Quantum® LeaPs
innovating embedded systems

MISRA

# Application Note
# QP/C™ MISRA-C:2004 Compliance Matrix

**Document Revision D**
**February 2011**



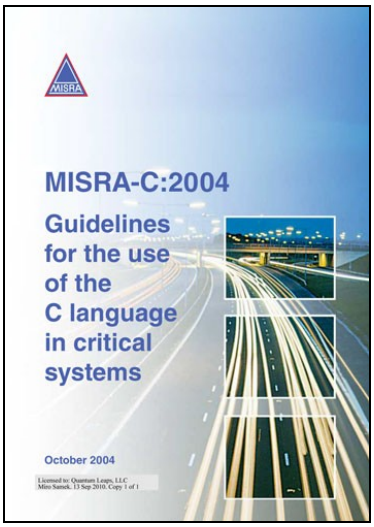
info@quantum-leaps.com
www.state-machine.com

## Table of Contents

# 1 Introduction

This Application Note describes the compliance of the QP/C™ state machine framework version **4.4.00** 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:2004]. This Application Note is designed to be applied to production code in safety-related embedded systems.

## 1.1 About MISRA-C:2004

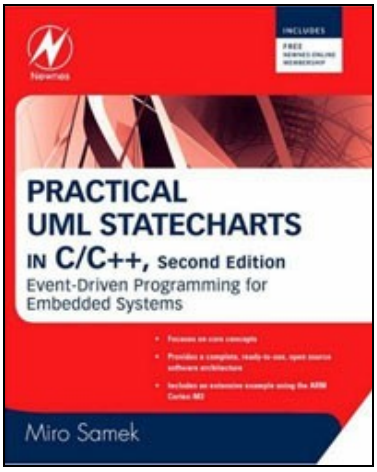



MISRA, the Motor Industry Software Reliability Association (www.misra.org.uk), is a collaboration between vehicle manufacturers, component suppliers, and engineering consultancies, which seeks to promote best practices in developing **safety-related electronic systems** in road vehicles and other embedded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

Since it's original publication in 1998 [MISRA-C:1998], the 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 2004 MISRA published the revised set of rules known as the MISRA-C:2004.

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 embedded systems. The [MISRA-C:2004] guidelines define this language subset by means of 141 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.

## 1.2 About QP™




**QP/C™** is a lightweight, open source, state machine framework for developing event-driven embedded software. The QP/C framework enables software developers to build well-structured embedded applications as systems of concurrently executing hierarchical state machines (UML statecharts). 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**. To this end, a framework based on the proven concepts of state machines and active objects provides a more robust and safer design platform than a primitive “super-loop” or an unstructured design based on a traditional Real-Time Operating System (RTOS). The QP/C framework enforces **safe** design and encapsulates or eliminates the troublesome RTOS mechanisms, such as semaphores, which many programmers do not use correctly or safely.

# 2 Checking MISRA Compliance with PC-Lint/FlexeLint

The [MISRA-C:2004] 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:2004 required rule 21.1.

> **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** 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 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:2004 guidelines in PC/Lint is via the option file [PC-Lint/MISRA-C:2004]:

```
au-misra2.lnt
```

This file contains the appropriate options to activate and annotate PC-Lint messages dealing with MISRA-C:2004. PC-Lint can report deviations from several MISRA C rules with messages 960 and 961. Additional rules, are covered in other messages, the details of which you can find listed in the `au-misra2.lnt` file.

> **NOTE:** The `au-misra2.lnt` configuration file is also the best overall **documentation** on MISRA-C:2004 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 [PC-Lint 08]). 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 `qpc\include\` and `qpc\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 (qpc\include\) and "lining" QP/C source code itself (qpc\ports\lint\).**

```
%QPC%\                    - QP/C Root Directory (environment variable QPC)
  |
  +-include\              - QP/C platform-independent includes
  | +-au-ds.lnt           - Dan Saks recommendations
  | +-au-misra2.lnt       - Main PC-Lint MISRA-C:2004 compliance options
  | +-lib-qpc.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
  | | | +-MISRA_Exemplar_Suite_test\ - MISRA Exemplar Suite rule coverage test
  | | | | +-lin.bat       - Batch file to invoke PC-Lint to run analysis of MES
  | | | | +-options.lnt   - PC/Lint options for "linting" MES
  | | | | +-. . .
  | | | |
  | | | +-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
```

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

As shown in Listing 1, the directory `%QPC%\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%QPC%\include\` include directory to PC-Lint accomplishes both inclusion of QP/C header files and PC-Lint options.

Note that the `%QPC%\include\` directory contains **all** PC-Lint option files used in "linting" the code, including the standard MISRA-C:2004 `au-misr2.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.

**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.

### 2.1.1 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-qpc.lnt option file

The most important file for "linting" QP/C applications is the `lib-qpc.lnt` option file. This file handles all deviations from the MISRA-C:2004 rules, which might arise at the **application-level** code from the use of the QP/C framework. In other words, the `lib-qpc.lnt` option file allows completely clean "linting" of the application-level code, as long as the application code does not violate any of the MISRA-C:2004 rules.

At the same time, the `lib-qpc.lnt` option file has been very carefully designed **not** to suppress any MISRA-C:2004 rule checking outside the very specific context of the QP/C API. In other words, the `lib-qpc.lnt` option file still supports **100% of the MISRA-C:2004 rule checks** that PC-Lint is capable of performing.

For example, for reasons explained in Section 5.10, QP/C extensively uses function-like macros, which deviates from the MISRA-C:2004 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-qpc.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-qpc.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-qpc.lnt**

```
// general (event.h)
-emacro((960), Q_DIM)          // MISRA04-17.4(req) pointer arithmetic
-emacro(866, Q_DIM)            // Unusual use of 'SYM' in argument to sizeof
-emacro(923, Q_UINT2PTR_CAST) // MISRA04-11.1(req) cast from int to pointer
```

```
-estring(961,                 // MISRA04-19.7(adv) function-like macro
 Q_DIM,
 Q_UINT2PTR_CAST)

// Assertions
-estring(960, l_this_file)    // MISRA04-8.7(req) could use block scope
-estring(961,                 // MISRA04-19.7(adv) function-like macro
 Q_ASSERT,
 . . .)
-function(exit, Q_onAssert)   // give Q_onAssert() the semantics of "exit"

// QEP
-emacro(960,                  // MISRA04-12.10(req) comma operator used
 Q_TRAN,
 Q_SUPER)
-emacro(929,                  // MISRA04-11.4(adv) cast pointer to pointer
 Q_TRAN,
 Q_SUPER,
 Q_STATE_CAST,
 Q_EVENT_CAST)
-emacro(740,                  // MISRA04-1.2(req) & MISRA4-11.2(req) pointer cast
 Q_TRAN,                     // (incompatible indirect types)
 Q_SUPER)
 . . .                       // MISRA04-19.7(adv) function-like macro

// QF
-emacro(950,                  // MISRA04-2.1(req) assembly language
 QF_INT_DISABLE,
 QF_INT_ENABLE,
 QF_CRIT_ENTRY,
 QF_CRIT_EXIT)
 . . .
-emacro(929, Q_NEW)           // MISRA04-11.4(adv) cast from pointer to pointer
-emacro(717,                  // do ... while(0)
 QPSet64_insert,
 QPSet64_remove,
 QPSet64_findMax,
 QTimeEvt_postIn,
 QTimeEvt_postEvery)

// QK
-emacro(950,                  // MISRA04-2.1(req) assembly language
 QK_ISR_ENTRY,
 QK_ISR_EXIT)
 . . .                       // MISRA04-19.7(adv) function-like macro

// QS
-emacro(506, QS_*)            // MISRA04-13.7(req) constant value boolean
                              // MISRA04-14.1(req) no unreachable code
-emacro(774, QS_*)            // MISRA04-13.7(req) 'if' always True
                              // MISRA04-14.1(req) no unreachable code
-emacro(923,                  // MISRA04-11.3(req) cast from pointer to int
 QS_OBJ_,
 QS_FUN_)
-estring(961,                 // MISRA04-19.7(adv) function-like macro
 QS_INIT,
```

```
 . . .)

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

## 2.2 QS Software Tracing and the Spy (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:2004 rules reported in the upcoming MISRA Compliance Matrix do **not** pertain to the production code.

## 2.3 Checking MISRA Compliance of a QP/C Source Code

The directory `%QPC%\ports\lint\` (see Listing 1) contains also the `lin.bat` batch file for "linting" the QP/C source code. The `lin.bat` batch file invokes PC-Lint and stores the lint 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:2-004 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.

## 2.4 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 qpc\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 qpc\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.

## 2.5 Testing Rule Coverage Against the MISRA-C Exemplar Suite

In 2007 the MISRA consortium has released the MISRA-C Exemplar Suite (MES) [MES 07], which provides a very convenient code base for testing the rule coverage. In particular, MES can be statically analyzed with PC-Lint in exactly the same manner as any QP/C application code.

The objective of such tests is to find the MISRA-C rule coverage of various option files. The general idea of a rule coverage test to first perform a baseline analysis of the MES code just with the `au-misra2l.lnt` option file and compare it to the analysis with additional option files. The **differences** in the PC-Lint outputs show clearly which MISRA-C rules are no longer checked.

As shown in Listing 1, the directory `%QPC%\ports\lint\MISRA_Exemplar_Suite_test\` contains lint options and the `lin.bat` file for linting the MES with various options.

> **NOTE:** The MISRA-C Exemplar Suite is copyright by MISRA and cannot be included in the QP/C distribution. You need to download it directly from the MISRA website (after registration)

The directory `MISRA_Exemplar_Suite_test\` contains the PC-Lint output files for the `au-misra2.lnt` option file (`lint_MES_misra2.txt`) and the output when additionally the `lib-qpc.lnt` option file is applied (`lint_MES_qpc.txt`). The differences between these two files demonstrate clearly that no MISRA-C:2004 rules (detectable by PC-Lint) have been lost by applying the `lib-qpc.lnt` option file.

> **NOTE:** It is strongly recommended to repeat the MES test for every option file used for MISRA-C compliance checking.

# 3 MISRA-C:2004 Compliance Matrix

As recommended in Section 4.3.1 of the [MISRA-C:2004] guidelines, this section presents the compliance matrix, which lists each MISRA-C:2004 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:2004 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.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:2004 rule is determined by means of two sources (1) the Gimpel Software matrix [PC-Lint-MISRA-C:2004] and (2) the test against the MISRA Exemplar Suite [MISRA-C:Test Suite 07]. 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. **Description** column contains a short description of the rule, as published in Appendix A of the [MISRA-C:2004] guidelines.

## 3.1 Environment

| Rule No. | PC-Lint | QP/C | QP/C app. | Description |
|---|---|---|---|---|
| **1.1(req)** | ☑ | **A** | | All code shall conform to ISO 9899:1990 Programming languages – C, amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996. |
| **1.2(req)** | ☑ | **A** | (1) | No reliance shall be placed on undefined or unspecified behavior. |
| **1.3(req)** | ☐ | **M** | | Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the language/compilers/assemblers conform. |
| **1.4(req)** | ☐ | n/a | | The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers. |
| **1.5(adv)** | ☐ | **M** | | Floating-point implementations should comply with a defined floating-point standard. |

(1) PC-Lint gives warning 740 (incompatible indirect types, MISRA rule 1.2) for the macro `Q_STATE_CAST`, which really does not cause any undefined behavior. The real deviation occurs for rule 11.1, which is explained in Section **6.1**.

## 3.2 Language Extensions

| Rule No. | PC-Lint | QP/C | QP/C app. | Description |
|---|---|---|---|---|
| **2.1(req)** | ☑ | **A**(1) | (2) | Assembler language shall be encapsulated and isolated. |
| **2.2(req)** | ☑ | **A** | | Source code shall only use `/* ... */` style comments. |
| **2.3(req)** | ☑ | **A** | | The character sequence `/*` shall not be used within a comment. |
| **2.4(adv)** | ☐(3) | **M** | | Sections of code should not be “commented out”. |

(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()`
(2) The option file `lib-qpc.lib` silences the PC-Lint warning 950 for the encapsulated assembler use.
(3) PC-Lint checks for nested comments (rule 2.2), which could be indicative for “commented out” code

## 3.3 Documentation

| Rule No. | PC-Lint | QP/C | QP/C app. | Description |
|---|---|---|---|---|
| **3.1(req)** | ☐ | **M** | | All usage of implementation-defined behavior shall be documented. |
| **3.2(req)** | ☐ | **M**(1) | | The character set and the corresponding encoding shall be documented. |
| **3.3(adv)** | ☐ | **M**(2) | | The implementation of integer division in the chosen compiler should be determined, documented, and taken into account. |
| **3.4(req)** | ☑(3) | **A** | | All uses of the `#pragma` directive shall be documented and explained. |
| **3.5(req)** | ☐ | **M**(4) | | If it is being relied upon, the implementation-defined behavior and packing of bitfields shall be documented. |
| **3.6(req)** | ☑(5) | **A** | | All libraries used in production code shall be written to comply with the provisions of [MISRA-C:2004] guidelines, and shall have been subject to appropriate validation. |

(1) QP/C source code uses only ASCII character set
(2) QP/C does not use integer division or modulo operations anywhere in the code
(3) PC-Lint `au-misra2.lnt` reports all unknown pragmas, except `push_macro` and `pop_macro`
(4) QP/C does not use bit fields anywhere in the code
(5) Requires analysis of the complete application source code, including all libraries

## 3.4 Character sets

| Rule No. | PC-Lint | QP/C | QP/C app. | Description |
|---|---|---|---|---|
| **4.1(req)** | ☑ | **A**(1) | | Only those escape sequences that are defined in the ISO C standard shall be used. |
| **4.2(req)** | ☑ | **A** | | Trigraphs shall not be used. |

(1) QP/C does not use any character escape sequences anywhere in the code

## 3.5 Identifiers

| Rule No. | PC-Lint | QP/C | QP/C app. | Description |
|---|---|---|---|---|
| **5.1(req)** | ☑(1) | **A** | | Identifiers (internal and external) shall not rely on the significance of more than 31 characters. |
| **5.2(req)** | ☑ | **A** | | Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier. |
| **5.3(req)** | ☑(1) | **A** | | A `typedef` name shall be a unique identifier. |
| **5.4(req)** | ☑(1) | **A** | | A tag name shall be a unique identifier. |
| **5.5(adv)** | ☑ | **A** | | No object or function identifier with static storage duration should be reused. |
| **5.6(adv)** | ☑(1) | **A** | | No identifier in one namespace should have the same spelling as an identifier in another namespace, with the exception of structure member and union member names. |
| **5.7(adv)** | ☑(1) | **A** | | No identifier name should be reused. |

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

## 3.6 Types

| Rule No. | PC-Lint | QP/C | QP/C app. | Description |
|---|---|---|---|---|
| **6.1(req)** | ☑(1) | **A**(2) | | Plain `char` type shall be used only for the storage and use of character values. |
| **6.2(req)** | ☑ | **A** | | `signed` and `unsigned char` type shall be used only for the storage and use of numeric values. |
| **6.3(adv)** | ☑ | **A**(3) | | `typedefs` that indicate size and signedness should be used in place of the basic types. |
| **6.4(req)** | ☑ | **A**(4) | | Bitfields shall only be defined to be of type `unsigned int` or `signed int` |
| **6.5(req)** | ☑ | **A**(4) | | Bitfields of signed type shall be at least 2 bits long |

(1) PC-Lint enforces this rule by disallowing all uses of `char` type completely.
(2) QP/C provides special `typedef char_t` for exclusive use as character values
(3) QP/C uses the standard exact-with integer types `stdint.h` (WG14/N843 C99, Section 7.18.1.1)
(4) QP/C does not use bit fields anywhere in the code

## 3.7 Constants

| Rule No. | PC-Lint | QP/C | QP/C app. | Description |
|---|---|---|---|---|
| **7.1(req)** | ☑ | **A** | | Octal constants (other than zero) and octal escape sequences shall not be used. |

## 3.8 Declarations and Definitions

| Rule No. | PC-Lint | QP/C | QP/C app. | Description |
|---|---|---|---|---|
| 8.1(req) | ☑ | A | | Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call. |
| 8.2(req) | ☑ | A | | Whenever an object or function is declared or defined, its type shall be explicitly stated. |
| 8.3(req) | ☑ | A | | For each function parameter, the type given in the declaration and definition shall be identical and the return types shall also be identical. |
| 8.4(req) | ☑ | A | | If objects or functions are declared more than once, their types shall be compatible. |
| 8.5(req) | ☑ | A | | There shall be no definitions of objects or functions in a header file. |
| 8.6(req) | ☑ | A | | Functions shall be declared at file scope. Declarations and definitions |
| 8.7(req) | ☑ | 5.1 | 5.1 | Objects shall be defined at block scope if they are only accessed from within a single function. |
| 8.8(req) | ☑ | A | | An external object or function shall be declared in one and only one file. |
| 8.9(req) | ☑ | A | | An identifier with external linkage shall have exactly one external definition. |
| 8.10(req) | ☑ | A | | All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required. |
| 8.11(req) | ☑ | A | | The `static` storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage. |
| 8.12(req) | ☑ | A | | When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization. |

## 3.9 Initialization

| Rule No. | PC-Lint | QP/C | QP/C app. | Description |
|---|---|---|---|---|
| 9.1(req) | ☑ | A | | All automatic variables shall have been assigned a value before being used. |
| 9.2(req) | ☑ | A | | Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures. |
| 9.3(req) | ☑ | A | | In an enumerator list, the "=" construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized. |

## 3.10 Arithmetic type conversions

| Rule No. | PC-Lint | QP/C | QP/C app. | Description |
|---|---|---|---|---|
| **10.1(req)** | ☑ | **A** | | The value of an expression of integer type shall not be implicitly converted to a different underlying type if:<br>a. it is not a conversion to a wider integer type of the same signedness, or<br>b. the expression is complex, or<br>c. the expression is not constant and is a function argument, or<br>d. the expression is not constant and is a return expression. |
| **10.2(req)** | ☑ | **A** | | The value of an expression of floating type shall not be implicitly converted to a different underlying type if:<br>a. it is not a conversion to a wider floating type, or<br>b. the expression is complex, or<br>c. the expression is a function argument, or<br>d. the expression is a return expression. |
| **10.3(req)** | ☑ | **A** | | The value of a complex expression of integer type shall only be cast to a type that is not wider and of the same signedness as the underlying type of the expression. |
| **10.4(req)** | ☑ | **A** | | The value of a complex expression of floating type shall only be cast to a floating type which is narrower or of the same size. |
| **10.5(req)** | ☑ | **A** | | If the bitwise operators ~ and << are applied to an operand of underlying type unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand. |
| **10.6(req)** | ☑ | **A**(1) | | A "`U`" suffix shall be applied to all constants of `unsigned` type. |

(1) Being strong-type compliant, the QP/C source goes beyond this rule by explicitly casting all constants to the exact-width integer type (e.g., `(uint8_t)1`).

## 3.11 Pointer type conversions

| Rule No. | PC-Lint | QP/C | QP/C app. | Description |
|---|---|---|---|---|
| **11.1(req)** | ☑ | **A** | **6.1** | Conversions shall not be performed between a pointer to a function and any type other than an integral type. |
| **11.2(req)** | ☑ | **A** | | Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to object type, or a pointer to `void`. |
| **11.3(adv)** | ☑ | **5.2**(1) | (2) | A cast should not be performed between a pointer type and an integral type. |
| **11.4(adv)** | ☑(3) | **A**(3) | **6.2** | A cast should not be performed between a pointer to object type and a different pointer to object type. |
| **11.5(req)** | ☑ | **5.3** | | A cast shall not be performed that removes any `const` or `volatile` qualification from the type addressed by a pointer. |

(1) QP/C deviates from this rule only in the macros `QS_OBJ_()` and `QS_FUN_()` in the `Q_SPY` configuration
(2) QP/C provides macro `Q_UNIT2PTR_CAST()`, which could be used for some QP/C **ports** and for **application use**.
(3) PC/Lint reports 11.4 as warning 929 (pointer to ponter cast) for every deviation from 11.5.

## 3.12 Expressions

| Rule No. | PC-Lint | QP/C | QP/C app. | Description |
|---|---|---|---|---|
| **12.1(adv)** | ☑ | **A** | | Limited dependence should be placed on the C operator precedence rules in expressions. |
| **12.2(req)** | ☑ | **A** | | The value of an expression shall be the same under any order of evaluation that the standard permits. |
| **12.3(req)** | ☑ | **A** | | The `sizeof` operator shall not be used on expressions that contain side effects. |
| **12.4(req)** | ☑ | **A** | | The right-hand operand of a logical `&&` or `\|\|` operator shall not contain side effects. |
| **12.5(req)** | ☑ | **A** | | The operands of a logical `&&` or `\|\|` shall be primary expressions. |
| **12.6(adv)** | ☑ | **A** | | The operands of logical operators (`&&`, `\|\|`, and `!`) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (`&&`, `\|\|`, `!`, `=`, `==`, `!=`, and `?:`). |
| **12.7(req)** | ☑ | **A** | | Bitwise operators shall not be applied to operands whose underlying type is signed. |
| **12.8(req)** | ☑ | **5.4**(1) | (1) | 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. |
| **12.9(req)** | ☑ | **A** | | The unary minus operator shall not be applied to an expression whose underlying type is `unsigned`. |
| **12.10(req)** | ☑ | **A** | **6.1** | The comma operator shall not be used. |
| **12.11(adv)** | ☑ | **A** | | Evaluation of constant unsigned integer expressions should not lead to wrap-around. |
| **12.12(req)** | ☑ | **A**(2) | | The underlying bit representations of floating-point values shall not be used. |
| **12.13(adv)** | ☑ | **5.5**(3) | | The increment (`++`) and decrement (`--`) operators should not be mixed with other operators in an expression. |

(1) QP/C deviates from this rule only in the macros `QS_BEGIN_()` and `QS_BEGIN_CRIT_()` in the `Q_SPY` configuration
(2) QP/C does not use floating point expressions anywhere in the code
(3) QP/C deviates from this rule only in the macro `QS_TEC_()` in the `Q_SPY` configuration

## 3.13 Control statement expressions

| Rule No. | PC-Lint | QP/C | QP/C app. | Description |
|---|---|---|---|---|
| 13.1(req) | ☑ | A | | Assignment operators shall not be used in expressions that yield a Boolean value. |
| 13.2(adv) | ☑ | A | | Tests of a value against zero should be made explicit, unless the operand is effectively Boolean. |
| 13.3(req) | ☐ | A | | Floating-point expressions shall not be tested for equality or inequality. |
| 13.4(req) | ☑ | A | | The controlling expression of a `for` statement shall not contain any objects of floating type. |
| 13.5(req) | ☑ | A | | The three expressions of a `for` statement shall be concerned only with loop control. |
| 13.6(req) | ☑ | A | | Numeric variables being used within a `for` loop for iteration counting shall not be modified in the body of the loop. |
| 13.7(req) | ☑ | 5.4[1)] | | Boolean operations whose results are invariant shall not be permitted. |

[1)] QP/C deviates from this rule only in QS macros in the `Q_SPY` configuration

## 3.14 Control flow

| Rule No. | PC-Lint | QP/C | QP/C app. | Description |
|---|---|---|---|---|
| 14.1(req) | ☑ | 5.4[1)] | 5.4[1)] | There shall be no unreachable code. |
| 14.2(req) | ☑ | A | | All non-null statements shall either have at least one side effect however executed, or cause control flow to change. |
| 14.3(req) | ☑ | 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 whitespace character. |
| 14.4(req) | ☑ | A | | The `goto` statement shall not be used. |
| 14.5(req) | ☑ | A | | The `continue` statement shall not be used. |
| 14.6(req) | ☑ | A | | For any iteration statement, there shall be at most one `break` statement used for loop termination. |
| 14.7(req) | ☑ | A | 6.3[2)] | A function shall have a single point of exit at the end of the function. |
| 14.8(req) | ☑ | A | | The statement forming the body of a `switch`, `while`, `do...while`, or `for` statement shall be a compound statement. |
| 14.9(req) | ☑ | A | | An `if` expression construct shall be followed by a compound statement. The `else` keyword shall be followed by either a compound statement or another `if` statement. |
| 14.10(req) | ☑ | A | | All `if ... else if` constructs shall be terminated with an `else` clause. |

[1)] QP/C deviates from this rule only in QS macros in the `Q_SPY` configuration
[2)] QP/C applications might deviate from this rule in the state machine code.

## 3.15 Switch statements

| Rule No. | PC-Lint | QP/C | QP/C app. | Description |
|---|---|---|---|---|
| **15.0(req)** | ☑ | **A** | | The MISRA `switch` syntax shall be used |
| **15.1(req)** | ☑ | **A** | | A `switch` label shall only be used when the most closely-enclosing compound statement is the body of a `switch` statement. |
| **15.2(req)** | ☑ | **A** | **6.3**(1) | An unconditional `break` statement shall terminate every non-empty `switch` clause. |
| **15.3(req)** | ☑ | **A** | **6.3**(1) | The final clause of a `switch` statement shall be the `default` clause. |
| **15.4(req)** | ☑ | **A** | | A `switch` expression shall not represent a value that is effectively Boolean. |
| **15.5(req)** | ☑ | **A** | | Every `switch` statement shall have at least one `case` clause. |

(1) QP/C applications might deviate from this rule in the state machine code

## 3.16 Functions

| Rule No. | PC-Lint | QP/C | QP/C app. | Description |
|---|---|---|---|---|
| **16.1(req)** | ☑ | **A** | | Functions shall not be defined with a variable number of arguments. |
| **16.2(req)** | ☑ | **A** | | Functions shall not call themselves, either directly or indirectly. |
| **16.3(req)** | ☑ | **A** | | Identifiers shall be given for all of the parameters in a function prototype declaration. |
| **16.4(req)** | ☑ | **A** | | The identifiers used in the declaration and definition of a function shall be identical. |
| **16.5(req)** | ☑ | **A** | | Functions with no parameters shall be declared and defined with the parameter list `void`. |
| **16.6(req)** | ☑ | **A** | | The number of arguments passed to a function shall match the number of parameters. |
| **16.7(adv)** | ☑ | **5.6**(1) | | A pointer parameter in a function prototype should be declared as pointer to `const` if the pointer is not used to modify the addressed object. |
| **16.8(req)** | ☑ | **A** | | All exit paths from a function with non-`void` return type shall have an explicit `return` statement with an expression. |
| **16.9(req)** | ☐(2) | **A** | | A function identifier shall only be used with either a preceding `&`, or with a parenthesized parameter list, which may be empty. |
| **16.10(req)** | ☑ | **A** | (3) | If a function returns error information, then that error information shall be tested. |

(1) QP/C deviates from this rule only in certain QP **ports**
(2) Contrary to [PC-Lint MISRA-C:2004], this rule is **not** checked correctly in PC-Lint 9.x. In fact, PC-Lint issues Note 546 “Suspicious use of &” when '&' is actually used in front of a function identifier, which is exactly the opposite to MISRA rule 16.9.
(3) QP/C applies Design by Contract (assertions) instead of returning error codes from its API.

## 3.17 Pointers and arrays

| Rule No. | PC-Lint | QP/C | QP/C app. | Description |
|---|---|---|---|---|
| 17.1(req) | ☑ | A | | Pointer arithmetic shall only be applied to pointers that address an array or array element. |
| 17.2(req) | ☑(1) | A | | Pointer subtraction shall only be applied to pointers that address elements of the same array. |
| 17.3(req) | ☑ | A(2) | | >, >=, <, <= shall not be applied to pointer types except where they point to the same array. |
| 17.4(req) | ☑ | 5.8 | (3) | Array indexing shall be the only allowed form of pointer arithmetic. |
| 17.5(adv) | ☑ | A | | The declaration of objects should contain no more than two levels of pointer indirection. |
| 17.6(adv) | ☑ | 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. |

(1) PC-Lint reports for 17.2 every deviation from 17.3.
(2) QP/C deviates from this rule only in one assertion in `qm_put.c`
(3) PC-Lint reports deviation from this rule for the macro `Q_DIM()`, which is used to calculate the dimension of a 1-dimensional array as follows (file `qevent.h`). However, this seems to be a false-positive.

## 3.18 Structures and unions

| Rule No. | PC-Lint | QP/C | QP/C app. | Description |
|---|---|---|---|---|
| 18.1(req) | ☑ | A | | All `struct`ure and `union` types shall be complete at the end of the translation unit. |
| 18.2(req) | ☐ | M | | An object shall not be assigned to an overlapping object. |
| 18.3(req) | ☐ | M | | An area of memory shall not be used for unrelated purposes. |
| 18.4(req) | ☑ | 5.9(1) | | Unions shall not be used. |

(1) QP/C deviates from this rule only in QS functions `QS_f32` and `QS_f64` in the `Q_SPY` configuration

## 3.19 Preprocessing directives

| Rule No. | PC-Lint | QP/C | QP/C app. | Description |
|---|---|---|---|---|
| 19.1(adv) | ☑ | A | | `#include` statements in a file should only be preceded by other preprocessor directives or comments. |
| 19.2(adv) | ☑ | A | | Non-standard characters should not occur in header file names in `#include` directives. |
| 19.3(req) | ☑ | A | | The `#include` directive shall be followed by either a `<filename>` or `"filename"` sequence. |

| Rule No. | PC-Lint | QP/C | QP/C app. | Description |
|---|---|---|---|---|
| **19.4(req)** | ☑ | **A** | | C macros shall only expand to a braced initializer, a constant, a string literal, a parenthesized expression, a type qualifier, a storage class specifier, or a `do-while`-zero construct. |
| **19.5(req)** | ☑ | **A** | | Macros shall not be `#define`'d or `#undef`'d within a block. |
| **19.6(req)** | ☑ | **A** | | `#undef` shall not be used. Preprocessing directives |
| **19.7(adv)** | ☑ | **5.10** | **5.10** | A function should be used in preference to a function-like macro. |
| **19.8(req)** | ☑ | **A** | | A function-like macro shall not be invoked without all of its arguments. |
| **19.9(req)** | ☑ | **A** | | Arguments to a function-like macro shall not contain tokens that look like preprocessing directives. |
| **19.10(req)** | ☑ | **A**(1) | | 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 `#` or `##`. |
| **19.11(req)** | ☑ | **A**(1) | | All macro identifiers in preprocessor directives shall be defined before use, except in `#ifdef` and `#ifndef` preprocessor directives and the `defined()` operator. |
| **19.12(req)** | ☑ | **A**(1) | | There shall be at most one occurrence of the `#` or `##` preprocessor operators in a single macro definition. Preprocessing directives |
| **19.13(adv)** | ☑ | **A**(1) | | The `#` and `##` preprocessor operators should not be used. |
| **19.14(req)** | ☑ | **A** | | The `defined` preprocessor operator shall only be used in one of the two standard forms. |
| **19.15(req)** | ☑(2) | **A**(3) | | Precautions shall be taken in order to prevent the contents of a header file being included twice. |
| **19.16(req)** | ☑ | **A** | | Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor. |
| **19.17(req)** | ☑ | **A** | | All `#else`, `#elif`, and `#endif` preprocessor directives shall reside in the same file as the `#if` or `#ifdef` directive to which they are related. |

(1) QP/C does not use the `#` or `##` operators anywhere in the code
(2) PC-Lint reports warning 537 for any repeated include file regardless of the the standard `#ifndef xxx_h...#endif` protection used in the header file (which is checked independently by warning 451). The warning 537 appears only in the `Q_SPY` configuration and only for the `qep_port.h` header file.
(3) QP/C uses the standard `#ifndef xxx_h...#endif` protection in all header files.

## 3.20 Standard libraries

| Rule No. | PC-Lint | QP/C | QP/C app. | Description |
|---|---|---|---|---|
| **20.1(req)** | ☑ | **A** | | Reserved identifiers, macros, and functions in the standard library shall not be defined, redefined, or undefined. |
| **20.2(req)** | ☑ | **A** | | The names of Standard Library macros, objects, and functions shall not be reused. |
| **20.3(req)** | ☑ | **A**(1) | | The validity of values passed to library functions shall be checked. |
| **20.4(req)** | ☑ | **A** | | Dynamic heap memory allocation shall not be used. |
| **20.5(req)** | ☑ | **A**(1) | | The error indicator `errno` shall not be used. |
| **20.6(req)** | ☑ | **A**(1) | | The macro `offsetof` in the `stddef.h` library shall not be used. |
| **20.7(req)** | ☑ | **A**(1) | | The `setjmp` macro and the `longjmp` function shall not be used. |
| **20.8(req)** | ☑ | **A**(1) | | The signal handling facilities of `signal.h` shall not be used. |
| **20.9(req)** | ☑ | **A**(1) | | The input/output library `stdio.h` shall not be used in production code. |
| **20.10(req)** | ☑ | **A**(1) | | The functions `atof`, `atoi`, and `atol` from the library `stdlib.h` shall not be used. |
| **20.11(req)** | ☑ | **A**(1) | | The functions `abort`, `exit`, `getenv`, and system from the library `stdlib.h` shall not be used. |
| **20.12(req)** | ☑ | **A**(1) | | The time handling functions of `time.h` shall not be used. |

(1) Except `stdint.h` used in QP/C ports, QP/C code does not rely in any way on any standard C libraries.

## 3.21 Run-time libraries

| Rule No. | PC-Lint | QP/C | QP/C app. | Description |
|---|---|---|---|---|
| **21.1(req)** | ☑ | **A** | | Minimization of runtime failures shall be ensured by the use of at least one of:<br>a. static analysis tools/techniques<br>b. dynamic analysis tools/techniques<br>c. explicit coding of checks to handle runtime fault |

# 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 pont 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 is 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:2004 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:2004 rules in the **QP/C source code**.

## 5.1 Rule 8.7(req)

**Objects shall be defined at block scope if they are only accessed from within a single function.**

Deviation from this rule occurs occasionally in using QP assertion macros `Q_DEFINE_THIS_FILE` and `Q_DEFINE_THIS_MODULE` (the "`qassert.h`" file). To save memory, the file name string is defined only once in static variable `l_this_file[]` and then subsequently reused in every assertion.

However, if only one assertion happens to be used in a given file scope, the variable `l_this_file[]` could be demoted to block scope. But this would break again if another assertion would be added at a later time. Therefore, for the sake of maintainability the deviation is allowed in this particular context only.

## 5.2 Rule 11.3(req)

**A cast should not be performed between a pointer type and 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.

## 5.3 Rule 11.5(req)

**A cast shall not be performed that removes any *const* or *volatile* qualification from the type addressed by a pointer**

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 ensuring that a given event is indeed dynamic (by testing the `QEvent.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.

## 5.4 Rule 12.8(req), 13.7(req), and 14.1(req)

**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**

**Boolean operations whose results are invariant shall not be permitted**

**There shall be no unreachable code**

Deviation from these rules occurs only in the QS software tracing instrumentation (**not** in production code) and is related to QS filters. Certain QS trace records don't use all the filters, which manifests itself as an excessive shift. The deviation is allowed only in the context of the macros `QS_BEGIN_` and `QS_BEGIN_NOCRIT_`.

## 5.5 Rule 12.13(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).

## 5.6 Rule 16.7(adv)

**A pointer parameter in a function prototype should be declared as pointer to *const* if the pointer is not used to modify the addressed object.**

Deviation from this rule occurs only 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()` and `Qactive_stop()` cannot declare the stack parameter as `const`, so in this particular context the deviation is allowed.

## 5.7 Rule 17.3(req)

**>, >=, <, <= shall not be applied to pointer types 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.c`). 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 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.

## 5.8 Rule 17.4(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 blocks 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 17.4.

The deviation form rule 17.4 is encapsulated in the QP/C internal macro `QF_PTR_AT_()`, and this context is allowed to deviate per this procedure.

> **NOTE:** PC-Lint also reports deviation from this rule for the macro `Q_DIM()`, which is used to calculate the dimension of a 1-dimensional array as follows (file `qevent.h`). However, this seems to be a false-positive.

## 5.9 Rule 18.4(req)

**Unions shall not be used.**

Deviation from these rules 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`.

## 5.10 Rule 19.7(adv)

**A function should be used in preference to a function-like macro.**

QP/C uses function-like macro extensively, because it must be widely portable yet efficient, and not all embedded cross-compilers support the "inline" function semantics. All function-like macros deviating from the rule 19.7 are listed explicitly in the option files, so any other macros violating the rule will be reported.

# 6 Deviation Procedures for Application-Level Code

This section documents deviations from MISRA-C:2004 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.

## 6.1 Rule 11.1(req), and 12.10(req)

**11.1(req): Conversions shall not be performed between a pointer to a function and any type other than an integral type**

**12.10(req): The comma operator shall not be used**

The QP/C applications deviate from rules 11.1, and 12.10 by using the QP/C macros `Q_STATE_CAST()`, `Q_TRAN()`, and `Q_SUPER()`, which are needed for coding hierarchical state machines in QP/C.

The macro `Q_STATE_CAST()` deviates from MISRA-C rule 11.1, because it performs cast to `(QStateHandler)`. Here is the definition of the `Q_STATE_CAST()` macro (file `qep.h`):

```
#define Q_STATE_CAST(handler_)  ((QStateHandler)(handler_))
```

In the QP/C application code the macro `Q_STATE_CAST()` is used only to cast from pointers to state-handler functions, which are all compatible with `QStateHandler`. For example, the constructor of a custom active object must call the constructor of the base class `QActive_ctor()` with the pointer to the initial state handler function, like this:

```
QActive_ctor(&me->super, Q_STATE_CAST(&Philo_initial));
```

The state-handler functions are **compatible**, because they have almost the same signatures and differ only in the type of the first argument "`me`". However, the "`me`" argument is **derived** from the `QHsm` base class (in the sense described in the Recipe "Simple Encapsulation and Inheritance in C" [QL-OOPC 02]), but the C compiler does not "know" about this relationship—hence the cast is necessary.

The need to deviate from the rule 11.1 is a consequence of using function pointers in conjunction with "inheritance of structures", which are both fundamental to the QP/C framework. This, very particular, deviation from rule 11.1 is safe and is allowed only in the context of state-handler functions, which are related.

Additionally, macros `Q_TRAN()` and `Q_SUPER()` deviate from the rule 12.10 (comma operator use). This deviation is needed for encapsulation of state-machine concepts (transition and superstate, respectively).

## 6.2 Rule 11.4(req)

**11.4(req): A cast should not be performed between a pointer to object type and a different pointer to object type.**

The QP/C applications deviate from rule 11.4 because of downcasting the generic event pointer (`QEvent 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 14.7(req), 15.2(req), and 15.3(req)

**14.7(req): A function shall have a single point of exit at the end of the function.**

**15.2(req): An unconditional *break* statement shall terminate every non-empty *switch* clause.**

**15.3(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 14.7, 15.2, and 5.3. However, it is also possible to avoid all these deviations, in exchange for a slight change in the UML semantics of guard processing, 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 7. The following Listing 4 shows an example of MISRA-compliant state handler function. The explanation section immediately following the listing highlights the important points.

**Listing 4: MISRA-compliant state handler implementation**

```
    static QState Philo_hungry(Philo *me, QEvent const *e) {
(1)     QState ret;
        switch (e->sig) {
            case Q_ENTRY_SIG: {
                TableEvt *pe = Q_NEW(TableEvt, HUNGRY_SIG);
                pe->philoNum = PHILO_ID(me);
                QACTIVE_POST(AO_Table, &pe->super, me);
(2)             ret = Q_HANDLED();
(3)             break;
            }
            case EAT_SIG: {
(4)             if (Q_EVENT_CAST(TableEvt)->philoNum == PHILO_ID(me)) {
(5)                 ret = Q_TRAN(&Philo_eating);
                }
(6)             else {
(7)                 ret = Q_HANDLED();
                }
(8)             break;
            }
(9)         default: {
(10)            ret = Q_SUPER(&QHsm_top);
(11)            break;
            }
        }
(12)    return ret;
    }
```

(1) The automatic variable `ret` will store the return value. Please note that the `ret` variable is not initialized.

> **NOTE:** The `ret` 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 `Pholo_eating` needs to be taken.

(6,7) When the guard condition evaluates to FALSE, the return value is set from the macro `Q_HANDLED()`.

**NOTE:** The `else`-branch of the guard condition represents an internal transition, which must be added to satisfy the MISRA-C `switch`-statement structure. The problem arises if such additional internal transition is unintended, and the real intend of the state machine designer is to treat the event as unhandled and propagate it to the higher levels of state nesting when the guard evaluates to FALSE. The latter behavior would correspond to the UML semantics, but the problem is that this semantics cannot be coded with a MISRA-compliant `switch` statement.

(8) According to the recommended MISRA-C `switch` statement structure, the case is terminated with a `break`

(9) According to the recommended MISRA-C `switch` statement structure, the `default`-clause is the final clause of the `switch` statement

(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.

**NOTE:** The QM graphical modeling tool currently generates traditional state machine code, which deviates from the MISRA-C rules 14.7, 15.2, and 5.3. However, the future releases of QM will have an option to generate MISRA-compliant code, as described in this section.

# 7 Summary

The QP/C framework complies with most of the MISRA-C:2004 rules and all remaining deviations are carefully insulated and encapsulated into very specific contexts. The framework goes even beyond MISRA, by complying with string 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

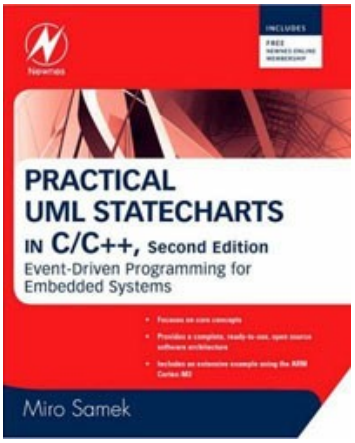
| Document | Location |
|---|---|
| [MISRA-C:2004] MISRA-C:2004 Guidelines for the Use of the C language in Critical Systems, MISRA, October 2004, ISBN: 978-0-9524156-2-6 paperback ISBN: 978-0-9524156-4-0 PDF | Available for purchase from MISRA website http://www.misra.org.uk |
| [MISRA-C:1998] Guidelines for the Use of the C Language in Vehicle Based Software, April 1998, October 2002. ISBN 978-0-9524156-6-5 | Available for purchase from MISRA website http://www.misra.org.uk |
| [MES 07] MISRA-C Exemplar Suite, MISRA C Working Group, 2007 | Available for downlaod from MISRA website (requires registration) http://www.misra.org.uk |
| [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:2004] PC-Lint/FlexeLint Support for MISRA C 2004, | Available for downlaod from Gimpel website at http://www.gimpel.com/html/misra.htm |
| [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 |
| [QL OOPC 02] Recipe: Simple Encapsulation and Inheritance in C, Quantum Leaps, LLC 2002 | http://www.state-machine.com/resources/Recipe_SimpleOOC.pdf |
| [Hatton 02] MISRA-C1 Exemplar Test Suite, 2002 | http://www.leshatton.org/MISRA_CNF_1002.html |

# 9 Contact Information

**Quantum Leaps, LLC**
103 Cobble Ridge Drive
Chapel Hill, NC 27516
USA

+1 866 450 LEAP (toll free, USA only)
+1 919 869-2998 (FAX)

e-mail: info@quantum-leaps.com
WEB : http://www.quantum-leaps.com
http://www.state-machine.com




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