AMQ RFC002
C Coding Standards

version 0.1

iMatix Corporation <amq@imatix.com>

Revised: 2005/03/14

Contents

1 Cover

1.1 State of this Document

This document is a request for comments. Distribution of this document is currently limited to iMatix and JPMorgan internal use.

This document is a formal standard. This document is ready for review.

1.2 Copyright Notice

1.3 Authors

This document was written by Rakan El-Khalil <rakan@imatix.com>.

1.4 Abstract

This document describes the ANSI C coding standards that are to be used in the OpenAMQ project.

2 Introduction

Coding standards are useful so as to enable syntactical uniformity in any contributed code. In the following standard, we have chosen to retain compatibility with existing iMatix standards where possible, with an added emphasis on readability and clarity. All OpenAMQ ANSI C code must conform to it.

3 Coding Standards

3.1 Header Files

All applications must include icl_types.h and conform to the datatypes and macros defined in it. Specifically: - the use of the 'byte', 'dbyte', 'qbyte' data types for all unsigned integers where the size is important to know - the use of 'Bool' for all Boolean values - the use of 'uint' for unsigned integers - the use of 'streq' and 'strneq' for string comparison

The goal here is to use short names for common things and to be consistent across all our code. Further, in C, there is good reason to know the size of integers and the byte/dbyte/qbyte nomenclature is both accurate and efficient.

3.2 Language Definition

The language standard is ANSI 1989 C with a specific set of extensions as listed below: - inline and static inline functions may be supported. - 64-bit integers may be supported.

3.3 Spacing

- Code should be formatted to fit an 80 character width terminal [may push that to at most 90 char width].
- Indent with 4 spaces, never use tabs.
- Spaces outside () and [], not inside.
- Space after commas.
- No space after -> or . in accessing pointers/structs.

3.4 Naming Conventions

- All names are in English.
- Variable names are always lowercase with '_' as delimiter.
- Exported functions and variables are prefixed as lib_module_func.
- In rare cases, may abbreviate to lib_func.
- Macros are always capitalized.
- Type names end in _t,
- Functions or variables that are static to a single file are prefixed with s_,
- It is permissible to use short variable names within functions, as long as their definition is reasonably close to their place of use. (and functions should not be very long anyway.)

- Use xxx_nbr and xxx_ptr for local array indices and pointers in functions.
- Filenames are named lib_module.{c, h}.

3.5 C Constructs

Write function braces like this:

```
int foo (...)
{
    ...
}
```

Complex function APIs can be broken over several lines, both in prototypes and in function definitions:

```
some_t function (
    int arg1,                       /*  Comments                        */
    int arg2                        /*  ...                             */
)
```

Conditionals braces:

```
if (condition) {
    ...
}
else {
    ...
}
```

Declaration of several variables should span several lines:

```
int
    foo,
    bar,
    baz;
```

Nested If/Else statements must be enclosed within braces. E.g. Not:

```
if (foo)
    if (bar)
        ...
```

But:

```
if (foo) {
    if (bar)
        ...
}
```

Assignment may not happen within an if statement. I.e., Not:

```
if ((rv = foo ()))
    ...
```

But:

```
rv = foo ();
if (rv)
    ...
```

Assignment may happen within a while statement, but only with an explicit comparison made:

```
while ((c = getchar ()) != EOF)
    ...
```

3.6 C Comments

Same-line comments start in column 41 (first column being 1) and end on column 80:

```
amq_some_t somevar;                 /*  Message header frame            */
```

Full-line comments start indented matching following code and end on column 80.

Insert two spaces after the initial '/*'

3.7 Error Reporting

The standard practice for unexpected errors is to log a message and return a 'failed' code.

When errors need explicit handling, the standard practice is to return a distinctive code. Such codes are defined in header files and the application documentation if necessary.

Error messages will be issued using the 'coprintf' function, which is designed to copy messages to a console log file, optionally with a date/time prefix.

Your code should issue a message thus:

```
coprintf ("modulename severity: text", insertions...);
```

- modulename is a name that makes sense to the person who has to resolve the error; it can be the application name for configuration errors, or the class or source name for internal faults.
- severity is I, W or E; for informational, warning or error.
- text is a readable text, possibly followed by a system error message.

3.8 Implementation

Prefer using typedefs and static inline function over macros.

All static functions should be placed at the end of source files, with their function prototypes at the top.

The order of methods in code should be: new, destroy, ..., selftest. [if applicable]

3.9 Portability

All non-portable code will be isolated in a portability layer (iPR). The application may not use #if macros for system-specific functionality.

Application code MAY NOT include system header files. These are not portable and lead to non-portable applications. All system header files are encapsulated in sfl.h, which applications will include either directly or through an encapsulated header file.

3.10 Security Considerations

This proposal does not have any specific security considerations.