XMM OPTICAL MONITOR

MULLARD SPACE SCIENCE LABORATORY
UNIVERSITY COLLEGE LONDON

Author: H.E.Huckle

^{XMM}OM C-Language Coding Conventions

Document: XMM-OM/MSSL/SP/0008.02

Issue	Date	Comments
Draft	20 June ‘91	Draft Version for Comment
Issue 1	31 Jan ‘92	Comments Included
Issue 2	1 Jun ‘94	Revised in light of experience

This copy printed at 09:59 AM on 2-Oct-98

CONTENTS

Introduction

(NOTE - This release makes little attempt to define valid C code - for example, should we conform to the ANSI-C standard or to Kernighan and Ritchie 1st Edition with agreed extensions. That revision awaits further discussion. This release is intended to encourage a uniformity of style and layout.)

These suggested *XMM* OM C-programming conventions are intended, in part, to encourage higher quality code: every source module should have certain essential documentation, and the code and documentation should be in a format that has been found to be readable and accessible.

The conventions are also intended to provide a level of uniformity in the code produced by different programmers. Besides making the code look more professional, uniformity allows programmers to work on code written by others with less overhead in adjusting to stylistic differences. Also it allows automated processing of the source: tools can be written to generate manual entries, module summaries, change reports, etc.

These conventions are divided into the following categories:

1. Module Layout
2. Subroutine Layout
3. Code Layout
4. Naming Conventions
5. Style

Acknowledgement - The conventions specified herein have been suggested by, but are not identical to, the Wind River Systems standard for all C code and accompanying documentation.

Module Layout

A module is any unit of code that resides in a single source file. Thus a module may be a UNIX tool, a library of routines, or an applications task. The conventions in this section define the standard for the header which must come at the beginning of every source module.

The module header consists of the blocks described below. The blocks are separated by one or more blank lines and should contain no blank lines within the block. This facilitates automated processing of the header.

Title: The title should consist of one line containing the module names followed by a short description. The module name must be the same as the module file name. This line will become the title of any automatically generated manual entries and indexes.

Modification History: Each entry should consist of the version number, date of modification, initials of the programmer who made the change, and a complete description of the change. The version number should be of the form n.m, preceded by an optional ‘D’ indicating a development version and followed by an optional lower case letter in the range a-z. Incrementing `n' indicates a major release (for example, after major internal changes to the code or to the external user interface). Incrementing `m' indicates a minor release (for example, some additional functionality has been added). Adding or incrementing the letter suffix indicates a `bug-fix'.

- **General Module Documentation**: This should consist of a complete description of overall module purpose and functioning, and especially the external interface.

- **Includes**: This should consist of the `includes` required by the module.

The exact format of these blocks is elaborated in the following example of a standard module header:

```
/* fooLib.c - foo subroutine library */

/*
modification history
-----------------------
01a, 10feb 86, dnw written
01b, 15feb 86, dnw added routines fooGet and fooPut.
                  added check for invalid index in fooFind
*/

/*
DESCRIPTION
This module is an example of XMM-OM C coding conventions.
...
*/

#include "fooLib.h"
```

Subroutine Layout

The following conventions define the standard layout for every subroutine in a module.

Each subroutine should be preceded by a header of documentation that consists of the following blocks. There should be no blank lines in the header but each block should be separated with a line containing a single asterisk (*) in the first column.

- **Banner**: This is a slash character followed by asterisks across the page.
- **Title**: One line containing the routine name followed by a short description. The routine name in the title must match the declared routine name. This line will become the title of any automatically generated manual entries and indexes.
- **Description**: Complete description of what the subroutine does and how to use it.
- **Returns**: The word ‘‘RETURNS:’’ followed by a description of the possible result values of the subroutine.

Immediately following the subroutine header should be the subroutine declaration itself. Each argument to the routine must be described with a comment. This declaration will become the synopsis in any automatically generated manual entries.

The exact format of the subroutine header and declaration is shown in the following two examples of a standard subroutine layout, one non-ANSI-C:

```
/******* non ANSI-C Version **************************
*
* fooGet - get an element from a foo
*
* This routine finds the element of the specified
* index in the specified foo.
*
* RETURNS:
*  OK if element found, otherwise ERROR.
*  The value of the element found is returned
*  in *PValue.
*****************************************************/

STATUS fooGet (foo, index, pValue)
```

```
     FOO foo;     /* foo in which to find element */
     int index;   /* element to be found in foo */
     int *pValue; /* where to return value */

     {
     ...
     }

/******* ANSI-C Version *************************
*
* fooGet - get an element from a foo
*
* This routine finds the element of the specified
* index in the specified foo.
*
* RETURNS:
*  OK if element found, otherwise ERROR.
*  The value of the element found is returned
*  in *Pvalue.

***************************************************/

STATUS fooGet (FOO foo, int index, int *pValue)
     /* FOO foo;      foo in which to find element */
     /* int index;    element to be found in foo */
     /* int *pValue;  where to return value */

     {
     ...
     }
```

Code Layout

The following conventions define the standard for the graphic layout of C code.

Vertical Spacing

1. You should use blank lines to make code more readable and to group logically related sections of code together. Put a blank line before and after comment lines
.

2. You should not put more than one statement on a line. The only exception is the **`for`** statement where the initial, conditional, and loop statements may be written on a single line:

```
for (i = 0; i < count; i++)
```

The **`if`** statement is not an exception: the executed statement should always go on a separate line from the conditional expression:

```
if (i  > count)
  i  = count;
```

3. Section braces '**`{`**' and '**`}`**' and **`case`** labels should always get their own line.

Horizontal Spacing

1. You should put spaces around binary operators (including all forms of the assignment), after commas, and before open parenthesis. Do not put spaces around structure member and pointer operators.

   For example (where indicates a space)

```
status = fooGet (foo, i + 3, &value);
foo.index
pFoo->index
fooArray[(max + min) / 2]
string[0]
```

2. Continuation lines should line up with the part of the preceding line they continue. They should start at 'sensible' places, e.g. between major terms in equations, after the separating comma in argument lists, etc:

```
a = (b + c) *
     (d + e);
status = fooList (foo, a, b, c,
                  d,e);
if ((a == b) &&
    (c == d))
.
```

Indentation

1. Tab stops should be at least every 2 characters (i.e. 1, 3, 7, ...).

2. The module and subroutine headers and the subroutine declarations should start in column 1.

3. You should indent one tab stop after:

- **subroutine** declarations
- **conditionals** (see below)
- **looping** constructs
- `switch` statements
- `case` labels
- **structure** definitions in a `typedef`

4. The `else` of a conditional should have the same indentation as the `if`. Thus the form of the conditional is:

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

5. The general form of the `switch` should be:

```
switch (input)
```

```
  {
  case 'a':
    ...
    break;
  case 'b':
    ...
    break;
  default:
    ...
    break;
  }
```

If the actions are very short and nearly identical in all cases, an alternate form of the **switch** statement is acceptable:

```
switch (input)
  {
  case 'a': x = aVar; break;
  case 'b': x = bVar; break;
  case 'c': x = cVar; break;
  }
```

6. Comments should have the same indentation as the section of code to which they refer.

7. Section braces '**{**' and '**}**' should have the same indentation as the code they enclose.

Naming Conventions

The following conventions define the standards for naming modules, routines, variable, constants, macros, types, and structure and union members. The purpose of these conventions is uniformity and readability of code.

1. When making up names, keep the following in mind: the code will be written once but read many times. Make names meaningful and readable. Avoid obscure abbreviations.

2. Names of routines, variables, and structure and union members may be written with upper and lower case. Each ''word'' may be capitalised or separated by underlines:

```
aVariableName
another_name
```

3. Names of defined types (via **typedef**s), and constants and macros (via #**define**s), should be all upper case with underlines separating the words in the name

```
A_CONSTANT_VALUE
```

4. Pointer variables should be clearly indicated by a naming convention, for example they could have a ''p'' prefix for each level of indirection:

```
FOO_NODE  *pFooNode;
FOO_NODE  **ppFooNode;
```

Style

The following conventions define additional standards of programming style:

- **Comments**: Insufficiently commentedcode is unacceptable code.

- **Lint**:.- when available - should be run to check for possible errors that would not be caught by the compiler. All modules must be coded to eliminate all avoidable `lint` complaints.

- **Manual Generation**: A manual generation tool - if available - should be used togenerate manual entries for every module automatically. All modules must generate valid manual entries.

- **Numeric Constants**: Use **`#define`** to define meaningful names for all constants. Never use numeric constants in code or declarations (except for obvious uses of 0, 1, etc.).

- **Structure Definitions**: Always define structures and unions by a **`typedef`** declaration. Never use a structure or union definition to declare a variable directly.

- **Passing and Returning Structures**: Always pass and return pointers to structures. Never pass or return structures directly.

- **Use Defined Names**: Use defined names wherever possible. for example:

  ⇒ Use **`TRUE`** and **`FALSE`** for boolean compares

  ⇒ Use **`NULL`** for zero pointer tests

  ⇒ Use EOS for end-of-string tests.

  ⇒ etc