter shall be modified by one of: <code>-</code> , <code>++</code> , <code>--n</code> , or <code>+=n</code> ; where <code>n</code> remains constant for the duration of the loop.
6-5-5(req)	<input type="checkbox"/>	M		A loop-control-variable other than the loop-counter shall not be modified within condition or expression.
6-5-6(req)	<input type="checkbox"/>	M		A loop-control-variable other than the loop-counter which is modified in statement shall have type <code>bool</code> .

3.31 Jump statements

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
6-6-1(req)	<input type="checkbox"/>	M ⁽¹⁾		Any label referenced by a <code>goto</code> statement shall be declared in the same block, or in a block enclosing the <code>goto</code> statement.
6-6-2(req)	<input checked="" type="checkbox"/>	A ⁽¹⁾		The <code>goto</code> statement shall jump to a label declared later in the same function body.
6-6-3(req)	<input checked="" type="checkbox"/>	A ⁽²⁾		The <code>continue</code> statement shall only be used within a well-formed <code>for</code> loop.
6-6-4(req)	<input checked="" type="checkbox"/>	A ⁽³⁾		For any iteration statement there shall be no more than one <code>break</code> or <code>goto</code> statement used for loop termination.
6-6-5(req)	<input checked="" type="checkbox"/>	A	⁽⁴⁾	A function shall have a single point of exit at the end of the function.

⁽¹⁾ QP/C++ source code does not use `goto`.

⁽²⁾ QP/C++ source code does not use `continue`.

⁽³⁾ QP/C++ source code does not use `break` for loop termination.

⁽⁴⁾ The published examples of state handler function with QP/C++ deviate from MISRA-C++ rule 6-6-5. However, a fully compliant implementation is also possible and is described in Section 6.4.

3.32 Specifiers

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
7-1-1(req)	<input checked="" type="checkbox"/>	5.11		A variable which is not modified shall be <code>const</code> qualified.
7-1-2(req)	<input checked="" type="checkbox"/>	A		A pointer or reference parameter in a function shall be declared as pointer to <code>const</code> or reference to <code>const</code> if the corresponding object is not modified.

3.33 Enumeration declarations

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
7-2-1(req)	<input checked="" type="checkbox"/>	A		An expression with <code>enum</code> underlying type shall only have values corresponding to the enumerators of the enumeration.

3.34 Namespaces

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
7-3-1(req)	<input checked="" type="checkbox"/>	A ⁽¹⁾		The global namespace shall only contain <code>main</code> , namespace declarations and <code>extern "C"</code> declarations.
7-3-2(req)	<input checked="" type="checkbox"/>	A		The identifier <code>main</code> shall not be used for a function other than the global function <code>main</code> .
7-3-3(req)	<input checked="" type="checkbox"/>	A		There shall be no unnamed namespaces in header files.
7-3-4(req)	<input checked="" type="checkbox"/>	A		<code>using-directives</code> shall not be used.
7-3-5(req)	<input type="checkbox"/>	M		Multiple declarations for an identifier in the same namespace shall not straddle a <code>using-declaration</code> for that identifier.
7-3-6(req)	<input checked="" type="checkbox"/>	A		<code>using-directives</code> and <code>using-declarations</code> (excluding class scope or function scope <code>using-declarations</code>) shall not be used in header files.

⁽¹⁾ QP/C++ source code can be configured to place all declarations and definitions in the `QP::` namespace. This is the recommended setting for MISRA-C++:2008 compliance.

3.35 The asm declaration

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
7-4-1(doc)	<input type="checkbox"/>	M ⁽¹⁾		All usage of assembler shall be documented.
7-4-2(req)	<input checked="" type="checkbox"/>	A		Assembler instructions shall only be introduced using the <code>asm</code> declaration.
7-4-3(req)	<input type="checkbox"/>	M ⁽¹⁾		Assembly language shall be encapsulated and isolated.

⁽¹⁾ QP/C++ encapsulates and isolates potential use of assembler language in the macros `QF_INT_ENABLE()`, `QF_INT_DISABLE()`, `QF_CRIT_ENTRY()`, `QF_CRIT_EXIT()`, `QK_ISR_ENTRY()`, and `QK_ISR_EXIT()`

3.36 Linkage specifications

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
7-5-1(req)	<input checked="" type="checkbox"/>	A		A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.
7-5-2(req)	<input checked="" type="checkbox"/>	A		The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.
7-5-3(req)	<input checked="" type="checkbox"/>	A		A function shall not return a reference or a pointer to a parameter that is passed by reference or <code>const</code> reference.
7-5-4(adv)	<input checked="" type="checkbox"/>	A		Functions should not call themselves, either directly or indirectly.

3.37 Declarations — General

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
8-0-1(req)	<input checked="" type="checkbox"/>	A		An <i>init-declarator-list</i> or a <i>member-declarator-list</i> shall consist of a single <i>init-declarator</i> or <i>member-declarator</i> respectively.

3.38 Meaning of declarations

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
8-3-1(req)	<input checked="" type="checkbox"/>	A		Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.

3.39 Function definitions

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
8-4-1(req)	<input checked="" type="checkbox"/>	A		Functions shall not be defined using the ellipsis notation.
8-4-2(req)	<input checked="" type="checkbox"/>	A		The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.
8-4-3(req)	<input checked="" type="checkbox"/>	M		All exit paths from a function with non-void return type shall have an explicit return statement with an expression.
8-4-4(req)	<input checked="" type="checkbox"/> ⁽¹⁾	M		A function identifier shall either be used to call a function or it shall be preceded by <code>&</code> .

⁽¹⁾ PC-Lint reports the recommended use of the `&` operator on function names as the error 546 (suspicious use of `&`). This is actually contrary to the MISRA-C++:2008 guidelines, so the error 546 is suppressed in the `lib-qpcpp.lnt` option file.

3.40 Declarators — Initializers

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
8-5-1(req)	<input checked="" type="checkbox"/>	5.12		All variables shall have a defined value before they are used.
8-5-2(req)	<input checked="" type="checkbox"/>	A		Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.
8-5-3(req)	<input checked="" type="checkbox"/>	M		In an enumerator list, the <code>=</code> construct shall not be used to explicitly initialize members others than the first, unless all items are explicitly initialized.

⁽¹⁾ PC-Lint includes in this rule also checks of the constructor initializer lists. QP/C++ deviates from this rule only in the constructor initializer lists for `QEvt` and `QActive`.

3.41 Member functions

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
9-3-1(req)	<input checked="" type="checkbox"/>	A		<code>const</code> member functions shall not return non- <code>const</code> pointers or references to class-data.
9-3-2(req)	<input checked="" type="checkbox"/>	A		Member functions shall not return non- <code>const</code> handles to class-data.
9-3-3(req)	<input checked="" type="checkbox"/>	A		If a member function can be made <code>static</code> then it shall be made <code>static</code> , otherwise if it can be made <code>const</code> then it shall be made <code>const</code> .

3.42 Unions

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
9-5-1(req)	<input checked="" type="checkbox"/>	5.13 ⁽¹⁾		Unions shall not be used.

⁽¹⁾ QP/C++ deviates from this rule in the `QMAttr` type and in QS functions `QS::f32` and `QS::f64` in the `Q_SPY` configuration

3.43 Bit-fields⁽¹⁾

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
9-6-1(doc)	<input type="checkbox"/>	M		When the absolute positioning of bits representing a bit-field is required, then the behavior and packing of bit-fields shall be documented.
9-6-2(req)	<input checked="" type="checkbox"/>	A		Bit-fields shall be either <code>bool</code> type or an explicitly unsigned or signed integral type.
9-6-3(req)	<input checked="" type="checkbox"/>	A		Bit-fields shall not have <code>enum</code> type.
9-6-4(req)	<input checked="" type="checkbox"/>	A		Named bit-fields with signed integer type shall have a length of more than one bit.

⁽¹⁾ QP/C++ does not use bit fields at all.

3.44 Multiple base classes⁽¹⁾

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
10-1-1(adv)	<input checked="" type="checkbox"/>	A		Classes should not be derived from virtual bases.
10-1-2(req)	<input type="checkbox"/>	M		A base class shall only be declared virtual if it is used in a diamond hierarchy.
10-1-3(req)	<input checked="" type="checkbox"/>	A		An accessible base class shall not be both virtual and non-virtual in the same hierarchy.

⁽¹⁾ QP/C++ does not use virtual base classes at all.

3.45 Member name lookup

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
10-2-1(adv)	<input type="checkbox"/>	M ⁽¹⁾		All accessible entity names within a multiple inheritance hierarchy should be unique.

⁽¹⁾ QP/C++ does not use multiple inheritance at all.

3.46 Virtual Functions⁽¹⁾

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
10-3-1(req)	<input type="checkbox"/>	M		There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.
10-3-2(req)	<input checked="" type="checkbox"/>	A		Each overriding virtual function shall be declared with the <code>virtual</code> keyword.
10-3-3(req)	<input type="checkbox"/>	M		A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.

⁽¹⁾ Except for virtual destructors in QF_{sm} and QH_{sm} base classes, QP/C++ does not use virtual functions.

3.47 Constructors

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
12-1-1(req)	<input checked="" type="checkbox"/>	A		An object's dynamic type shall not be used from the body of its constructor or destructor.
12-1-2(adv)	<input checked="" type="checkbox"/>	A		All constructors of a class should explicitly call a constructor for all of its immediate base classes and all virtual base classes.
12-1-3(req)	<input type="checkbox"/>	M ⁽¹⁾		All constructors that are callable with a single argument of fundamental type shall be declared explicit.

⁽¹⁾ The QP/C++ code does not contain any constructors with single argument of fundamental type.

3.48 Copying class objects

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
12-8-1(req)	<input checked="" type="checkbox"/>	A		A copy constructor shall only initialize its base classes and the non-static members of the class of which it is a member.
12-8-2(req)	<input checked="" type="checkbox"/>	A		The copy assignment operator shall be declared protected or private in an abstract class.

3.49 Template declarations⁽¹⁾

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
14-5-1(req)	<input checked="" type="checkbox"/>	A		A non-member generic function shall only be declared in a namespace that is not an associated namespace.
14-5-2(req)	<input checked="" type="checkbox"/>	A		A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter.
14-5-3(req)	<input checked="" type="checkbox"/>	A		A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.

⁽¹⁾ QP/C++ code does not use templates anywhere in the code.

3.50 Name resolution⁽¹⁾

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
14-6-1(req)	<input checked="" type="checkbox"/>	A		In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or <code>this-></code>
14-6-2(req)	<input checked="" type="checkbox"/>	A		The function chosen by overload resolution shall resolve to a function declared previously in the translation unit.

⁽¹⁾ QP/C++ code does not use templates anywhere in the code.

3.51 Template instantiation and specialization⁽¹⁾

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
14-7-1(req)	<input checked="" type="checkbox"/>	A		All class templates, function templates, class template member functions and class template static members shall be instantiated at least once.
14-7-2(req)	<input type="checkbox"/>	M		For any given template specialization, an explicit instantiation of the template with the template-arguments used in the specialization shall not render the program ill-formed.
14-7-3(req)	<input checked="" type="checkbox"/>	A		All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template.

⁽¹⁾ QP/C++ code does not use templates anywhere in the code.

3.52 Function template specialization⁽¹⁾

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
14-8-1(req)	<input checked="" type="checkbox"/>	A		Overloaded function templates shall not be explicitly specialized.
14-8-2(adv)	<input checked="" type="checkbox"/>	A		The viable function set for a function call should either contain no function specializations, or only contain function specializations.

⁽¹⁾ QP/C++ code does not use templates anywhere in the code.

3.53 Exception handling — General⁽¹⁾

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
15-0-1(doc)	<input type="checkbox"/>	M		Exceptions shall only be used for error handling.
15-0-2(adv)	<input checked="" type="checkbox"/>	A		An exception object should not have pointer type.
15-0-3(req)	<input checked="" type="checkbox"/>	A		Control shall not be transferred into a <code>try</code> or <code>catch</code> block using a <code>goto</code> or a <code>switch</code> statement.

⁽¹⁾ QP/C++ code does not throw or catch exceptions anywhere in the code.

3.54 Throwing an exception

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
15-1-1(req)	<input type="checkbox"/>	M		The assignment-expression of a throw statement shall not itself cause an exception to be thrown.
15-1-2(req)	<input checked="" type="checkbox"/>	A		NULL shall not be thrown explicitly.
15-1-3(req)	<input checked="" type="checkbox"/>	A		An empty throw (<code>throw;</code>) shall only be used in the compound-statement of a catch handler.

3.55 Handling an exception

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
15-3-1(req)	<input checked="" type="checkbox"/>	A		Exceptions shall be raised only after start-up and before termination of the program.
15-3-2(adv)	<input type="checkbox"/>	A		There should be at least one exception handler to catch all otherwise unhandled exceptions
15-3-3(req)	<input type="checkbox"/>	M		Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.
15-3-4(req)	<input checked="" type="checkbox"/>			Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.
15-3-5(req)	<input checked="" type="checkbox"/>			A class type exception shall always be caught by reference.
15-3-6(req)	<input type="checkbox"/>			Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.
15-3-7(req)	<input checked="" type="checkbox"/>			Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.

3.56 Exception specification

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
15-4-1(req)	<input checked="" type="checkbox"/>	A		If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.

3.57 Exception handling — Special functions

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
15-5-1(req)	<input checked="" type="checkbox"/>	A		A class destructor shall not exit with an exception.
15-5-2(req)	<input checked="" type="checkbox"/>	A		Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).
15-5-3(req)	<input checked="" type="checkbox"/>	M		The <code>terminate()</code> function shall not be called implicitly.

3.58 Processing directives — General

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
16-0-1(req)	<input checked="" type="checkbox"/>	A		<code>#include</code> directives in a file shall only be preceeded by other preprocessor directives or comments.
16-0-2(adv)	<input checked="" type="checkbox"/>	A		Macros shall only be <code>#define</code> 'd or <code>#undef</code> 'd in the global namespace.
16-0-3(req)	<input checked="" type="checkbox"/>	M		<code>#undef</code> shall not be used.
16-0-4(req)	<input checked="" type="checkbox"/>	5.14		Function-like macros shall not be defined.
16-0-5(req)	<input checked="" type="checkbox"/>	A		Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.
16-0-6(req)	<input checked="" type="checkbox"/>	A		In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of <code>#</code> or <code>##</code> .
16-0-7(req)	<input checked="" type="checkbox"/>	A		Undefined macro identifiers shall not be used in <code>#if</code> and <code>#elif</code> preprocessor directives, except as operands to the <code>defined</code> operator.
16-0-8(req)	<input checked="" type="checkbox"/>	A		If the <code>#</code> token appears as the first token on a line, then it shall be immediately followed by a preprocessing token.

3.59 Conditional inclusion

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
16-1-1(req)	<input checked="" type="checkbox"/>	A		The <code>defined</code> preprocessor operator shall only be used in one of the two standard forms.
16-1-2(req)	<input checked="" type="checkbox"/>	A		All <code>#else</code> , <code>#elif</code> and <code>#endif</code> preprocessor directives shall reside in the same file as <code>#if</code> or <code>#ifdef</code> directive to which they are related.

3.60 Source file inclusion

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
16-2-1(req)	<input type="checkbox"/>	5.15		Preprocessor shall only be used for file inclusion and include guards.
16-2-2(req)	<input checked="" type="checkbox"/>	5.15		C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers.
16-2-3(req)	<input checked="" type="checkbox"/>	A		Include guards shall be provided.
16-2-4(req)	<input checked="" type="checkbox"/>	A		The <code>'</code> , <code>"</code> , <code>/*</code> or <code>//</code> characters shall not occur in a header file name.
16-2-5(adv)	<input checked="" type="checkbox"/>	A		The <code>\</code> character should not occur in a header file name.
16-2-6(req)	<input checked="" type="checkbox"/>	A		The <code>#include</code> directive shall be followed by either a <code><filename></code> or <code>"filename"</code> sequence.

3.61 Macro replacement

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
16-3-1(req)	<input checked="" type="checkbox"/>	A		There shall be at most one occurrence of the <code>#</code> or <code>##</code> operators in a single macro definition.
16-3-2(adv)	<input checked="" type="checkbox"/>	5.16 ⁽¹⁾	. ⁽¹⁾	The <code>#</code> and <code>##</code> operators should not be used.

⁽¹⁾ QP/C++ code uses a single `##` operator in the `Q_NEW()` macro and only when constructors for `QEvts` are allowed (`Q_EVT_CTR` macro is defined). Additionally, the “dictionary” QS macros use a single `#` operator.

3.62 Pragma directive

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
16-6-1(doc)	<input type="checkbox"/>	M		All uses of the <code>#pragma</code> directive shall be documented.

3.63 Library introduction — General

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
17-0-1(req)	<input checked="" type="checkbox"/>	A	(1)	Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.
17-0-2(req)	<input checked="" type="checkbox"/>	A	(1)	The names of standard library macros and objects shall not be reused.
17-0-3(req)	<input type="checkbox"/>	M	(1)	The names of standard library functions shall not be overridden.
17-0-4(doc)	<input checked="" type="checkbox"/>	A	(1)	All library code shall conform to MISRA C++.
17-0-5(req)	<input checked="" type="checkbox"/>	A	(1)	The <code>setjmp</code> macro and the <code>longjmp</code> function shall not be used.

(1) Requires analysis of the complete application source code, including all libraries

3.64 Language support library — General⁽¹⁾

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
18-0-1(req)	<input checked="" type="checkbox"/>	A		The C library shall not be used.
18-0-2(req)	<input checked="" type="checkbox"/>	A		The library functions <code>atof</code> , <code>atoi</code> and <code>atol</code> from library <code><cstdlib></code> shall not be used.
18-0-3(req)	<input checked="" type="checkbox"/>	A		The library functions <code>abort</code> , <code>exit</code> , <code>getenv</code> and <code>system</code> from library <code><cstdlib></code> shall not be used.
18-0-4(req)	<input checked="" type="checkbox"/>	A		The time handling functions of library <code><ctime></code> shall not be used.
18-0-5(req)	<input checked="" type="checkbox"/>	A		The unbounded functions of library <code><cstring></code> shall not be used.

(1) QP/C++ code does not rely in any way on any standard C or C++ libraries. The `stdint.h` header file used in QP/C++ does not require any code to be pulled from the standard libraries.

3.65 Language support library — Implementation properties

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
18-2-1(req)	<input checked="" type="checkbox"/>	A		The macro <code>offsetof</code> shall not be used.

3.66 Language support library — Dynamic memory management

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
18-4-1(req)	<input checked="" type="checkbox"/>	A		Dynamic heap memory allocation shall not be used.

3.67 Language support library — Other runtime support

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
18-7-1(req)	<input checked="" type="checkbox"/>	A		The signal handling facilities of <code><csignal></code> shall not be used.

3.68 Diagnostics library — Error numbers

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
19-3-1(req)	<input checked="" type="checkbox"/>	A		The error indicator <code>errno</code> shall not be used.

3.69 Input/Output library — General

Rule No.	PC-Lint	QP/C++	QP app.	Rule Headline
27-0-1(req)	<input checked="" type="checkbox"/>	A		The stream input/output library <code><cstdio></code> shall not be used.

4 Beyond MISRA: Compliance with Additional Rules and Standards

4.1 Strong Type Checking

The philosophy of the C++ language is to assume that the programmers know what they are doing, which can mean that if errors are made they are allowed to pass unnoticed by the language. An area in which C++ is particularly weak in this respect is that of “type checking”. C++ compilers will not object, for example, if the programmer tries to store a floating point number in an integer that they are using to represent a true/false value. Most such mismatches are simply forced to become compatible. If C++ is presented with a square peg and a round hole it doesn't complain, but makes them fit!

PC-Lint has an advanced **strong type checking** capabilities (see Chapter 9 in the PC-Lint Manual [PC-Lint 08]), which includes sophisticated dimensional analysis of the types resulting from **combining** other types (e.g., *velocity_miles_per_hour = distance_miles / time_hours*). The strong type checking is activated in PC-Lint with the `-strong (AJX)` option.

NOTE: The strong type checking of PC-Lint takes the static analysis to the next level beyond MISRA-C++, because it can turn C++ into a truly **strongly-typed language**.

However, a software system can become “strongly-typed” only if it is built from components that are also “strongly-typed”. Fortunately, the **QP/C++ framework is “strongly typed”**, meaning that it passes cleanly the PC-Lint analysis with the `-strong (AJX)` option activated. This is an immense benefit for QP/C++ users, because it allows the application-level code to take advantage of the strong type checking.

4.2 Quantum Leaps C/C++ Coding Standard

Although intentionally not addressed by the MISRA-C++:2008 guidelines, the use of a consistent coding style is a very important aspect of developing safety-related code. The QP/C++ code strictly follows to the Quantum Leaps C/C++ Coding Standard [QL-Code 11].

5 Deviation Procedures for QP/C++ Source Code

This section documents deviations from MISRA-C++:2008 rules in the **QP/C++ source code**.

NOTE: The use of the MISRA-C++:2008 rule headlines in this document has been licensed to Quantum Leaps by [MIRA Ltd.](http://MIRA.Ltd)

5.1 Rule 0-1-1(req) and 0-1-2(req)

A project shall not contain unreachable code.

A project shall not contain infeasible paths.

Deviation from these rules occurs only in the QS software tracing instrumentation and is related to QS filters. When QS tracing is disabled, certain QS trace records might look like unreachable code or infeasible paths (e.g., `if (false) ...`).

5.2 Rule 0-1-11(req)

There shall be no unused parameters (named or unnamed) in non-virtual functions.

Deviation from this rule occurs in the standard “vanilla” and QK ports of QP/C++ (and perhaps in other ports that do not use the per-task stacks). For the sake of wide-range portability, the signature of the functions `QActive::start()` must include the stack memory and stack size, but these parameters might not be actually used in all QP/C++ ports.

5.3 Rule 3-9-3(req)

The underlying bit representations of floating-point values shall not be used.

Deviation from this rule occurs only in the QS software tracing instrumentation (**not** in production code) and is related to serialization of floating point numbers. The deviation is allowed only in the context of the functions `QS::f32()` and `QS::f64()`.

5.4 Rule 5-0-15(req)

Array indexing shall be the only allowed form of pointer arithmetic.

Deviation from this rule is related to the general policy of the QP/C++ framework with respect to memory allocation. In QP/C++, all memory (e.g., memory pools or event queue buffers) is pre-allocated by the application code and then passed as pointer and size of the memory to the framework. Subsequently, the memory is accessed using array indexing, but from the original base pointer, not from a true array—hence the deviation from rule 5-0-15.

The deviation from rule 5-0-15 is encapsulated in the QP/C++ internal macro `QF_PTR_AT_()`, and this is the only context allowed to deviate per this procedure.

5.5 Rule 5-0-18(req)

>, >=, <, <= shall not be applied to pointers, except where they point to the same array.

Deviation from this rule occurs in only one assertion in the QP/C++ code (file `qmp_put.cpp`). The precondition assertion checks that the returned pointer to memory block indeed comes from the memory pool to which it is being returned. If the block belongs to the pool, the pointer comparison is made within

the same array, so the rule is actually **not** violated. The transgression occurs only if the block pointer is not in range.

The assertion of pointer range is encapsulated in the internal macro `QF_PTR_RANGE_` and has proved to be very valuable in ensuring the system integrity and in this particular context the benefits outweigh the risk of deviating from the MISRA rule.

5.6 Rule 5-2-4(req)

C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used.

Deviation from this rule occurs only in the QP assertions. For practical reasons, the `qassert.h` header file is designed to be useful both in C++ and C modules, as many embedded projects use both languages. The strict use of C++ style casts would preclude using assertions in C modules, which could discourage programmers from using QP assertions everywhere in the code. The deviation from rule 5-2-4 should be limited to QP assertions.

5.7 Rule 5-2-5(req)

A cast shall not remove any const or volatile qualification from the type of a pointer or reference.

QP/C++ code deviates from this rule only in the internal macros `QF_EVT_REF_CTR_INC_`, `QF_EVT_REF_CTR_DEC_`, and `QF_EPOOL_PUT_`, where the `const` qualification needs to be occasionally discarded. The discarding of `const` is always preceded by a check that a given event is indeed dynamic (by testing the `QEvt.poolId_` member). Deviation from this rule is considered a better tradeoff for safety and design correctness than not using the `const` qualification for events at all.

5.8 Rule 5-2-7(req)

An object with integer type or pointer to void type shall not be converted to an object with pointer type.

QP/C++ code deviates from this rule in the internal macro `QF_EPOOL_GET_`, where a generic pointer to a memory block is cast to a specific object type. This sort of cast is inevitable in any memory allocator, but is strictly encapsulated inside the `QF_EPOOL_GET_` macro.

5.9 Rule 5-2-9(adv)

A cast should not convert a pointer type to an integral type.

Deviation from this rule occurs only in the QS software tracing instrumentation (**not** in production code). The QS code needs to output pointers to functions and pointers to objects by means of the macros `QS_FUN_`, `QS_OBJ_`, respectively. The deviation is allowed only in the context of these macros.

5.10 Rule 5-2-10(adv)

The increment (++) and decrement (--) operators should not be mixed with other operators in an expression

Deviation from this rule occurs only in the QS software tracing macro `QS_TEC_` (**not** in production code).

5.11 Rule 7-1-1(req)

A variable which is not modified shall be const qualified.

Deviation from this rule might occur in some QP/C++ ports, which don't use all the parameters of the `QActive::start()` method. For example, QP/C++ ports to the built-in "Vanilla" and QK kernels don't need the per-task stack and re-use the stack and stack-size parameters for different purposes. In these cases the stack and/or the stack-size is not modified and could be const-qualified, but this would interfere with the portability of the QP/C++ framework.

5.12 Rule 8-5-1(req)

All variables shall have a defined value before they are used.

The QP/C++ code complies with this rule everywhere, except the compiler-initialization lists for `QActive` and `QEvt` classes, which PC-Lint includes in checking this rule.

The constructor of `QActive` intentionally does not initialize the `m_prio`, `m_eQueue`, `m_thread`, and `m_osObject` members, because in QP/C++ the initialization of active objects is a two-stage process (with the second stage of initialization happening in the top-most initial transition) to give the application designer the necessary control over the timeline of initialization. Also, some members of the `QActive` base class exist only in the certain QP/C++ ports and don't exist in others.

5.13 Rule 9-5-1(req)

Unions shall not be used.

For production code, deviation from this rule occurs in the QEP component in the data type `QMAAttr`, which used to specify the private state machine attribute in the `QMsm` base class. The use of a union in this case is strictly encapsulated inside the QEP event processor code and is never accessed in the application-level code. The justification of deviating from the rule 9-5-1 are significant savings in RAM for every state machine in the system as well as backwards-compatibility with the classes `QHsm` and `QFsm`.

Deviation from this rule occurs also in the QS software tracing instrumentation (**not** in production code) and is related to serialization of floating point numbers. The deviation is allowed only in the context of the functions `QS::f32()` and `QS::f64()`.

5.14 Rule 16-0-4(req), 16-2-1(req), and 16-2-2(req)

Function-like macros shall not be defined.

QP/C++ uses function-like macros in situations that an (inline) function cannot handle. For example, the QS software tracing or assertions need to be configurable to be activated or deactivated at compile-time (without actually changing the source code). Finally, function-like macros are used to encapsulate the very specific contexts of deviations from the MISRA rules, such as specific type casts. All function-like macros deviating from the rule 16-0-4 are listed explicitly in the option files, so any other macros violating the rule will be reported.

5.15 Rule 16-2-1(req) and 16-2-2(req)

16-2-1: Preprocessor shall only be used for file inclusion and include guards.

16-2-2: C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers.

A framework has much stronger requirements for configurability and portability than a specific application. The QP/C++ achieves the high degree and ease of configurability through the concept of the Platform

Abstraction Layer (PAL). The PAL, however, derives its portability from the use of preprocessor macros. For example, the QP/C++ ports must be able to configure the desired size of the event signal or the maximum number of active objects in the system. This information then determines the most optimal data structures and algorithms to use. This is not achievable through inline functions. Also, preprocessor macros are used to encapsulate the very specific contexts of deviations from the MISRA rules, such as specific type casts. All macros deviating from the rules 16-2-1 and 16-2-2 are listed explicitly in the option files, so any other macros violating the rule will be reported.

5.16 Rule 16-3-2(adv)

The # and ## operators should not be used.

The QP/C++ source code uses the ## operator in the `Q_NEW()` macro, in which case, the `Q_NEW()` macro must take the variable number of arguments for the specific `QEvt`-subclass constructor (variadic macro).

In the QS software tracing configuration, the QS “dictionary” records use a single # operator.

6 Deviation Procedures for Application-Level Code

This section documents deviations from MISRA-C++:2008 rules in the **application-level** code, which are caused by the QP/C++ framework design or implementation. This section also describes workarounds to avoid some deviations.

NOTE: The use of the MISRA-C++:2008 rule headlines in this document has been licensed to Quantum Leaps by [MIRA Ltd.](http://MIRA.Ltd)

6.1 Rule 5-2-6(req)

A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.

The QP/C++ applications deviate from rule 5-2-6, because the state-handler functions signatures in the derived subclasses of `QHsm` or `QFsm` are not exactly identical with the state-handler pointer-to-function `StateHandler`. The deviation from rule 5-2-6 is encapsulated in the macro `Q_STATE_CAST()`, which is defined as follows (file `qep.h`):

```
#define Q_STATE_CAST(handler_) (reinterpret_cast<QP::QStateHandler>(handler_))
```

In the QP/C++ applications, the macro `Q_STATE_CAST()` is used to cast the initial pseudo-state-handler in the constructor, like this:

```
Philo::Philo()  
: QP::QActive(Q_STATE_CAST(&Philo::initial)),  
  m_timeEvt(TIMEOUT_SIG)  
{}
```

The need to deviate from the rule 5-2-6 is a consequence of using function pointers in conjunction with class inheritance, which are both fundamental to the QP/C++ framework. This, very particular, deviation from rule 5-2-6 is safe and is allowed only in the context of state-handler functions, which are related.

Additionally, macros `Q_TRAN()` and `Q_SUPER()` also use the macro `Q_STATE_CAST()`, so they too indirectly deviate from the rule 5-2-6.

6.2 Rule 5-2-7(req)

An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.

The QP/C++ applications deviate from rule 5-2-7 because of downcasting the generic event pointer (`QEvt const *`) to the specific event in the state machine code. The QP/C++ framework encapsulates this deviation in the macro `Q_EVENT_CAST()`. The code snippet below shows a use case. Please note that the macro `Q_EVENT_CAST()` does not cast the `const` away, so writing to the event pointer is not allowed.

```
case EAT_SIG: {  
    if (Q_EVENT_CAST(TableEvt)->philoNum == PHILO_ID(me)) . . .
```

6.3 Rule 5-2-8(req)

An object with integer type or pointer to void type shall not be converted to an object with pointer type.

The QP/C++ applications might deviate from rule 5-2-8 when they need to access specific hard-coded hardware addresses directly. The QP/C++ framework encapsulates this deviation in the macro `Q_UINT2PTR_CAST()`. The following code snippet provides a use case of this macro:

```
#define QK_ISR_EXIT() . . . \  
    *Q_UINT2PTR_CAST(uint32_t, 0xE000ED04U) = \  
        static_cast<uint32_t>(0x10000000U); \  
    . . .
```

6.4 Rule 6-4-3(req), 6-4-5(req), and 6-4-6(req)

6-4-3(req): A switch statement shall be a well-formed switch statement.

6-4-5(req): An unconditional throw or break statement shall terminate every non-empty switch-clause.

6-4-6(req): The final clause of a switch statement shall be the default clause.

The traditional way of implementing state-handler functions in QP/C++, as described in the book “Practical UML Statecharts” [PSiCC2 08] deviates from the rules 6-4-3, 6-4-5, and 6-4-6. However, it is also possible to avoid all these deviations, which will become clearer after describing the implementation.

The MISRA-compliant state handler implementation is used in the DPP examples with lint described in Section 8. The following Listing 4 shows an example of MISRA-compliant state handler function. The explanation section immediately following the listing highlights the important points.

NOTE: Starting from the QM version **2.2.00**, the QM modeling tool generates MSRA-C++ compliant code structure shown in Listing 4.

Listing 4: MISRA-C++ compliant state handler implementation

```
QP::QState Philo::hungry(Philo * const me, QP::QEvt const * const e) {  
(1)     QP::QState status;  
    switch (e->sig) {  
        case Q_ENTRY_SIG: {  
            AQ_Table->POST(pe, Q_NEW(TableEvt, HUNGRY_SIG, PHILO_ID(me)));  
(2)             status = Q_HANDLED();  
(3)             break;  
        }  
        case EAT_SIG: {  
(4)             if (Q_EVENT_CAST(TableEvt)->philoNum == PHILO_ID(me)) {  
(5)                 status = Q_TRAN(&Philo::eating);  
            }  
(6)             else {  
(7)                 status = Q_UNHANDLED();  
            }  
(8)             break;  
        }  
(9)     default: {  
(10)        status = Q_SUPER(&QHsm::top);  
    }  
}
```

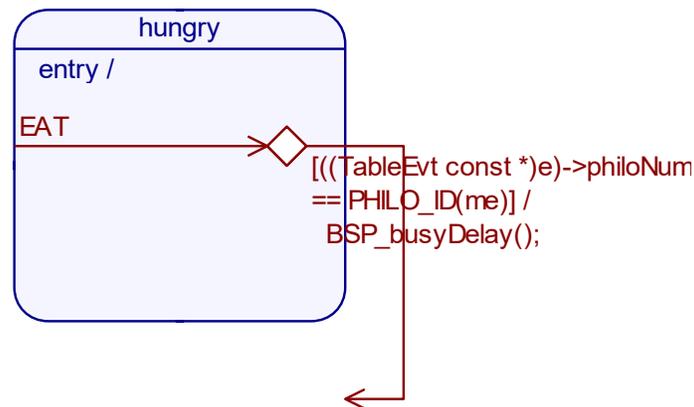
```
(11)         break;
            }
        }
(12)     return status;
    }
```

- (1) The automatic variable `status` will store the status of the processing to return. Please note that the `status` variable is not initialized.

NOTE: The `status` variable is not initialized intentionally, to catch any path through the code that would not set the value explicitly. The vast majority of compilers (including, of course PC-Lint) raise a warning about an uninitialized variable to alert of the problem. However, it is highly recommended to test each particular compiler for the ability to report this problem.

- (2) The return value is set to `Q_HANDLED()` macro. This tells the QEP event processor that the entry action has been handled.
- (3) According to the recommended MISRA-C `switch` statement structure, the case is terminated with a `break`.
- (4) The guard condition is coded as usual with an `if`-statement. Please note the use of the `Q_EVENT_CAST()` macro to downcast the generic event pointer to `TableEvt` class.
- (5) When the guard condition in the `if`-statement evaluates to `TRUE`, the return value is set to `Q_TRAN()` macro. This macro tells the QEP event processor that the event has been handled and that the transition to state `Philo::eating` needs to be taken.

Figure 1: A choice point without an explicit [else] branch



- (6,7) When the guard condition evaluates to `FALSE`, and the state model does not explicitly prescribe how to handle this situation (see Figure 1), the code should have an implicit `else` branch, in which the return value is set from the macro `Q_UNHANDLED()`. This specific return value will cause the QEP event processor to propagate the event to the superstate, which is exactly what the UML specification prescribes in this case.

NOTE: The `Q_UNHANDLED()` macro has been introduced in QP version **4.5.01** exactly for MISRA compliance. The QM graphical modeling tool generates MISRA-compliant code described in this section starting from the version **2.2.00**.

- (8) According to the recommended MISRA-C++ `switch` statement structure, the `case` is terminated with a `break`
- (9) According to the recommended `switch` structure, the `default`-clause is the final clause.
- (10) Inside the `default`-clause, the return value is set to `Q_SUPER()` macro. This tells the QEP event processor that `QHsm::top` is the superstate of this state.
- (11) According to the recommended MISRA-C `switch` statement structure, the `default`-clause is terminated with a `break`
- (12) In compliance with MISRA-C rules 14.7 and 16.8, the function terminates with the single `return` statement.

7 Summary

The QP/C++ framework complies with most of the MISRA-C++:2008 rules and all remaining deviations are carefully insulated and encapsulated into very specific contexts. The framework goes even beyond MISRA, by complying with strict type checking and a consistent, documented coding standard.

QP/C++ comes with extensive support for automatic rule checking by means of PC-Lint, which is designed not just for proving compliance of the QP/C++ framework code, but more importantly, to aid in checking compliance of the application-level code. Any organization engaged in designing safety-related embedded software could benefit from the unprecedented quality infrastructure built around the QP/C++ framework.

8 Related Documents and References

Document	Location
[MISRA-C++:2008] MISRA-C++:2008 Guidelines for the Use of the C++ language in Critical Systems, MISRA, June 2008, ISBN: 978-1-906400-03-3 paperback ISBN: 978-1-906400-04-0 PDF	Available for purchase from MISRA website http://www.misra.org.uk
[ISO C++ 03] ISO/IEC 14882:2003, The C++ Standard Incorporating Technical Corrigendum 1, International Organization for Standardization, 2003.	www.iso.org/iso/catalogue_detail.htm?csnumber=38110
[PC-Lint 08] "Reference Manual for PC-lint/FlexeLint: A Diagnostic Facility for C and C++", Software Version 9.00 and Later, Gimpel Software, September, 2008	Bundled with PC-Lint from Gimpel http://www.gimpel.com
[PC-Lint-MISRA-C++:2008] PC-Lint/FlexeLint Support for MISRA C++,	Available from Gimpel after request
[QL-Code 11] "Application Note: C/C++ Coding Standard", Quantum Leaps, LLC, 2011	http://www.state-machine.com/resources/AN_QL_Coding_Standard.pdf
[QL AN-DPP 08] "Application Note: Dining Philosopher Problem Application", Quantum Leaps, LLC, 2008	http://www.state-machine.com/resources/AN_DPP.pdf

9 Contact Information

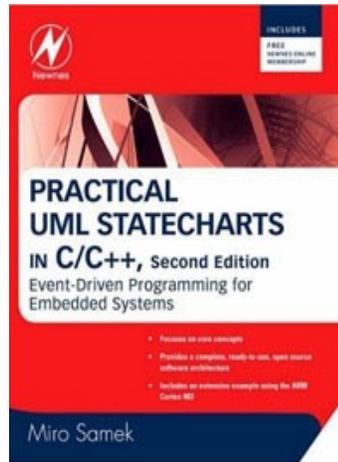
Quantum Leaps, LLC

+1 919 360-5668 (Office)

+1 919 869-2998 (FAX)

Email: info@state-machine.com

Web : <https://www.state-machine.com>



“*Practical UML Statecharts in C/C++, Second Edition: Event Driven Programming for Embedded Systems*”,
by Miro Samek,
Newnes, 2008

Legal Disclaimers

Information in this document is believed to be accurate and reliable. However, Quantum Leaps does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information.

Quantum Leaps reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

All designated trademarks are the property of their respective owners.

