Application Note

C/C++ Coding Standard

Document Revision G
August 2008

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

Table of Contents

1 Goals

Adopting a reasonable coding standard is perhaps the first and easiest step towards improving quality of code, increasing maturity level, or achieving any code certification. The following coding guidelines are intended to improve code portability, readability, uniformity and maintainability for software. The primary objective of this Coding Standard is to boost software productivity and lower maintenance cost by promoting commonality and avoiding misinterpretations and guessing. The Standard is designed mostly with embedded systems in mind and, where applicable, recommends rules established by Motor Industry Software Reliability Association (MISRA) [MISRA 98].

2 General Rules

The rules in this category are intended for better source code *portability* and consistent code rendering on a wide variety of devices and platforms like: different screens, printers, and various code editors working on different computer platforms. These rules should apply to all ASCII documents, not only strictly to code.

- **No tabs** are allowed in the source code because tabs are rendered differently on different devices and bring only insignificant memory savings. Preferably, tabs should be *disabled* at the editor level. At the very least, they should be replaced by spaces before saving the file.
- Source code lines should **never exceed 78 columns**, and not even "white characters" (spaces or tabs) are allowed past the 78th column. Limiting the column-count is intended to enable side-by side editing and side-by-side differencing of code without horizontal scrolling of the text. (As all GUI usability tests agree, horizontal scrolling of text is always a bad idea.) In practice, the source code is very often copied-and-pasted and then modified, rather than created from scratch. For this style of editing, it's very advantageous to see simultaneously and side-by-side both the original and the modified copy. Also, differencing the code is a routinely performed action of any VCS (*Version Control System*) whenever you check-in or merge the code. Limiting the column count allows to use the horizontal screen real estate much more efficiently for side-by-side-oriented text windows instead of much less convenient and error-prone top-to-bottom differencing.
- All source code should consistently use only **one end-of-line convention**. For improved portability, this Standard recommends consistent use of the Unix™ end-of-line convention, with only one LF character (0x0A) terminating each line. The DOS/Windows end-of-line convention with CR, LF character pair (0x0D, 0x0A) terminating each line is not recommended because it causes compilation problems on Unix™-like systems, such as Linux™. (Specifically, the C preprocessor doesn't correctly parse the multi-line macros.) On the other hand, most DOS/Windows compilers seem to tolerate the Unix™ EOL convention without problems.
- Highly portable source code intended for wide use on different platforms should be limited to the **old DOS 8.3 convention** for file and directory names. This restriction is imposed to allow using older tools like embedded cross-compilers, linkers etc. still working in the DOS environment. The 8.3 file name convention seems to be the least common denominator supported by all computer platforms.
- All source code should use **only lower-case names for files and directories**. This restriction promotes portability, because it eliminates the possibility of inconsistencies in the names. For example, a Unix™-like system might not include correctly a file "`qevent.h`" when the file name is actually "`Qevent.h`".

3 C/C++ Layout

3.1 Expressions

The following binary operators are written with no space around them:

->	Structure pointer operator	me->foo
.	Structure member operator	s.foo
[]	Array subscripting	a[i]
()	Function call	foo(x, y, z)

Parentheses after function names have no space before them. A space should be introduced after each comma to separate each actual argument in a function. Expressions within parentheses are written with no space after the opening parenthesis and no space before the closing parenthesis. Terminating semicolons should follow the instructions immediately with **no** space(s) before them:

```
strncat(t, s, n);
```

The unary operators are written with no spaces between them and their operands:

```
!p     ~b    ++i     j--     (void *)ptr    *p     &x   -k
```

The binary operators are preceded and followed by one (1) space, as is the ternary operator:

```
c1 == c2     x + y     i += 2     n > 0 ? n : -n
```

The keywords `if`, `while`, `for`, `switch`, and `return` are followed by one (1) space:

```
return foo(me->x) + y;
```

In case of compound expressions, parenthesizing should be used whenever the precedence is not "obvious". In general, over parenthesizing is recommended to remove any doubt and guessing. In the extreme case of MISRA-C Rules [MISRA 98], no dependence should be placed on C's operator precedence whatsoever (MISRA-C rule 47), so every expression must be parenthesized. In addition, MISRA rules require that the operands of `&&` and `||` shall be primary expressions (MISRA-C rule 34).

Following are examples of parenthesizing consistent with the strict MISRA-C rules:

```
(a < b) && (b < c) /* operands of && are primary expressions (MISRA rule 34) */
x = (a * b) + c;                 /* don't rely on precedence of '*'over '+' */
```

3.2 Indentation

All indentations must be exactly **four (4) spaces** to indicate scope. All declarators and control statements are formatted according to the following "line saver" template:

```
<declarator | control statement> {
    <statement>;
    . . .
}
```

where the <declarator | control statement> may be one of the following:

- function declarator
- structure/class declarator
- enumeration
- structure/array initializer
- control statement (if, while, for, switch, case, do)

The Standard requires that the statement following any of the keywords (if, else, while, for, switch, case, do) must be *compound*, that is, use of braces is obligatory. This requirement corresponds to MISRA-C rule 59. Although not strictly required by the C/C++ syntax, this rule is imposed to remove any doubts as far as statement nesting is concerned, and to allow easy addition/removal of nested statements with minimal differences showing when differencing the code using a VCS. For example, consider adding a statement to a while loop:

```
/* not a compound statement—not recommended */
while (i > 0)
    *t++ = *s++;
```

```
/* compound statement-recommended */
while (i > 0) {
    *t++ = *s++;
}
```

after modification:

```
/* not a compound statement—not recommended */
while (i > 0)
    *t++ = *s++;
    --i;
```

```
/* compound statement-recommended */
while (i > 0) {
    *t++ = *s++;
    --i;
}
```

With the not-compound statement case you either make a mistake by forgetting the braces (although the indentation clearly indicates the intention), or you must introduce the braces, which then show as a big difference when merging the code or checking it into the VCS.

If the <declarator | control statement> is so complex that it cannot fit in one line, then it should be formatted in the following manner:

```
<declarator | control statement ...>
    <... declarator | control statement ...>
    <... declarator | control statement>
{
    <statement>;
    . . .
}
```

The arguments of a declarator should be split into separate lines leaving separating comma ‘,’ as the last character in the line. The condition of the control statement should be split into separate

lines starting new lines with a binary operator. This formatting intends to avoid confusing continuation of the control statement with its body. For example:

```
if (!decodeData(&s, &dataType,
                &format, &version)
    && format != 2                          /* expecting only format 2 */
    && version != 1)                       /* expecting only version 1 */
{
    log("data corrupt");
}
```

3.2.1 The if Statement

```
if (x < 0) {
    z = 25;
}
if (--cnt == 0) {
    z = 10;
    cnt = 1000;
}
else {
    z = 200;
}
if (x > y) {
    foo(x, y);
    z = 100;
}
else {
    if (y < x) {
        foo(y, x);
        z = 200;
    }
    else {
        x = 0;
        y = 0;
    }
}
```

3.2.2 The for Statement

```
for (i = 0; i < MAX_ITER; ++i) {
    *p2++ = *p1++;
    xx[i] = 0;
}
```

3.2.3 The while Statement

```
while (--ctr != 0) {
    *p2++ = *p1++;
    *p3++ = 0;
}
```

3.2.4 The do..while Statement

```
do {
    --ctr;
    *p2++ = *p1++;
} while (cnt > 0);
```

3.2.5 The switch Statement

```
switch (key) {
    case KEY_BS: {
        if (--me->ctr == 0) {
            me->ctr = PERIOD;
        }
        break;
    }
    case KEY_CR: {
        ++me->crCtr;
        /* intentionally fall through */
    }
    case KEY_LF: {
        ++p;
        break;
    }
    default: {
        ASSERT(0);                                  /* should never be reached */
        break;
    }
}
```

Any fall though cases must be documented with comments confirming intentional fall through rather then an omission.

3.2.6 Function Declarations

```
void clrBuf(char *buf[], int len) {
    char *b = &buf[0];
    while (len-- != 0) {
        *b++ = '\0';
    }
}
```

3.2.7 C++ Class Declaration

```
class Foo : public Bar {
public:
    Foo(int8_t x, int16_t y, int32_t z)                          // ctor
        : Bar(x, y), m_z(z)
    {}

    virtual ~Foo();                                              // xtor
    virtual int32_t doSomething(int8_t x);                      // method

protected:
    virtual void *bar();

private:
    friend class Tar;
    friend void *fxyz(int16_t i);

    int8_t   m_x;
    uint16_t m_y;
    int32_t  m_z;
};
```

3.3 Commenting

Code implements an algorithm; the comments communicate the code's operation to yourself and others. Adequate comments allow you to understand the system's operation without having to read the code itself.

Comments can be generally divided into three categories:

- Elaborate high-level comments for major software components like: modules, classes, and exported APIs.
- Brief, fine granularity comments explaining algorithmic details of performed operations.
- References and notes attached to code in form of comments.

Comments in the first category explain high level interfaces and the code structure. With help of automatic documentation generating tools like JavaDoc for Java or DOC++ for C/C++, these comments can easily be turned into online (HTML) documentation. The Standard does not require that every top-level comment be verified in view of converting it into on-line documentation. However, adhering to the following simple rules will make such automatic documentation extraction much easier, if we want to take advantage of it:

Top-level comments should always come *before* the commented object.

No right-edge in comment boxes should be used, because keeping the right-edge of the box aligned is **counterproductive**:

```
/* INCORRECT: */
/*************************************************************************
* this is class Foo                                                      *
* . . .                                                                  *
*************************************************************************/
class Foo {
    ...
};

/* CORRECT: */
//////////////////////////////////////////////////////////////////////////
// class Foo performs the following
// ...
//
class Foo {
    ...
};
```

Comments in the second category are typically low-level, algorithmic details. They should be placed as close to the pertaining code as possible, preferably in the same line. Although close to the code, the comments should be visually separated from code as much as possible by right-justifying them. Following section gives examples of aligning such comments:

References and notes should be attached in form of the "notes" comment at the end of the module as shown in the next section describing the module layout.

Write comments in clear English. Use simple sentences: noun, verb, object. Use active voice. Be complete. Good comments capture everything important about the problem at hand. Ideally, it should be possible to get a sense of the system's operation by reading only the comments.

For portability, **never** use C++ comments (//) in C, although many C compilers recognize C++ comments. (But the point is that some don't!). Conversely, avoid using C comments (/*..*/) in C++.

3.4 Module Layout

The module is organized as follows:

- Header comment block
- #include statements
- Public Section Specification
- Package Section Specification
- Local Section Specification
- Implementation
- Notes

3.4.1 Header Comment

Each module starts with a header comment in the following format:

```
/*****************************************************************************
* Product:  . .
* Version:  . .
* Released: Dec 27 2003
* Updated:  Dec 17 2004
*
* Copyright (C) 2002-2004 Quantum Leaps. All rights reserved.
*
* <licensing terms> (if any)
*
* <Company contact information>
*****************************************************************************/
```

3.4.2 Included Header Files

```
#include "rtk.h"                                        /* Real-Time Kernel */
#include "qassert.h"               /* embedded-systems-friendly assertions */
```

3.4.3 Public Section Specification

Definitions of public (global) variables should appear at the top of the module:

```
/* Public-scope objects ----------------------------------------------------*/
QActive *UI_Mgr;    /* pointer to the User Interface Manager active object */
. . .
```

3.4.4 Package Section Specification

The public (global) variables should be followed by all package-scope variables:

```
/* Package-scope objects ---------------------------------------------------*/
QEvent const QEP_stdEvt[] = {
    { Q_EMPTY_SIG, 0},
    { Q_INIT_SIG,  0},
    { Q_ENTRY_SIG, 0},
    { Q_EXIT_SIG,  0}
};
```

3.4.5 Local Section Specification

The package-scope variables should be followed by local (module-scope) declarations and local variables (module-scope). All local-scope variables should be defined `static`.

```
/* Local-scope objects -----------------------------------------------------*/
static uint32_t l_svMask;    /* Space Vehicle mask indicating allocated SVs */
. . .
```

3.4.6 Implementation Section

The implementation section contains function definitions (in C) or class method definitions (in C++). Regardless of language, keep functions small. The ideal size is less than a page; in no case should a function ever exceed two pages. Break large functions into several smaller ones.

The only exception to this rule is the very rare case where you must handle very many events in one state handler. However, even in this case you should try to use state nesting (behavioral inheritance) to move some of the events to higher-level state handlers (See also [Samek 02, Section 4.3.1]).

Define a prototype for every function, even the static helper functions called only from within their own module. (The good place for such static prototypes is the local-scope section.) Prototypes let the compiler catch the all-too-common errors of incorrect argument types and improper numbers of arguments. They are cheap insurance.

In general, function names should follow the variable naming conventions (see below). Remember that functions are the “verbs” in programs - they do things. Incorporate the concept of “action words” into the variables’ names. For example, use “readAD” instead of “ADdata”.

Elaborate function comments (or class method comments in C++) should be placed in the header files, since the header files form the API documentation. These comments, generally, should not be repeated in the implementation section because this would create double points of maintenance. The comments at function definitions should rather refer to implementation details and revision history. At a minimum, the functions should be visually separated from each other.

```
/*..........................................................................*/
void swap(int *x, int *y) {
    Q_REQUIRE((x != (int *)0) && (y != (int *)0));/* pointers must be valid */
    int temp = *x;                       /* store value of x in a temporary */
    *x = *y;
    *y = temp;
}
/*..........................................................................*/
int pow(int base, unsigned int exponent) {
    int result = 1;
    for (; exponent > 0; exponent >>= 1) {
        if ((exponent & 1) != 0) {                     /* exponent is odd? */
            result *= base;
        }
        base *= base;
    }
    return result;
}
```

3.4.7 Notes Section

The Notes Section is the ideal place for longer explanations that apply to multiple sections of code or would otherwise distract the reader when placed directly at the code. In the pertinent code you place just a reference to the Note:

```
/*..........................................................................*/
void interrupt dumpISR() {
    . . .                           /* dump channel data to RAM, see NOTE03 */
}
```

At the end of the module, you place the Notes Section:

```
/****************************************************************************
* . . .
*
* NOTE03:
* this system has two hard real-time constraints:
* #1 processing of channel DUMP data must complete before next DUMP
*    for the channel. This constraint is ensured in Assertion (NOTE03.1).
*
* #2 no accumulated data can be missed due to new DUMP for a channel
*    before the previous data has been read. This constraint is ensured
*    in Assertion (NOTE03.2).
*/
```

4 Exact-Width Integer Types

Avoid the use of "raw" C/C++ types, as these declarations vary depending on the machine (MISRA-C rule 13). The recommended strategy is to always use a C-99 <stdint.h> header file [C99, Section 7.18]. In case this standard header file is not available (e.g., in a pre-standard compiler), you should create it and place in the compiler's include directory. At a minimum, this file should contain the typedefs for the following exact-width integer data types [C99 Section 7.18.1.1]:

exact size	signed	unsigned
8 bits	int8_t	uint8_t
16 bits	int16_t	uint16_t
32 bits	int32_t	uint32_t

The main goal of the <stdint.h> indirection layer is promotion of code portability across different platforms. To achieve this goal the C99-style types listed above should be consistently used instead of the "raw" C/C++ types, such as long or unsigned char, and inventing different aliases for the C/C++ types is forbidden.

5 Names

5.1 Reserved Names

The ANSI C specification restricts the use of names that begin with an underscore and either an uppercase letter or another underscore (regular expression: _[A-Z_][0-9A-Za-z_]). Much compiler runtime code also starts with leading underscores.

These names are also reserved by ANSI for its future expansion:

Regular expression	purpose
E[0-9A-Z][0-9A-Za-z]*	errno values
is[a-z][0-9A-Za-z]*	Character classification
to[a-z][0-9A-Za-z]*	Character manipulation
LC_[0-9A-Za-z_]*	Locale
SIG[_A-Z][0-9A-Za-z_]*	Signals
str[a-z][0-9A-Za-z_]*	String manipulation
mem[a-z][0-9A-Za-z_]*	Memory manipulation
wcs[a-z][0-9A-Za-z_]*	Wide character manipulation

To improve portability and avoid name conflicts, never use a name with a leading underscore or one of the name patterns reserved for future expansion.

5.2 Naming Conventions

This section does not intend to impose strict "Hungarian-type" notation. However, the following simple rules in naming various identifiers are strongly recommended:

- No identifier should be longer than 31 characters (this is a stricter version of MISRA-C rule 11).
- **Type names** (`typedef`, `sturct` and `class`) should start with an **upper-case** letter e.g., `struct Foo`. Optionally, the type name can be prefixed with the module identifier, e.g., `typedef uint16_t` **`QSignal`**, `class` **`QActive`**.
- Ordinary C **functions** and C++ class member functions start with a lower-case letter.
- Member functions of classes coded in C (see Section 6) are prefixed with the class name and an underscore, so per the previous rule must begin with an upper-case letter. (`QActive_start()`). Besides clearly distinguishing the member functions, this rule minimizes link-time name conflicts with other functions (including third-party library functions).
- Global functions are prefixed with a module name and an underscore (e.g., `QF_start()`). Package-scope functions, visible only from a closely related group of source files—the pack-

age, are additionally suffixed with an underscore (`QF_add_()`). Besides clearly distinguishing global and package-scope functions, this rule minimizes link-time name conflicts with other functions (including third-party library functions).

- Ordinary **variables** should start with a **lower-case** letter (`foo`).
- Global variables should be prefixed with the module name and an underscore (e.g., `QK_readySet`).
- **Local variables** (visible within one module only) should start with "`l_`", e.g., `l_bitmask`. All local variables should be declared `static` at the file scope (MISRA-C rule 23).
- C++ **class attributes** (data members) should should start with "`m_`", e.g. `int8_t m_foo`. This convention allows easy distinction between the class data members and other variables like, for example, member function arguments.
- **Constants** (numeric macros or enumerations) should be in upper-case with underscores "`_`" between each word or abbreviation (`FOO_BAR`). Global constants should be prefixed with the module name/identifier (`Q_USER_SIG`).
- All other parts of identifiers composed form multiple words should be constructed with capitalizing letters at word boundaries like: `fooBarTar`, and not `foo_bar_tar`.
- Generally, the more broad the scope the more descriptive the name should be. For a very limited scope, it is recommended to use single letter identifiers. For example:
    - `i, j, k, m, n`, for integers like loop counters
    - `p, q, r, s, t, u, v, w`, for pointers or floating point numbers

6 Object Oriented Programming in C

The following guidelines are intended to help in adoption of best practices from Object Oriented Programming into C programming. Contrary to widespread beliefs, it's quite easy to implement encapsulation, single inheritance, and even polymorphism (late binding) in procedural languages, such as C [Samek 97]. Knowing how to implement encapsulation, inheritance, or even polymorphism in C can be very beneficial, because it leads to better code organization and better code reuse.

At the C programming level, encapsulation and inheritance become two simple design patterns. The following section describes the recommended ways of implementing these patterns.

6.1 Encapsulation

Encapsulation is the ability to package data with functions into classes. This concept should actually come as very familiar to any C programmer because it's quite often used even in the traditional C. For example, in the Standard C runtime library, the family of functions that includes `fopen()`, `fclose()`, `fread()`, and `fwrite()` operates on objects of type `FILE`. The `FILE` structure is thus *encapsulated* because client programmers have no need to access the internal attributes of the `FILE` `struct` and instead the whole interface to files consists only of the aforementioned functions. You can think of the `FILE` structure and the associated C-functions that operate on it as the `FILE` *class*. The following bullet items summarize how the C runtime library implements the `FILE` "class":

- Attributes of the class are defined with a C `struct` (the `FILE` `struct`).
- Methods of the class are defined as C functions. Each function takes a pointer to the attribute structure (`FILE *`) as an argument. Class methods typically follow a common naming convention (e.g., all `FILE` class methods start with prefix `f`).
- Special methods initialize and clean up the attribute structure (`fopen()` and `fclose()`). These methods play the roles of class constructor and destructor, respectively.

This is exactly how QP/C and QP-nano frameworks implement classes. For instance, the following snippet of QP/C code declares the `QActive` (active object) "class". Please note that all class methods start with the class prefix ("`QActive`" in this case) and all take a pointer to the attribute structure as the first argument "`me`":

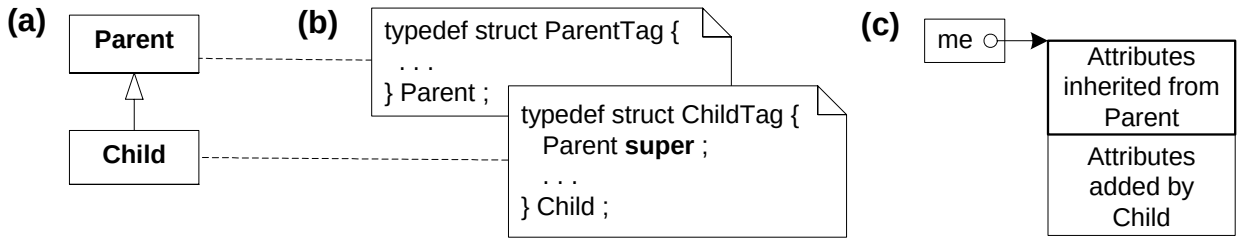
```
typedef struct QActiveTag {
    QHsm super;                                          /* derive from QHsm */
    . . .
    uint8_t prio;                 /* private priority of the active object */
} QActive;                                    /* Active Object base struct */

/* public methods */
int QActive_start(QActive *me, uint8_t prio,
                  QEvent *qSto[], uint16_t qLen,
                  void *stkSto, uint32_t stkSize,
                  QEvent const *ie);
void QActive_postFIFO(QActive *me, QEvent const *e);
void QActive_postLIFO(QActive *me, QEvent const *e);

/* protected methods ...*/
void QActive_ctor(QActive *me, QPseudoState initial);
void QActive_xtor(QActive *me);
void QActive_stop(QActive *me);    /* stopps thread; nothing happens after */
void QActive_subscribe(QActive const *me, QSignal sig);
```

6.2 Inheritance

Inheritance is the ability to define new classes based on existing classes in order to reuse and organize code. QP/C and QP-nano implement single *inheritance* by literally embedding the parent class attribute structure as the first member of the child class structure. As shown in the following figure, this arrangement lets you treat any pointer to the Child class as a pointer to the Parent:



In particular, such memory alignment of the `Child` attributes with the `Parent` attributes allows you to always pass a pointer to the `Child` class to a C function that expects a pointer to the `Parent` class. (To be strictly correct in C, you should explicitly upcast this pointer.) Therefore, all methods designed for the `Parent` class are automatically available to `Child` classes; that is, they are *inherited*.

For example, in the code snippet from the previous section class `QActive` inherits from class `QHsm`. Please note the first protected attribute "`super`" of type `QHsm` in the `QActive` `struct` definition.

7 Design by Contract

Design by Contract is a very powerful set of techniques introduced by Bertrand Meyer [Meyer 97]. The techniques are based on the concept of contract that formally captures assumptions and delicate dependencies between software components. The central idea is to enforce fulfillment of these contracts by instrumenting code and explicit checking against contract violations. Contracts may be in form of preconditions, postconditions and invariants. In C/C++ they are implemented in form of assertions. Some of the listed benefits of Design by Contract include [Samek 03]:

- A better understanding of software construction
- A systematic approach to building bug free systems
- An effective framework for debugging, testing and, more generally, Software Quality Assurance (SQA)
- A method for documenting software components
- Better understanding and control of code reuse
- A technique for dealing with abnormal cases, leading to a sage and effective language constructs for exception handling.

In deeply embedded systems, assertion failure must be treated differently than on desktop computers. Typically, standard actions of printing out assertion failure statement and exit are not the right approach. For that reason a customized behavior in case of contract violation is coded in "qassert.h" include file [QL 04]. This header file defines the following assertions:

Q_ASSERT()	General-purpose assertions
Q_ALLEGE()	Assertions with side-effects in testing the expression, when the side effects are desired even if assertions are disabled
Q_REQUIRE()	For asserting preconditions
Q_ENSURE()	For asserting postconditions
Q_INVARIANT()	For asserting invariants
Q_ASSERT_COMPILE()	Compile-time assertions

8 C/C++ Codesize Metrics

Practice shows (see [Humphrey 95]) that most important aspect of code size metrics is consistency, rather than very elaborate techniques. For the sake of constancy, therefore, the metrics applied should be standardized. The proposed C/C++ codesize metrics is just to count number of matches of following regular expression in the code.

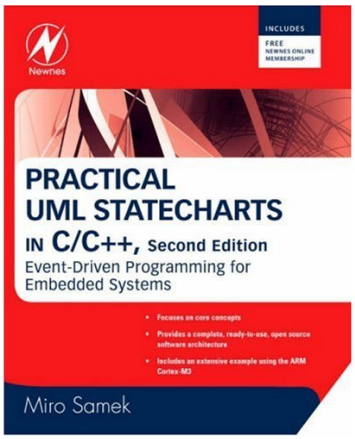
```
;|if \(|else|for \(|while \(
```

9 Related Documents and References

Document	Location
[C99]	ISO/IEC 9899 C Approved Standards http://www.open-std.org/jtc1/sc22/open/n2794
[Humphrey 95]	Humphrey, Watts, S., A Discipline in Software Engineering, Addison-Wesley, Reading 1995.
[Labrosse 00]	Labrosse, Jean J., AN2000: C Coding Standard, http://www.micrium.com
[Meyer 97]	Meyer, Bertrand, Object-Oriented Software Construction 2nd Edition, Prentice Hall, 1997. ISBN: 0-136-29155-4
[MISRA 98]	Motor Industry Software Reliability Association (MISRA), MISRA Limited, MISRA-C:1998 Guidelines for the Use of the C Language in Vehicle Based Software, April 1998, ISBN 0-9524156-9-0. See also http://www.misra.org.uk
[Samek 97]	Portable Inheritance and Polymorphism in C, ESP, December 1997
[Samek 02]	Samek, Miro, Practical Statecharts in C/C++, CMP Books 2002.
[Samek 03]	Samek, Miro, “An Exception or a Bug?” C/C++ Users Journal, August, 2003, pages 36-40
[QL 04]	www.quantum-leaps.com

10 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



"Practical UML Statecharts in C/C++, Second Edition" (**PSiCC2**), by Miro Samek, Newnes, 2008, ISBN 0750687061