



Quantum[®]Leaps
innovating embedded systems

MISRA

Application Note

QP-nano[™] MISRA-C:2004 Compliance Matrix

Document Revision F
December 2014

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:2004	1
1.2 About QP-nano™	1
2 Checking MISRA Compliance with PC-Lint/FlexeLint	2
2.1 Structure of PC-Lint Options for QP-nano	2
2.2 Checking MISRA Compliance of a QP-nano Source Code	6
2.3 Checking MISRA Compliance of a QP-nano Application Code	6
2.4 Testing Rule Coverage Against the MISRA-C Exemplar Suite	7
3 MISRA-C:2004 Compliance Matrix	8
3.1 Environment	9
3.2 Language Extensions	9
3.3 Documentation	10
3.4 Character sets	10
3.5 Identifiers	11
3.6 Types	11
3.7 Constants	12
3.8 Declarations and Definitions	12
3.9 Initialization	13
3.10 Arithmetic type conversions	13
3.11 Pointer type conversions	14
3.12 Expressions	15
3.13 Control statement expressions	16
3.14 Control flow	16
3.15 Switch statements	17
3.16 Functions	17
3.17 Pointers and arrays	18
3.18 Structures and unions	18
3.19 Preprocessing directives	19
3.20 Standard libraries	20
3.21 Run-time libraries	20
4 Beyond MISRA: Compliance with Additional Rules and Standards	21
4.1 Strong Type Checking	21
4.2 Quantum Leaps C/C++ Coding Standard	21
5 Deviation Procedures for QP-nano Source Code	22
5.1 Rule 8.7(req)	22
5.2 Rule 8.12(req)	22
5.3 Rule 17.4(req)	22
5.4 Rule 18.4(req)	23
5.5 Rule 19.7(adv)	23
6 Deviation Procedures for Application-Level Code	24
6.1 Rule 11.4(adv)	24
6.2 Rule 11.1(req), and 12.10(req)	25
6.3 Rule 14.7(req), 15.2(req), and 15.3(req)	25
7 Summary	27
8 Related Documents and References	28
9 Contact Information	29

1 Introduction

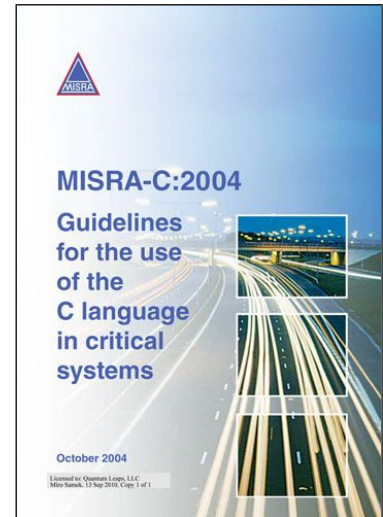
This Application Note describes the compliance of the QP-nano™ state machine framework version 5.1.0 or higher and the application code based on this framework with the Motor Industry Software Reliability Association (MISRA) Guidelines for the use of the C Language in Critical Systems [MISRA-C:2004]. 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 its 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.

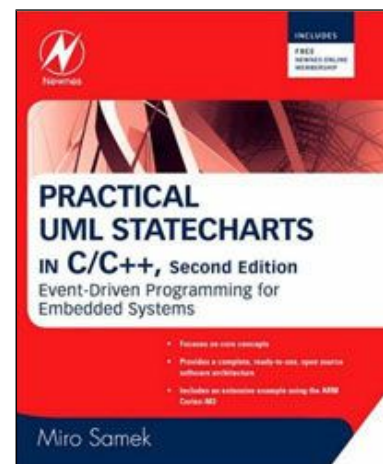


NOTE: MISRA and MISRA C are registered trademarks of [MIRA Ltd.](http://www.mira.com), held on behalf of the [MISRA Consortium](http://www.misra.org.uk).

1.2 About QP-nano™

QP-nano™ is an ultra-lightweight, open source, active object framework for low-end 8-, and 16-bit microcontrollers and SoCs. The QP-nano framework 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-nano has been described in Chapter 12 of 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-nano, 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 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** 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 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-nano

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-nano 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: Except for one instance in the `qfn.h` header file (see Section ??), the QP-nano source code is **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-nano 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-nano, 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-nano are divided into

two groups, located in directories `qp\include\` and `qp\lint\`. These two groups are for analyzing QP-nano applications and QP-nano source code, respectively.

Listing 1: PC-Lint options for “linting” QP-nano applications (`qp\include\`) and “lining” QP-nano source code itself (`qp\lint\`).

```

%QPN%\
|
|--include\
| |--au-ds.lnt
| |--au-misra2.lnt
| |--lib-qp.lnt
| |--std.lnt
| |--qassert.h
| |--qepn.h
| |--. . .
| |
| |--lint\
| | |--MISRA_Exemplar_Suite_test\
| | | |--lin.bat
| | | |--options.lnt
| | | |--. . .
| | |
| | |--lin.bat
| | |--options.lnt
| | |--lint_gen.txt
| | |--qp_port.h
| | |--stdint.h
  
```

- QP-nano Root Directory (environment variable QPN)
- QP-nano platform-independent includes
- Dan Saks recommendations
- Main PC-Lint MISRA-C:2004 compliance options
- PC-Lint options for QP-nano applications
- Standard PC-Lint settings recommended by Quantum Leaps
- QP-nano header file
- QP-nano header file
- . . .
- QP-nano “port” to PC-Lint
- MISRA Exemplar Suite rule coverage test
- Batch file to invoke PC-Lint to run analysis of MES
- PC-Lint options for “linting” MES
- . . .
- Batch file to invoke PC-Lint to analyse QP-nano code
- PC-Lint options for “linting” QP-nano source code
- PC-Lint output for QP-nano
- QP-nano “port” to a generic ANSI C compiler
- Standard exact-width integers for an ANSI C compiler

NOTE: This Application Note assumes that the baseline distribution of the QP-nano framework has been downloaded and installed and that the environment variable `QPN` has been defined to point to the QP-nano installation directory.

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

Note that the `%QPN%\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 `%QPN%\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-qp.lnt` option file

The most important file for “linting” QP-nano applications is the `lib-qp.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-nano framework. In other words, the `lib-qp.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-qp.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-nano API. In other words, the `lib-qp.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 5.5, QP-nano 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-qp.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-qp.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-qp.lnt`

```
// general
-emacro((960), Q_DIM) // MISRA04-17.4(req) pointer arithmetic
-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)
-epsilon(866, Q_DIM) // Unusual use of 'SYM' in argument to sizeof

// 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,
  Q_ASSERT_ID,
  . . .
  Q_ASSERT_COMPILE,
  Q_DEFINE_THIS_MODULE)
-epsilon(522, // Highest operation lacks side-effects
  Q_ASSERT,
  Q_ERROR,
  Q_ALLEGE)
-function(exit, Q_onAssert) // give Q_onAssert() the semantics of "exit"
-estring(752, Q_assert_compile) // not referenced

// QEP-nano
-epsilon(740, // MISRA04-1.2(req) incompatible indirect types
  Q_SIG,
  Q_PAR,
  Q_STATE_CAST,
  Q_TRAN,
  Q_SUPER)
-epsilon(960, // MISRA04-12.10(req) comma operator used
  Q_TRAN,
  Q_SUPER)
-epsilon(929, // MISRA04-11.4(adv) cast pointer to pointer
  Q_SIG,
  Q_PAR,
  Q_STATE_CAST,
  Q_TRAN,
  Q_SUPER,
  Q_EVENT_CAST)
-epsilon(960, // MISRA04-17.4(req), pointer arithmetic
  QEP_ACT_PTR_INC_)
-epsilon(960, Q_STATE_CAST) // MISRA04-11.1(req) Conversions involving
// fun. pointers must be to/from integral types
-estring(960, QAttrTag) // MISRA04-18.4(req) declaration of union type
-estring(961, // MISRA04-19.7(adv) function-like macro
  Q_SIG,
  Q_PAR,
  Q_ROM_BYTE,
  Q_IGNORED,
  Q_HANDLED,
  Q_TRAN,
  Q_SUPER,
  Q_STATE_CAST,
  Q_ACTION_CAST,
  QEP_ACT_PTR_INC_,
  QHsm_state,
  QP_getVersion)
```

```
// QF-nano
-emacs(740, QF_ACTIVE_CAST) // MISRA04-1.2(req) incompatible indirect types
-emacs(826, QF_ACTIVE_CAST) // suspicious pointer-to-pointer conversion
-emacs(950, // MISRA04-2.1(req) assembly language
  QF_INT_DISABLE,
  QF_INT_ENABLE,
  QF_ISR_DISABLE,
  QF_ISR_ENABLE)
-emacs(929, // MISRA04-11.4(adv) cast pointer to pointer
  QF_ACTIVE_CAST)
-estring(961, // MISRA04-19.7(adv) function-like macro
  QF_INT_DISABLE,
  . . .
  QACTIVE_POST,
  QACTIVE_POST_X,
  QACTIVE_POST_ISR,
  QACTIVE_POST_X_ISR,
  QActive_post,
  QActive_postISR)

// QK-nano
-emacs(950, // MISRA04-2.1(req) assembly language
  QK_ISR_ENTRY,
  QK_ISR_EXIT)
-estring(961, // MISRA04-19.7(adv) function-like macro
  QK_ISR_ENTRY,
  QK_ISR_EXIT,
  QK_SCHEDULE_)

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

2.2 Checking MISRA Compliance of a QP-nano Source Code

The directory %QPN%\lint\ (see [Listing 1](#)) contains also the `lin.bat` batch file for “linting” the QP-nano 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 file `lint_qpn.txt`.

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.

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-nano 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.

2.3 Checking MISRA Compliance of a QP-nano Application Code

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

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

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 PELICAN 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.

2.4 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-nano 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 is to first perform a baseline analysis of the MES code just with the `au-misra21.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 `%QPN%\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-nano 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-qpn.lnt` option file is applied (`lint_MES_qpn.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-qpn.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-nano** column lists the compliance status of the QP-nano source code. Letters **A** or **M** in this column mean that the QP-nano 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-nano Source Code](#), which describes in detail the nature and particular circumstances of the deviation.
4. **QP-nano app.** column lists the deviations of the QP-nano **application-level** code imposed by the QP-nano framework. No entry in this column indicates that QP-nano 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-nano 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-nano **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:2004] guidelines.

NOTE: The use of the MISRA-C:2004 rule headlines in this document has been licensed to Quantum Leaps by [MIRA Ltd](#).

3.1 Environment

Rule No.	PC-Lint	QP-nano	QP-nano app.	Rule Headline
1.1(req)	<input checked="" type="checkbox"/>	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)	<input checked="" type="checkbox"/>	A	(1)	No reliance shall be placed on undefined or unspecified behavior.
1.3(req)	<input type="checkbox"/>	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/compiler/assemblers conform.
1.4(req)	<input type="checkbox"/>	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)	<input type="checkbox"/>	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 macros `Q_SIG`, `Q_PAR`, and `QF_ACTIVE_CAST` as well as `Q_TRAN`, `Q_SUPER`, which really do not cause any undefined behavior. The real deviation occurs for rules 11.4 and 11.1, which is explained in Sections 6.1 and 6.2, respectively.

3.2 Language Extensions

Rule No.	PC-Lint	QP-nano	QP-nano app.	Rule Headline
2.1(req)	<input checked="" type="checkbox"/>	A ⁽¹⁾	(2)	Assembler language shall be encapsulated and isolated.
2.2(req)	<input checked="" type="checkbox"/>	A		Source code shall only use <code>/* . . . */</code> style comments.
2.3(req)	<input checked="" type="checkbox"/>	A		The character sequence <code>/*</code> shall not be used within a comment.
2.4(adv)	<input type="checkbox"/> ⁽³⁾	M		Sections of code should not be “commented out”.

(1) QP-nano encapsulates and isolates potential use of assembler language in the macros `QF_INT_ENABLE()`, `QF_INT_DISABLE()`, `QF_ISR_DISABLE()`, `QF_ISR_ENABLE()`, `QK_ISR_ENTRY()`, and `QK_ISR_EXIT()`

(2) The option file `lib-qp.n.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-nano	QP-nano app.	Rule Headline
3.1(req)	<input type="checkbox"/>	M		All usage of implementation-defined behavior shall be documented.
3.2(req)	<input type="checkbox"/>	M ⁽¹⁾		The character set and the corresponding encoding shall be documented.
3.3(adv)	<input type="checkbox"/>	M ⁽²⁾		The implementation of integer division in the chosen compiler should be determined, documented, and taken into account.
3.4(req)	<input checked="" type="checkbox"/> ⁽³⁾	A		All uses of the <code>#pragma</code> directive shall be documented and explained.
3.5(req)	<input type="checkbox"/>	M ⁽⁴⁾		If it is being relied upon, the implementation-defined behavior and packing of bitfields shall be documented.
3.6(req)	<input checked="" type="checkbox"/> ⁽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-nano source code uses only ASCII character set

⁽²⁾ QP-nano 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-nano 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-nano	QP-nano app.	Rule Headline
4.1(req)	<input checked="" type="checkbox"/>	A ⁽¹⁾		Only those escape sequences that are defined in the ISO C standard shall be used.
4.2(req)	<input checked="" type="checkbox"/>	A		Trigraphs shall not be used.

⁽¹⁾ QP-nano does not use any character escape sequences anywhere in the code

3.5 Identifiers

Rule No.	PC-Lint	QP-nano	QP-nano app.	Rule Headline
5.1(req)	<input checked="" type="checkbox"/> ⁽¹⁾	A		Identifiers (internal and external) shall not rely on the significance of more than 31 characters.
5.2(req)	<input checked="" type="checkbox"/>	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)	<input checked="" type="checkbox"/> ⁽¹⁾	A		A <code>typedef</code> name shall be a unique identifier.
5.4(req)	<input checked="" type="checkbox"/> ⁽¹⁾	A		A tag name shall be a unique identifier.
5.5(adv)	<input checked="" type="checkbox"/>	A		No object or function identifier with static storage duration should be reused.
5.6(adv)	<input checked="" type="checkbox"/> ⁽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)	<input checked="" type="checkbox"/> ⁽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-nano	QP-nano app.	Rule Headline
6.1(req)	<input checked="" type="checkbox"/> ⁽¹⁾	A ⁽²⁾		Plain <code>char</code> type shall be used only for the storage and use of character values.
6.2(req)	<input checked="" type="checkbox"/>	A		<code>signed</code> and <code>unsigned char</code> type shall be used only for the storage and use of numeric values.
6.3(adv)	<input checked="" type="checkbox"/>	A ⁽³⁾		<code>typedefs</code> that indicate size and signedness should be used in place of the basic types.
6.4(req)	<input checked="" type="checkbox"/>	A ⁽⁴⁾		Bitfields shall only be defined to be of type <code>unsigned int</code> or <code>signed int</code>
6.5(req)	<input checked="" type="checkbox"/>	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-nano provides special `typedef char_t` for exclusive use as character values

⁽³⁾ QP-nano uses the standard exact-width integer types `stdint.h` (WG14/N843 C99, Section 7.18.1.1)

⁽⁴⁾ QP-nano does not use bit fields anywhere in the code

3.7 Constants

Rule No.	PC-Lint	QP-nano	QP-nano app.	Rule Headline
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-nano	QP-nano app.	Rule Headline
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 <code>static</code> storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
8.12(req)	☑	5.2	(1)	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.

⁽¹⁾ Instead of specifying the size of the `QF_active[]` array explicitly in the QP-nano framework, all QP-nano *applications* check the size of this array by a compile-time assertion. This provides actually a stronger protection against incorrect `QF_active[]` initialization.

3.9 Initialization

Rule No.	PC-Lint	QP-nano	QP-nano app.	Rule Headline
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-nano	QP-nano app.	Rule Headline
10.1(req)	☑	A		The value of an expression of integer type shall not be implicitly converted to a different underlying type if: a. it is not a conversion to a wider integer type of the same signedness, or b. the expression is complex, or c. the expression is not constant and is a function argument, or 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: a. it is not a conversion to a wider floating type, or b. the expression is complex, or c. the expression is a function argument, or 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-nano 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-nano	QP-nano app.	Rule Headline
11.1(req)	☑	A	6.2	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 <code>void</code> .
11.3(adv)	☑	A	(1)	A cast should not be performed between a pointer type and an integral type.
11.4(adv)	☑ ⁽²⁾	A	6.1	A cast should not be performed between a pointer to object type and a different pointer to object type.
11.5(req)	☑	A		A cast shall not be performed that removes any <code>const</code> or <code>volatile</code> qualification from the type addressed by a pointer.

⁽¹⁾ QP-nano provides macro `Q_UNIT2PTR_CAST()`, which could be used for some QP-nano **ports** and for **application use**.

⁽²⁾ PC-Lint reports 11.4 as warning 929 (pointer to pointer cast) for every deviation from 11.5.

3.12 Expressions

Rule No.	PC-Lint	QP-nano	QP-nano app.	Rule Headline
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 <code>sizeof</code> operator shall not be used on expressions that contain side effects.
12.4(req)	☑	A		The right-hand operand of a logical <code>&&</code> or <code> </code> operator shall not contain side effects.
12.5(req)	☑	A		The operands of a logical <code>&&</code> or <code> </code> shall be primary expressions.
12.6(adv)	☑	A		The operands of logical operators (<code>&&</code> , <code> </code> , and <code>!</code>) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (<code>&&</code> , <code> </code> , <code>!</code> , <code>=</code> , <code>==</code> , <code>!=</code> , and <code>?:</code>).
12.7(req)	☑	A		Bitwise operators shall not be applied to operands whose underlying type is signed.
12.8(req)	☑	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.
12.9(req)	☑	A		The unary minus operator shall not be applied to an expression whose underlying type is <code>unsigned</code> .
12.10(req)	☑	A	6.2	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 ⁽¹⁾		The underlying bit representations of floating-point values shall not be used.
12.13(adv)	☑	A		The increment (<code>++</code>) and decrement (<code>--</code>) operators should not be mixed with other operators in an expression.

⁽¹⁾ QP-nano does not use floating point expressions anywhere in the code

3.13 Control statement expressions

Rule No.	PC-Lint	QP-nano	QP-nano app.	Rule Headline
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 <code>for</code> statement shall not contain any objects of floating type.
13.5(req)	☑	A		The three expressions of a <code>for</code> statement shall be concerned only with loop control.
13.6(req)	☑	A		Numeric variables being used within a <code>for</code> loop for iteration counting shall not be modified in the body of the loop.
13.7(req)	☑	A		Boolean operations whose results are invariant shall not be permitted.

3.14 Control flow

Rule No.	PC-Lint	QP-nano	QP-nano app.	Rule Headline
14.1(req)	☑	A	A	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 <code>goto</code> statement shall not be used.
14.5(req)	☑	A		The <code>continue</code> statement shall not be used.
14.6(req)	☑	A		For any iteration statement, there shall be at most one <code>break</code> statement used for loop termination.
14.7(req)	☑	A	6.3 ⁽¹⁾	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 <code>switch</code> , <code>while</code> , <code>do...while</code> , or <code>for</code> statement shall be a compound statement.
14.9(req)	☑	A		An <code>if</code> expression 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.
14.10(req)	☑	A		All <code>if...else if</code> constructs shall be terminated with an <code>else</code> clause.

⁽¹⁾ QP-nano applications might deviate from this rule in the state machine code

3.15 Switch statements

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

⁽¹⁾ QP-nano applications might deviate from this rule in the state machine code

3.16 Functions

Rule No.	PC-Lint	QP-nano	QP-nano app.	Rule Headline
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 <code>void</code> .
16.6(req)	☑	A		The number of arguments passed to a function shall match the number of parameters.
16.7(adv)	☑	A		A pointer parameter in a function prototype should be declared as pointer to <code>const</code> if the pointer is not used to modify the addressed object.
16.8(req)	☑	A		All exit paths from a function with non- <code>void</code> return type shall have an explicit <code>return</code> statement with an expression.
16.9(req)	☐ ⁽¹⁾	A		A function identifier shall only be used with either a preceding <code>&</code> , or with a parenthesized parameter list, which may be empty.
16.10(req)	☑	A	⁽²⁾	If a function returns error information, then that error information shall be tested.

⁽¹⁾ 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.

⁽²⁾ QP-nano applies Design by Contract (assertions) instead of returning error codes from its API.

3.17 Pointers and arrays

Rule No.	PC-Lint	QP-nano	QP-nano app.	Description
17.1(req)	<input checked="" type="checkbox"/>	A		Pointer arithmetic shall only be applied to pointers that address an array or array element.
17.2(req)	<input checked="" type="checkbox"/> ⁽¹⁾	A		Pointer subtraction shall only be applied to pointers that address elements of the same array.
17.3(req)	<input checked="" type="checkbox"/>	A		>, >=, <, <= shall not be applied to pointer types except where they point to the same array.
17.4(req)	<input checked="" type="checkbox"/>	5.3	⁽³⁾	Array indexing shall be the only allowed form of pointer arithmetic.
17.5(adv)	<input checked="" type="checkbox"/>	A		The declaration of objects should contain no more than two levels of pointer indirection.
17.6(adv)	<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.

⁽¹⁾PC-Lint reports for 17.2 every deviation from 17.3.

⁽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 `qepn.h`). However, this seems to be a false-positive.

3.18 Structures and unions

Rule No.	PC-Lint	QP-nano	QP-nano app.	Rule Headline
18.1(req)	<input checked="" type="checkbox"/>	A		All <code>structure</code> and <code>union</code> types shall be complete at the end of the translation unit.
18.2(req)	<input type="checkbox"/>	M		An object shall not be assigned to an overlapping object.
18.3(req)	<input type="checkbox"/>	M		An area of memory shall not be used for unrelated purposes.
18.4(req)	<input checked="" type="checkbox"/>	5.4 ⁽¹⁾		Unions shall not be used.

⁽¹⁾QP-nano deviates from this rule in the `QMAttr` type

3.19 Preprocessing directives

Rule No.	PC-Lint	QP-nano	QP-nano app.	Rule Headline
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.
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)	☑	5.5	5.5	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-nano 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).

⁽³⁾ QP-nano uses the standard #ifndef xxx_h...#endif protection in all header files.

3.20 Standard libraries

Rule No.	PC-Lint	QP-nano	QP-nano app.	Rule Headline
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 <code>errno</code> shall not be used.
20.6(req)	☑	A ⁽¹⁾		The macro <code>offsetof</code> in the <code>stddef.h</code> library shall not be used.
20.7(req)	☑	A ⁽¹⁾		The <code>setjmp</code> macro and the <code>longjmp</code> function shall not be used.
20.8(req)	☑	A ⁽¹⁾		The signal handling facilities of <code>signal.h</code> shall not be used.
20.9(req)	☑	A ⁽¹⁾		The input/output library <code>stdio.h</code> shall not be used in production code.
20.10(req)	☑	A ⁽¹⁾		The functions <code>atof</code> , <code>atoi</code> , and <code>atol</code> from the library <code>stdlib.h</code> shall not be used.
20.11(req)	☑	A ⁽¹⁾		The functions <code>abort</code> , <code>exit</code> , <code>getenv</code> , and <code>system</code> from the library <code>stdlib.h</code> shall not be used.
20.12(req)	☑	A ⁽¹⁾		The time handling functions of <code>time.h</code> shall not be used.

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

3.21 Run-time libraries

Rule No.	PC-Lint	QP-nano	QP-nano app.	Rule Headline
21.1(req)	☑	A		Minimization of runtime 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 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 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-nano 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-nano 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-nano code strictly follows to the Quantum Leaps C/C++ Coding Standard [QL-Code 11].

5 Deviation Procedures for QP-nano Source Code

This section documents deviations from MISRA-C:2004 rules in the **QP-nano source code**.

NOTE: The use of the MISRA-C:2004 rule headlines in this document has been licensed to Quantum Leaps by [MIRA Ltd.](#)

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-nano 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 8.12(req)

When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.

Deviation from this rule occurs for the QP-nano constant array `QF_active[]`, which is intentionally declared without the size specification, which is special, because it is `const` and allocated in ROM.

Instead of declaring the size of this array in the QP-nano framework, all QP-nano applications use a **compile-time assertion** to ensure that the explicit initialization of the `QF_active[]` array indeed matches the `QF_MAX_ACTIVE` macro specified in the `qep_port.h` header file. In this case, an explicit size specification of `QF_active[]` array would prevent any check of the size, even if the explicit initialization would not match the `QF_MAX_ACTIVE` macro.

NOTE: The reporting of the deviation from the rule 8.12 is suppressed by a PC-Lint comment inside the `qfn.h` header file. This is the only instance in the QP-nano source code, where a PC-Lint comment is used.

5.3 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-nano framework with respect to memory allocation. In QP-nano, event queue buffers are 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 from rule 17.4 is encapsulated in the QP-nano internal macro `QF_ROM_QUEUE_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 `qepn.h`). However, this seems to be a false-positive.

5.4 Rule 18.4(req)

Unions shall not be used.

Deviation from this rule occurs in the QEP-nano component in the data type `QMAttr`, 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-nano event processor code and is never accessed in the application-level code. The justification of deviating from the rule 18.4 are significant savings in RAM for every state machine in the system as well as backwards-compatibility with the classes `QHsm` and `QFsm`.

5.5 Rule 19.7(adv)

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

QP-nano 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-nano framework design or implementation. This section also describes workarounds to avoid some deviations.

NOTE: The use of the MISRA-C:2004 rule headlines in this document has been licensed to Quantum Leaps by [MIRA Ltd.](http://MIRA.Ltd)

6.1 Rule 11.4(adv)

A cast should not be performed between a pointer to object type and a different pointer to object type.

The QP-nano applications deviate from rule 11.4 in the QP-nano macros `Q_SIG()`, `Q_PAR()`, and `QF_ACTIVE_CAST`.

Macros `Q_SIG()` and `Q_PAR()` are used by the QP-nano applications to access signal and parameter of the current event, respectively. These macros perform casting of the “me” argument to the `QFsm` base class. This particular cast is safe, because it is an up-cast (from the subclass to the base class). The following snippet of code shows examples of using the macros `Q_SIG()` and `Q_PAR()`:

```
switch (Q_SIG(me)) {
    case MY_EVENT_SIG: {
        me->foo = Q_PAR(me);
        . . .
    }
    . . .
}
```

The macro `QF_ACTIVE_CAST()` performs also an up-cast, but from a **partially defined** type. For example, to achieve strict encapsulation of active objects, the header file `pelican.h` in the PELICAN crossing example declares active objects only as external structs, which are not actually provided:

```
extern struct PelicanTag AO_Pelican;
extern struct PedTag     AO_Ped;
```

Such partially-specified types are sufficient, because the rest of the application needs to use only **pointers** to the active objects (e.g., `&AO_Pelican` or `&AO_Ped`) in order to post events to them. At the same time the addresses to these objects can be very efficiently resolved at link time.

The problem is that PC-Lint, not knowing the actual type, raises warning 826 “Suspicious pointer-to-pointer conversion (area too small)”. This is because PC-Lint assumes that the size of the partially-defined `struct PelicanTag`, for example, might be smaller than the `struct QActive`, to which the pointer is up-cast.

However, in this particular context, the upcasting of `&AO_Pelican` or `&AO_Ped` to `(QActive *)` is safe, even though PC-Lint does not “know” the actual size of the `struct PelicanTag`. The option file `lib-qpn.lnt` suppresses the PC-lint warning 826 for the macro `QF_ACTIVE_CAST()` and the QP-nano functions for posting events to active objects `QActive_post()` and `QActive_postISR()`. The following snippet of code shows examples of using the macro `QF_ACTIVE_CAST()` and the function `QActive_post()`:

```
QActive_post(QF_ACTIVE_CAST(&AO_Pelican),  
            (QSignal)PEDS_WAITING_SIG, QParam)0);
```

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

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-nano 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(&Pelican_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-nano 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).

6.3 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-nano, 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 PELICAN examples with lint described in Section 6. 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.

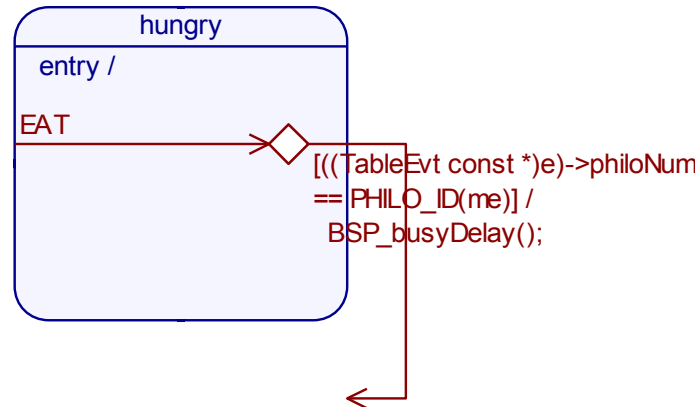
Listing 4: MISRA-compliant state handler implementation

```
static QState Philo_hungry(Philo * const me) {
(1)   QState status;
      switch (Q_SIG(me)) {
          case Q_ENTRY_SIG: {
              QActive_post((QActive *)&AO_Table, HUNGRY_SIG, me->super.prio);
(2)           status = Q_HANDLED();
(3)           break;
          }
          case EAT_SIG: {
(4)           if (Q_PAR(me) == PHILO_ID(me)) {
(5)               status = Q_TRAN(&Philo_eating);
          }
(6)           else {
(7)               status = Q_UNHANDLED();
          }
(8)           break;
          }
          default: {
(9)           status = Q_SUPER(&QHsm_top);
(10)          break;
(11)          }
      }
(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-nano 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.
- (5) When the guard condition in the `if`-statement evaluates to TRUE, the return value is set to `Q_HANDLED()` macro. This macro tells the QEP-nano event processor that the event has been handled.

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 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.

7 Summary

The QP-nano 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-nano comes with extensive support for automatic rule checking by means of PC-Lint, which is designed not just for proving compliance of the QP-nano 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-nano framework.

8 Related Documents and References

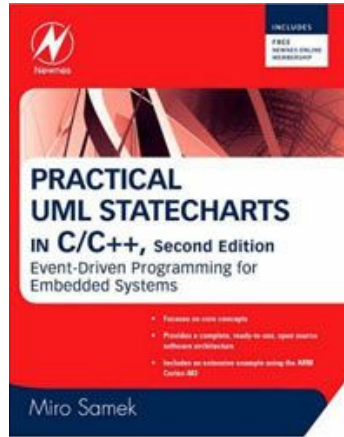
Document	Location
[MISRA-C:2004] MISRA-C:2004 Guidelines for the Use of the C language in Critical Systems, MISRA, October 2004, ISBN: 978-0-9524156-2-6 paperback ISBN: 978-0-9524156-4-0 PDF	Available for purchase from MISRA website http://www.misra.org.uk
[MISRA-C:1998] Guidelines for the Use of the 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 download 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 download 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-PELICAN 08] “Application Note: PEdestrian LIght CONtrolled Crossing Application”, Quantum Leaps, LLC, 2008	http://www.state-machine.com/resources/AN_PELICAN.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

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.

