Micriµm, Inc.

C Coding Standard

Application Note AN-2000

Jean J. Labrosse Jean.Labrosse@Micrium.com www.Micrium.com

1.00 Introduction

Conventions should be established early in a project. These conventions are necessary to maintain consistency throughout the project. Adopting conventions increases productivity and simplify project maintenance.

There are many ways to code programs in C (or any other language). The style you use is just as good as any other as long as you strive to attain the following goals:

Portability Consistency Neatness Easy maintenance Easy understanding Simplicity

Whichever style you use, I would emphasize that it should be adopted consistently throughout all your projects. I would further insist that a single style be adopted by all team members in a large project. Adopting a common coding style reduces code maintenance headaches and costs. Adopting a common style will avoid code rewrites. This application note describes the style I've been adopting for years.

2.00 Basic Principals

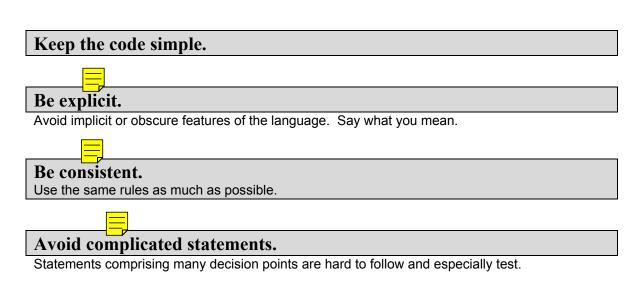
The fundamental purpose of these standards is to promote maintainability of the code. This means that the code must be readable, understandable, testable and portable.

Keep the *spirit* of the standards.

Where you have a coding decision to make and there is no direct standard, then you should always keep within the spirit of the standard.

All code should be written to ANSI C standards.

This means that function prototypes should be ANSI and therefore, the type definitions should be included within the parenthesis.



Do not use GOTO.

Updating old code.

Whenever existing code is modified try to update the document to abide with the conventions outlined in this document. This will ensure that old code will be upgraded over time.

3.00 Source Files

Line width.

You should NOT limit the width of C source code to 80 characters just because yesterday's monitors only allowed you to display 80 characters wide. The width of a line could be based on how many characters can be printed on an 8.5" by 11" page using a reasonable font size. You should be able to accommodate up to 132 characters (portrait mode) and have enough room on the left of the page for holes for insertion in a three ring binder. Allowing 132 characters per line prevents having to interleave source code with comments. If more characters are needed to make the code clearer then you should not be limited to 132 characters. In fact, you could have code that contains initialized structures (placed in Read-Only-Memory, ROM) that are over 300 characters wide. Of course, you can't see (nor print) all the elements of these tables at once but at least the different fields line up neatly.

Use of TAB character.

TAB characters (ASCII character 0x09) MUST NOT be used. Indentation MUST be done using the SPACE character only (ASCII character 0x20).

TAB characters expand differently on different computers and printers. Avoiding them ensures that the intended spacing is maintained.

Indent level is 4 spaces.

Indentation of code will consist of 4 spaces (ASCII character 0×20). Note that statements under a case statement is actually indented by 5 spaces (see the section on Construct). Always try to start on multiples of 4 spaces (column 1, 5, 9, 13, 17 etc.).

Include a file heading.

At the beginning of each file, include a comment block containing the company name, address, copyright notice, list of programmers, description of the file, etc. See below.

```
/*
        **
                            Company Name
                          Company Address
                          City, State ZIP
                              Country
               (c) Copyright YYYY, Company Name, City, State
* All rights reserved. Company Name's source code is an unpublished work and the
* use of a copyright notice does not imply otherwise. This source code contains
* confidential, trade secret material of Micrium, Inc. Any attempt or participation
* in deciphering, decoding, reverse engineering or in any way altering the source
* code is strictly prohibited, unless the prior written consent of Company Name
* is obtained.
* Filename
            :
* Programmer(s): Joe Programmer (JP)
             John Doe
                           (JD)
* Created
            · YYYY/MM/DD
* Description :
******
                      *******
*/
```

An implementation file is a file that contains executable statements whereas a header file does not. Both types of files are laid out in a similar fashion as shown below. A file will contain either

Micriµm, Inc. ^{C Coding Standard}

part or all of the these sections. Empty sections can be ommitted but, if a section is included it must take its place in the following order as shown below.

Implementation File Layout:

File heading Revision history #include #define constants Macros Local data types Local data types Local variables Local tables Local function prototypes Same order as they are implementated Global functions Order functions by functionality Local functions Order functions by functionality

Header File Layout:

File heading Revision history #define constants Global macros Global data types Global variables Externals Global function prototypes Same order as the implementation file Separate sections for functions declared in other files. #error section Section used for 'flagging' missing or illegal #define values.

Separate major sections.

Every section should be preceded with a comment block as shown below.

BOOLEAN OSIsTaskRdy(void);

Header files SHOULD be guarded from duplicate inclusion by testing for the definition of a value.

Note that !defined (X) is preferable to #ifndef X.

#if !defined (module_H)
#define module_H

Body of the header file.

#endif

/* End of module_H

*/

Use #error to flag missing #define constants or macros and, to check for invalid values.

The standard C preprocessor directive #error should be used to notify the programmer when #define constants or macros are not present and to indicate that a #define value is out of range. These statements are normally found in a module's .H file. The #error directive will display the message within the double quotes when the condition is not met.

```
#ifndef OS_MAX_TASKS
#error "OS_CFG.H, Missing OS_MAX_TASKS: Max. number of tasks in your application"
#else
    #if OS_MAX_TASKS < 2
    #error "OS_CFG.H, OS_MAX_TASKS must be >= 2"
    #endif
    #if OS_MAX_TASKS > 63
    #error "OS_CFG.H, OS_MAX_TASKS must be <= 63"
    #endif
#endif
#endif
#endif</pre>
```

4.00 Commenting

Only use C style comments (i.e. /* and */) and not the C++ style comments (i.e. //).

Make every comment count.

Keep code and comments visually separate.

Minimize comments embedded among statements. **NEVER** start comments immediately above the code as shown below. This makes the code difficult to follow because the comments are distracting the visual scanning of the code.

```
void ClkUpdateTime (void)
                                      /* DO NOT comment like this!
                                                                                */
ł
    /* Update the seconds */
   if (ClkSec >= CLK_MAX_SEC) {
       ClkSec = 0;
       /* Update the minutes */
       if (ClkMin >= CLK MAX MIN) {
           ClkMin = 0;
            /* Update the hours */
           if (ClkHour >= CLK_MAX_HOURS) {
               ClkHour = 0;
           } else {
               ClkHour++;
           }
       } else {
           ClkMin++;
       }
    } else {
       ClkSec++;
   }
}
```

Don't use multi-line comments with a single comment terminator.

NEVER do the following:

/* This type of comment can lead to confusion especially when describing a function like ClkUpdateTime (). The function looks like actual code! */

Use comment blocks to separate sections of code.

A comment block is shown below. Note that the comment block heading is centered and is written using UPPER CASE characters.

Do not use 'Emotions' in comments.

For example, do NOT use comments such as "Let's make this one big happy structure!".

Use structured sentences as much as possible.

You can use UPPER CASE words to emphasize the meaning.

It's also appropriate to use acronyms, abbreviations and mnemonics as long as everybody understand the meaning of those.



Use trailing comments as much as possible.

As much as possible, always start the trailing comment on the same column. If the code goes beyond the selected column, place the comment on the line just above while still starting at the same column. As much as possible, line up the terminating comment charaters. Using trailing comments allows the code to be visually separate from the code.

```
void ClkUpdateTime (void)
{
```

}

```
if (ClkSec >= CLK_MAX_SEC) {
    ClkSec = 0;
    if (ClkMin >= CLK_MAX_MIN) {
        ClkMin = 0;
        if (ClkHour >= CLK_MAX_HOURS) {
            ClkHour = 0;
        } else {
            ClkHour++;
        }
    } else {
            ClkMin++;
    }
} else {
            ClkSec++;
}
```

/* Update	the	seconds	*/
/* Update	the	minutes	*/
/* Update	the	hours	*/

Use #if 0 and #endif to comment out blocks of code.

Comments should never be nested. Instead, use **#if 0** and **#endif** to 'comment out' large portions of code.

#if 0		/* Indicate the reason the code is commented out */
#define DISP TBL SIZE	5	<pre>/* Size of display buffer table */</pre>
#define DISP_MAX_X	80	<pre>/* Max. number of characters in X axis */</pre>
#define DISP_MAX_Y	25	<pre>/* Max. number of characters in Y axis */</pre>
#define DISP_MASK	0x5F	
#endif		

Use special comments to indicate the presense of known bugs, past and future implementations.

For example, you can use the following comments. You can then easily search through your code to find these instances.

Naming Convention 5.00

General conventions:

#de	fine constants:
#de	fine macros:
tyr	edefs:
ent	m tags:
	All upper case characters
	Words separated by an underscore (i.e. '_')
	Examples: DISP BUF SIZE, MIN(), MAX(), etc.
Loc	al variables (i.e. function scope):
	All lower case
	Words separated by an underscore (i.e. '_')
	Use standard names (e.g. i, j, k for loop counters, p for pointers etc.)
File	scope variables:
	Prefixed with the module name followed by an underscore (i.e. '_').
	Names separating words start with an initial capital.
	Declared static.
	Examples: Disp_Buf[], Comm_Ch, etc.
Glo	bal variables:
	Prefixed with the module name.
	Names separating words start with an initial capital.
	Examples: DispMapTbl[], CommErrCtr, etc.
Loc	al functions:
	Prefixed with the module name followed by an underscore (i.e. '_').
	Names separating words start with an initial capital.
	Declared static.
	Example: static void Comm_PutChar()
Glo	bal functions:
	Prefixed with the module name followed by an underscore (i.e. '_').
	Names separating words start with an initial capital.
	Example: void CommInit()

Name separate words with an initial capital (e.g. DispBuf[]).

Old code which contains only lower case characters MUST be separated by the underscore character (i.e. '_', ASCII character 0x2D).

Use acronyms, abbreviations and mnemonics consistently.

Create a standard acronyms, abbreviation and mnemonics dictionary for all to use and adopt. Below is an example of such a list although, not complete. A reverse list sorted by the Acronym, Abbreviation, or Mnemonic field should be generated as well.

Use 'module-object-operation' format with acronyms, abbreviations and mnemonics.

When creating global constant, variable and function identifiers, specify the name of the module (or sub-system) first, followed by the object and then the operation as shown below.

OSSemPost() OSSemPend() etc.

K

Description	Acronym, Abbreviation, or Mnemonic			
Argument	Arg			
Buffer	Buf			
Clear	Clr			
Clock	Clk			
Compare	Стр			
Configuration	Cfg			
Context	Ctx			
Delay	Dly			
Device	Dev			
Display	Disp			
Error	Err			
Function	Fnct			
Hexadecimal	Hex			
High Priority Task	HPT			
I/O System	IOS			
Initialize	Init			
Mailbox	Mbox			
Manager	Mgr			
Maximum	Мах			
Message	Msg			
Minimum	Min			
Operating System	OS			
Overflow	Ovf			
Pointer	Ptr			
Previous	Prev			
Priority	Prio			
Read	Rd			
Ready	Rdy			
Schedule	Sched			
Semaphore	Sem			
Stack	Stk			
Synchronize	Sync			
Timer	Tmr			
Trigger	Trig			
Write	Wr			

ALWAYS use the standard acronyms, abbreviations or mnemonics.

Always use the acronym, abbreviation or mnemonic even though you can write the full word. For example, ALWAYS use Init instead of Initialize!

6.00 Data Types

All data types MUST be declared using upper case characters.

Words are separated by the underscore character (i.e. '_', ASCII character 0x2D).

Use the following portable data types.

All standard C data types MUST be avoided because their size is not portable. Instead, the following data types (INT?? and FP??) should be declared based on the target processor and compiler used.

typedef	unsigned	char	BOOLEAN;	/*	Logical data type (TRUE or FALSE) *	/
typedef	unsigned	char	CHAR;	/*	Unsigned 8 bit character *	/
typedef	unsigned	char	INTO8U;	/*	Unsigned 8 bit value *	/
typedef	signed	char	INT08S;	/*	Signed 8 bit value *	/
typedef	unsigned	short	INT16U;	/*	Unsigned 16 bit value *	/
typedef	signed	short	INT16S;	/*	Signed 16 bit value *	/
typedef	signed	short	INT32U;	/*	Unsigned 32 bit value *	/
typedef	signed	short	INT32S;	/*	Signed 32 bit value *	/
typedef	signed	short	INT64U;	/*	Unsigned 64 bit value (if available)*	/
typedef	signed	short	INT64S;	/*	Signed 64 bit value (if available)*	/
typedef	float		FP32;	/*	32 bit, single prec. floating-point *	/
typedef	double	/	FP64;	/*	64 bit, double prec. floating-point *	/
	$- \gamma$					
2 s	baces					

Structures and Unions MUST be typed.

All structures and unions MUST be typed as shown below. Also, the data type MUST be written using ALL upper case characters. Context will make it obvious that all upper case characters in front of a variable or function must mean that it's a data type as opposed to a constant or macro.

typedef s	truct {		
char	<pre>RxBuf[COMM_RX_SIZE];</pre>	<pre>/* Storage of characters received *</pre>	•/
char	*RxInPtr;	<pre>/* Pointer to next free loc. in buffer *</pre>	·/
char	*RxOutPtr;	<pre>/* Pointer to next char. to extract *</pre>	•/
INT1	6U RxCtr;	<pre>/* Number of characters in Rx buffer *</pre>	•/
char	<pre>TxBuf[COMM_TX_SIZE];</pre>	<pre>/* Storage for characters to send *</pre>	۲.
char	*TxInPtr;	<pre>/* Pointer to next free loc. in Tx Buf *</pre>	·/
char	<pre>*TxOutPtr;</pre>	<pre>/* Pointer to next char to send *</pre>	•/
INT1	6U TxCtr;	<pre>/* Number of characters left to send *</pre>	·/
} COMM_BU	F;		

Structure alignment.

The data types of each member are indented 4 spaces and the structure member names are also ligned up with respect to each other. Notice also that the comments are lined up starting at the same column (possibly starting on the previous line if the structure member ends up going beyond this column).

Data type scope.

If a data type is only to be used in the implementation file then, it MUST be declared in the implementation file. If the data type is global, it MUST be placed in the module's header file.

7.00 Layout



Only have one action per line of code:

Example #1:

DispSegTblIx = 0; DispDigMsk = 0x80;

I Instead of:

DispSegTblIx = 0; DispDigMsk = 0x80;

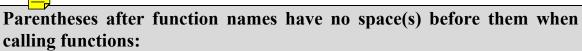
Example #2:

ptcb++;
*ptcb = (OS_TCB *)0;

Instead of:

*++ptcb = (OS_TCB *)0;

Separate code chuncks with blank lines or comments.



DispInit();

At least one space is needed after each comma to separate function arguments:

DispStr(x, y, s);

The binary operators (and the ternary operator) are written with at least one space between them and their operand: c1 = c2;At least 1 space х + у i += 2; n > 0 ? n - n;a < b c >= 2 At least one space is needed after each semicolon: for (i = 0; i < 1); i++)The keywords if, else, while, for, switch and return are followed by one space. if (a > b)while (x > 0)for (i = 0; i < 10; i++)

 \checkmark

} else {

switch (x)

return (y);

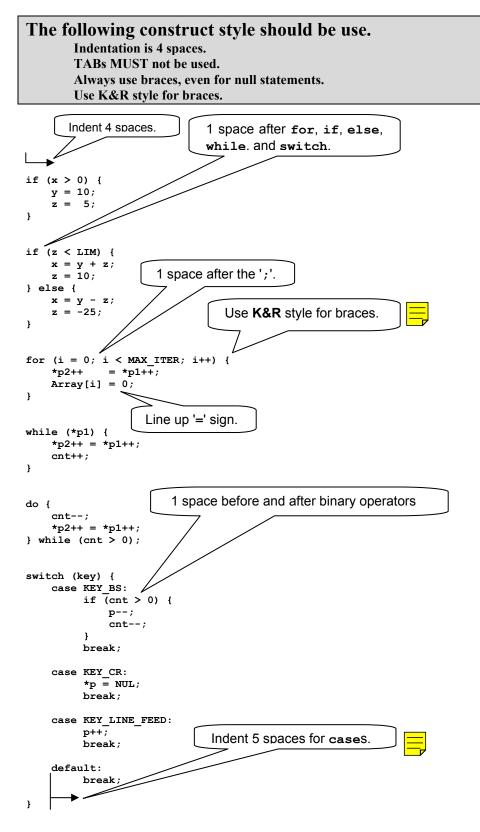
For assignments, the equal signs (i.e. '='), and numbers should be vertically aligned to keep layout tidy.

Note also that the least significant portion of integers and floating-point numbers are lined up.

```
DispSegTblIx = 0;
DispDigMsk = 0x80;
DispScale = 1.25;
```

Expressions within parentheses are written with no space after the opening parenthesis and no space before the closing parenthesis: x = (a + b) * c;

8.00 Constructs



Micriµm, Inc. C Coding Standard

9.00 **Functions**

INT16U OS STK INT32U

void

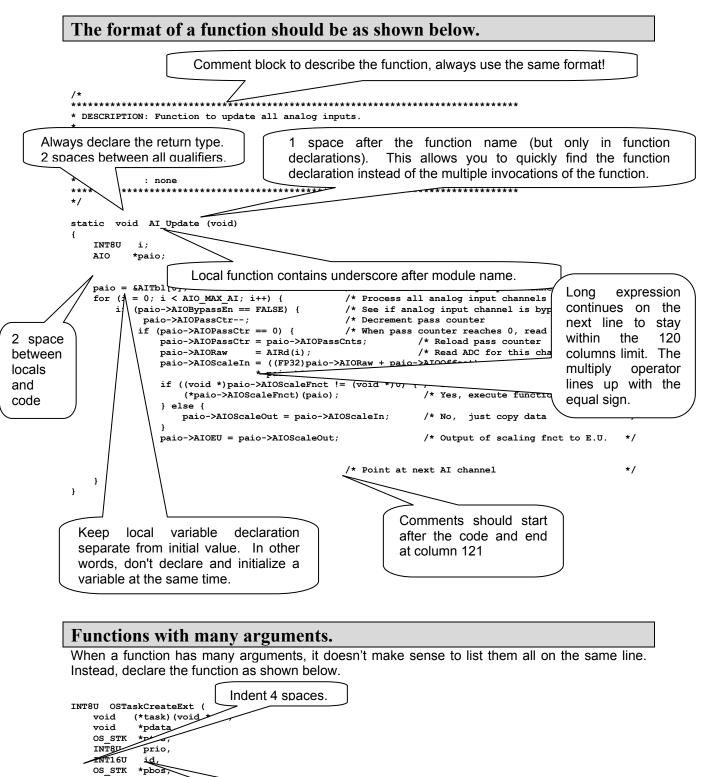
INT16U

stk_size

*pext,

opt)

/* Function body */



Line up the first character of each variable.

One function per page.

As much as possible, there will be one function declared per page.

More than one very small functions can be declared on a single page. However, they must all contain the comment block describing the function. The beginning of a function must start between two and three lines after the end of the previous function.

Functions should be made to fit on one page. There are few instances where it makes sense to have functions spanning pages and pages. In those rare instances that this is absolutely necessary, page breaks should occur in logical positions within the function.

More than one small function could share a page by separating the function declarations by at least 2 spaces. Functions must also start on page boundaries.

Functions that are only used within the file should be declared **static** to hide them from other functions in different files.

10.00 Initialized Tables

Line up similar fields in a table.

Very often, it's useful to create tables that are based on data structures as shown in the example below. You should note that the initialized fields are neatly organized in columns. This makes the code quite easy to read. You should also not limit yourself to a column width for such a situation since it doesn't make sense to wrap table entries at the beginning of the next line.

typedef struct {						
INT16U	ParamNbr;					
void	*ParamAddr;					
void	*ParamMin;					
void	*ParamMax;					
void	*ParamDflt;					
void	*ParamInc;					
} PARAM;						

	<pre>RPM_ParamTb1[] = { *ParamAddr</pre>	*ParamM	in *Paraml	Max *ParamDf]	lt *ParamInc
{ 2981,	(void *) & RPM,	(void *)&FP0,	(void *)&FP0,	(void *)&FP0,	(void *)&FP0},
{ 2982,	(void *) & RPMAvg,	(void *)&FP0,	(void *)&FP0,	(void *)&FP0,	(void *)&FP0, },
{ 2984,	(void *) & RPMFilterC	onst, (void *)	&FP0, (void	*)&FP1, (void	*)&FP0Pt1, (void
*)&FP0Pt01},					
{ 2985,	(void *) & RPMTeeth,	(void *)&W4,	(void *) & W500,	(void *)&W60,	(void *)&W1},
};					

const UWORD RPM ParamTblSize = sizeof(RPM ParamTbl) / sizeof(PARAM);

11.00 Global Variables

Only place global variable declarations in the .H file.

As you know, a global variable needs to be allocated storage space in RAM and must be referenced by other modules using the C keyword <code>extern</code>. Declarations must thus be placed in both the .C and the .H files. This duplication of declarations, however, can lead to mistakes. The technique described below only requires that the declaration be done in one place, the .H file.

In all .H files that define global variables, you will need to add the following code:
#ifdef xxx_GLOBALS
#define xxx_EXT
#else
#define xxx_EXT extern
#endif

Each variable that needs to be declared global will be prefixed with xxx_EXT in the .H file as shown below. 'xxx' represents a prefix identifying the module name.

```
xxx_EXT INT16U ParamVars;
xxx_EXT FP32 ParamMaxVal;
```

The module's . c file will contain the following declaration:

#define xxx_GLOBALS
#include ``INCLUDES.H"

The module for which the global variables belongs to will 'allocate' storage for the variables while all the other modules that simply includes the .H file will see <code>extern</code> statements in front of those variables.

Micriµm, Inc. ^{C Coding Standard}

References

μC/OS-II, The Real-Time Kernel, 2nd Edition Jean J. Labrosse R&D Technical Books, 2002 ISBN 1-57820-103-9

Embedded Systems Building Blocks

Jean J. Labrosse R&D Technical Books, 2000 ISBN 0-87930-604-1

C Style: Standards and Guidelines David Straker Prentice Hall, 1992 ISBN 0-13-116898-3

Contacts

Micriµm, Inc. 949 Crestview Circle Weston, FL 33327 954-217-2036 954-217-2037 (FAX) e-mail: Jean.Labrosse@Micrium.com WEB: www.Micrium.com

R&D Books, Inc. 1601 W. 23rd St., Suite 200 Lawrence, KS 66046-9950 (785) 841-1631 (785) 841-2624 (FAX) WEB: <u>http://www.rdbooks.com</u> e-mail: <u>rdorders@rdbooks.com</u>