Micriμm

Empowering Embedded Systems

C Coding Standard

Application Note
AN-2000 Rev. B

www.Micrium.com

Disclaimer

Specifications written in this manual are believed to be accurate, but are not guaranteed to be entirely free of error. Specifications in this manual may be changed for functional or performance improvements without notice. Please make sure your manual is the latest edition. While the information herein is assumed to be accurate, Micrium assumes no responsibility for any errors or omissions and makes no warranties. Micrium specifically disclaims any implied warranty of fitness for a particular purpose.

Copyright notice

Trademarks

Names mentioned in this manual may be trademarks of their respective companies. Brand and product names are trademarks or registered trademarks of their respective holders.

Contact address

Micrium
949 Crestview Circle
Weston, FL 33327-1848
U.S.A.
Phone : +1 954 217 2036
FAX : +1 954 217 2037
WEB : **www.micrium.com**
Email : **support@micrium.com**

Manual versions

If you find any errors in this document, please inform us and we will make the appropriate corrections for future releases.

Manual Version	Date	By	Description
Version B	2008/09/01	BAN/ITJ	Updated.
Version A	2003/09/11	JJL	First version.

Table Of Contents

Introduction

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

The chosen style should be used consistently throughout all your projects. Furthermore, a single style should 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 helps avoid code rewrites.

The conventions for Micrium software are outlined in this document. These standards should be followed for all new software modules, ports, BSPs and applications.

I.1 Basic Principles

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. To achieve this, a few principles should be followed:

- **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.

- **Comply with ANSI C standards**. For portability, code should comply with an accepted release of the C standard. ISO/IEC 9899:TC2. Furthermore, the safety-aware spirit of the Motor Industry Software Reliability Association's *Guidelines for the Use of the C Language in Vehicle-Based Software* (MISRA rules) should guide code development. MISRA rules will be cited throughout this document when practices are recommended.

- **Keep the code simple**.
- **Be explicit**. Avoid implicit or obscure features of the language. Say what you mean.
- **Be consistent**. Use the same rules as much as possible.
- **Avoid complicated statements**. Statements comprising many decision points are hard to follow and test.
- **Update 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.

Chapter 1

Source Files

1.01 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 (if you need to print the code) on an 8.5" by 11" page using a reasonable font size. Using 7-pt Arial, 132 characters (in portrait mode) can be accommodated while leaving enough room on the left side of the page for holes for insertion in a three ring binder.

A line-width of 132 characters prevents needing 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 neither see nor print all the elements of these tables at once, but at least the different fields line up neatly.

1.02 File Heading.

A comment block must be placed at the beginning of each source code file (both code and header files) containing the module name and description, copyright, copyright or distribution terms, file description, file name, module version, programmer initial(s) and note(s). This information should be structured as shown in Figure 1-1. The note(s) section should be omitted if no notes are given.

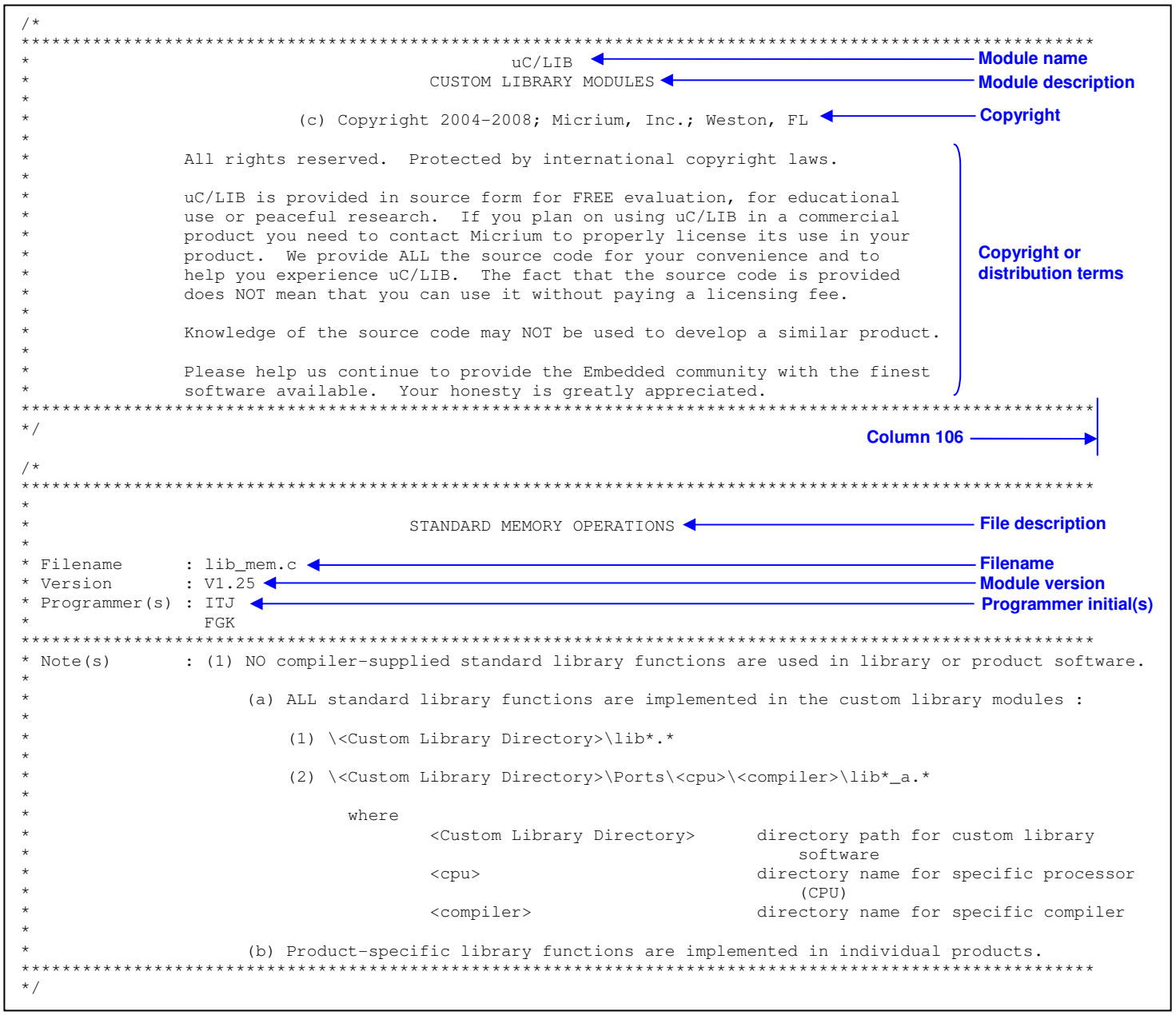


Figure 1-1. Example File Header.

1.03 Implementation (Code) File Layout.

After the file header, every code file must contain the following sections in the following order:

- `INCLUDE FILES`
- `LOCAL DEFINES`
- `LOCAL CONSTANTS`
- `LOCAL DATA TYPES`
- `LOCAL TABLES`
- `LOCAL GLOBAL VARIABLES`
- `LOCAL FUNCTION PROTOTYPES`
- `LOCAL CONFIGURATION ERRORS`

Each section must be preceded by a comment block; see Listing 1-2. The contents of a section must be followed by two blank lines. Even if a section is empty, its comment block must be included in the file and followed by two blank lines. These sections may have additional sub-sections.

```
/*
*********************************************************************************************************
*                                              SECTION NAME
*********************************************************************************************************
*/
```

Listing 1-2. Comment Block Format.

Functions definitions must follow these sections in the following order:

- Global functions.
- Local functions.

1.03.01 `INCLUDE FILES` Section.

The `INCLUDE FILES` section should include all necessary header files. If the code file has a matching header file, it may be most convenient to include that header file, as is done in Listing 1-3; the matching header file will include the header files for external modules or other internal header files, as appropriate. See Section 1.04.02.

```
/*
*********************************************************************************************************
*                                             INCLUDE FILES
*********************************************************************************************************
*/

#define    LIB_MEM_MODULE
#include  <lib_mem.h>
```

Listing 1-3. `INCLUDE FILES` Section.

1.03.02 Functions.

After the configuration errors in the `LOCAL CONFIGURATION ERRORS` section, function definitions must be given. All definitions of global functions must come first, followed by definitions of local functions. The local functions shall be separated from the global functions using the comment block shown in Listing 1-4.

```
/*
*********************************************************************************************************
*********************************************************************************************************
*                                            LOCAL FUNCTIONS
*********************************************************************************************************
*********************************************************************************************************
*/
```

Listing 1-4. `LOCAL FUNCTIONS` Comment Block Format.

Within the global functions section and the local functions section, functions shall be ordered in the order the prototypes are given (in the `LOCAL FUNCTION PROTOTYPES` section of the code file and the `FUNCTION PROTOTYPES` section of the header file, respectively). The order of the prototypes is not specified, though two orderings are suggested:

- Alphabetical order
- Functional order

1.04 Header File Layout.

After the file header, every header file must contain the following sections in the following order:

- `MODULE`
- `INCLUDE FILES`
- `EXTERNS` (if a matching code file exists)
- `DEFAULT CONFIGURATION` (if necessary)
- `DEFINES`
- `DATA TYPES`
- `GLOBAL VARIABLES`
- `MACRO'S`
- `FUNCTION PROTOTYPES`
- `CONFIGURATION ERRORS`
- `MODULE END`

Each section must be preceded by a comment block; see Listing 1-2. The contents of a section should be followed by two blank lines. Even if a section is empty, its comment block must be included in the file and followed by two blank lines. These sections may have additional sub-sections.

1.04.01 MODULE and MODULE END Sections.

Header files shall be guarded from duplicate inclusion by testing for the definition of a value. The `MODULE` performs the test (`#ifndef`) and definition (`#define`). The matching `#endif` for the test is located in the `MODULE END` section.

```
/*
*********************************************************************************************************
*                                                MODULE
*********************************************************************************************************
*/

#ifndef  LIB_MEM_MODULE_PRESENT
#define  LIB_MEM_MODULE_PRESENT

                .      .      .      .      .      .      .      .      .      .      .
                .      .      .      .      .      .      .      .      .      .      .
                .      .      .      .      .      .      .      .      .      .      .

/*
*********************************************************************************************************
*                                             MODULE END
*********************************************************************************************************
*/

#endif                                                  /* End of lib mem module include.               */
```

Listing 1-5. MODULE and MODULE END Sections.

1.04.02 INCLUDE FILES Section.

The header file should include all necessary base files, internal files or external module includes files. These includes should be placed in the `INCLUDE FILES` section.

```
/*
*********************************************************************************************************
*                                            INCLUDE FILES
*********************************************************************************************************
*/

#include  <cpu.h>
#include  <lib_def.h>
#include  <app_cfg.h>
```

Listing 1-6. MODULE and MODULE END Sections.

1.04.03 EXTERNS and GLOBAL VARIABLES Sections.

A global variable needs to be allocated storage space in RAM and must be referenced in other modules using the C keyword `extern`. Consequently, declarations must be placed in both the code (*.*c*) and header (*.*h*) files, possibly leading to mistakes. The `EXTERNS` sections eliminates this source of error, so that declarations need only be done in the header (*.*h*) file. The code, as shown in Listing 1-7, conditionally defines a `xxxx_EXT` constant (in this example, `LIB_MEM_EXT`) that must prefix the declaration of global variables, as shown in Listing 1-8. The matching code (*.c) file should contain the `#define` of `xxxx_MODULE` (in this example, `LIB_MEM_MODULE`); see Listing ---.

```
/*
*********************************************************************************************************
*                                                EXTERNS
*********************************************************************************************************
*/

#ifdef   LIB_MEM_MODULE
#define  LIB_MEM_EXT
#else
#define  LIB_MEM_EXT  extern
#endif
```

Listing 1-7. EXTERNS Section.

```
/*
*********************************************************************************************************
*                                          GLOBAL VARIABLES
*********************************************************************************************************
*/

LIB_MEM_EXT  CPU_INT08U  Mem_ExampleGlobal;
```

Listing 1-8. GLOBAL VARIABLES Section.

1.04.04 CONFIGURATION ERRORS Section.

`#error` should be used to flag missing `#define` constant or macros and to check for invalid values. The `#error` directive will cause the compiler to display the message with the double quotes when the condition is not met.

```
/*
*********************************************************************************************************
*                                          CONFIGURATION ERRORS
*********************************************************************************************************
*/

#ifndef  LIB_MEM_CFG_ARG_CHK_EXT_EN
#error  "LIB_MEM_CFG_ARG_CHK_EXT_EN        not #define'd in 'app_cfg.h'"
#error  "                             [MUST be  DEF_DISABLED]           "
#error  "                             [     ||  DEF_ENABLED ]           "

#elif  ((LIB_MEM_CFG_ARG_CHK_EXT_EN != DEF_DISABLED) && \
         (LIB_MEM_CFG_ARG_CHK_EXT_EN != DEF_ENABLED ))
#error  "LIB_MEM_CFG_ARG_CHK_EXT_EN  illegally #define'd in 'app_cfg.h'"
#error  "                             [MUST be  DEF_DISABLED]           "
#error  "                             [     ||  DEF_ENABLED ]           "
#endif



#ifndef  LIB_MEM_CFG_POOL_EN
#error  "LIB_MEM_CFG_POOL_EN               not #define'd in 'app_cfg.h'"
#error  "                             [MUST be  DEF_DISABLED]           "
#error  "                             [     ||  DEF_ENABLED ]           "

#elif  ((LIB_MEM_CFG_POOL_EN != DEF_DISABLED) && \
         (LIB_MEM_CFG_POOL_EN != DEF_ENABLED ))
#error  "LIB_MEM_CFG_POOL_EN         illegally #define'd in 'app_cfg.h'"
#error  "                             [MUST be  DEF_DISABLED]           "
#error  "                             [     ||  DEF_ENABLED ]           "


#elif   (LIB_MEM_CFG_POOL_EN == DEF_ENABLED)

#ifndef  LIB_MEM_CFG_HEAP_SIZE
#error  "LIB_MEM_CFG_HEAP_SIZE             not #define'd in 'app_cfg.h'"
#error  "                             [MUST be  > 0]                    "

#elif   (LIB_MEM_CFG_HEAP_SIZE < 1)

#error  "LIB_MEM_CFG_HEAP_SIZE       illegally #define'd in 'app_cfg.h'"
#error  "                             [MUST be  > 0]                    "
#endif

#endif
```

Listing 1-9. CONFIGURATION ERRORS Section.

Chapter 2

Source Code

2.01 Tab Character

Tab characters (ASCII character `0x09`) must not be used. Indentation must be done using the space characters only (ASCII character `0x20`). Tab characters expand differently of different computers and printers. Avoiding them ensures that the intended spacing is maintained.

2.02 Indentation

Indentation of code will consist of 4 spaces, except statements under a switch/case statement, which are indented by 5 spaces. Always try to start on multiples of 4 spaces (columns 1, 5, 9, 13, 17 …). See Listings 2.1, 2.2 and 2.3.

2.03 Line Feeds

For enhanced clarity, code blocks can be separated with line feeds. A long function is often logically divided into a set of major blocks, each of which is divided in smaller sub-blocks that may be further subdivided. From the smallest blocks to the largest, an increasing quantity of line feeds should be used to convey the nature of the hierarchy. Neighboring or nested control statements may also benefit from line feeds inserted between statements. Listing 2-1 demonstrates this concept.

```
 if (data_size < 1) {                                   /* ------------ HANDLE NULL-SIZE DATA PKT ------------ */
    *psum_err = NET_UTIL_16_BIT_SUM_ERR_NULL_SIZE;

     if (prev_octet_valid != DEF_NO) {                  /* If null size & last octet from prev pkt buf avail ..*/
\n
Enhance separation between clauses of if & else statements
         if (last_pkt_buf != DEF_NO) {                  /* ...   & on last pkt buf,              ...           */
             sum_val_32   = (CPU_INT32U)*poctet_prev;/* ...   cast prev pkt buf's last octet, ...           */
             sum_val_32 <<=  DEF_OCTET_NBR_BITS;        /* ... pad odd-len pkt len (see Note #5) ...           */
             sum_32       =  sum_val_32;                /* ...  & rtn prev pkt buf's last octet as last sum.   */
\n
         } else {                                       /* ... & NOT on last pkt buf, ...                      */
            *poctet_last = *poctet_prev;                /* ... rtn last octet from prev pkt buf as last octet. */
             DEF_BIT_SET(*psum_err, NET_UTIL_16_BIT_SUM_ERR_LAST_OCTET);
\n
         }
\n
     } else {
         ;                                              /* If null size & NO prev octet, NO action(s) req'd.   */
     }

     return (sum_32);                                   /* Rtn 16-bit sum (see Note #5c1).                     */
 }
\n
\n
\n                                                      /* ------------ HANDLE NON-NULL DATA PKT ------------- */
 size_rem       =  data_size;
*psum_err       =  NET_UTIL_16_BIT_SUM_ERR_NONE;

                                                        /* See Notes #3 & #4.                                  */
 modulo_16      = (CPU_INT08U)((CPU_ADDR)p_data % sizeof(CPU_INT16U));
 pkt_aligned_16 = (((modulo_16 == 0) && (prev_octet_valid == DEF_NO )) ||
                   ((modulo_16 != 0) && (prev_octet_valid == DEF_YES))) ? DEF_YES : DEF_NO;
\n
\n
 pdata_08 = (CPU_INT08U *)p_data;
 if (prev_octet_valid == DEF_YES) {                     /* If last octet from prev pkt buf avail,   ...        */
     sum_val_32   = (CPU_INT32U)*poctet_prev;
     sum_val_32 <<=  DEF_OCTET_NBR_BITS;                /* ... prepend last octet from prev pkt buf ...        */
\n
     sum_val_32  += (CPU_INT32U)*pdata_08++;
     sum_32      += (CPU_INT32U) sum_val_32;            /* ... to first octet in cur pkt buf.                  */
\n
     size_rem    -=  sizeof(CPU_INT08U);
 }

 if (pkt_aligned_16 == DEF_YES) {                       /* If pkt data aligned on 16-bit boundary, ..          */
                                                        /* .. calc sum with 16- & 32-bit data words.           */
        pdata_16  = (CPU_INT16U *)pdata_08;

        .
        .
        .
```

Three line feeds between these major blocks

Two line feeds between these blocks

Separate steps in calculation

Separate steps in calculation

Listing 2-1. Line Feed Example.

2.04 Horizontal Alignment

Equal signs for a code block must vertically align two columns after the longest left-hand side expression for the code block; i.e., the longest left-hand side expression should be followed by a single space and the equals sign.

Left-hand side expressions must vertically align their left-most alphanumeric characters on an indentation column of 4 where any preceding characters (asterisks, ampersands, parentheses) will align in the column immediately preceding the indentation column.

Listings 2-2 demonstrate these concepts.

```
  datum        = (datum      >> 16) ^ datum;
  datum_temp   = (datum      >>  4) ^ datum;
  datum_temp   = (datum_temp >>  2) ^ datum_temp;
  datum_temp   = (datum_temp >>  1) ^ datum_temp;
  hamming     |=  datum_temp        & DEF_BIT_00;
  hamming    <<= 1;                                                    (a)
```

```
  max_seg_size   = *popt;
  max_seg_size <<=  DEF_OCTET_NBR_BITS;
 *perr           =  NET_TCP_ERR_NONE;                                  (b)
```

```
  addr_tbl_qty = *paddr_tbl_qty;
*paddr_tbl_qty =  0;                                                   (c)
```

Listing 2-2. Code Alignment Examples.

The least-significant portion of integers and floating-point numbers should be aligned. Listing 2-3 demonstrates this concept.

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

Listing 2-3. Numeric Alignment Example.

2.05 Horizontal Spacing

The unary operators are written with no space between the operator and the operand:

```
!value
~bits
++i
j--
(CPU_INT32U)x
*ptr
&x
sizeof(x)
```

The binary operators (and the ternary operator) are written with at least on space between the operator and operands:

```
c1 = c2;
x + y
i += 2;
n > 0 ? n : -n
a < b
c >= 2
```

At least one space is needed after each semicolon:

```
for (i = 0; i < 10; i++)
```

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

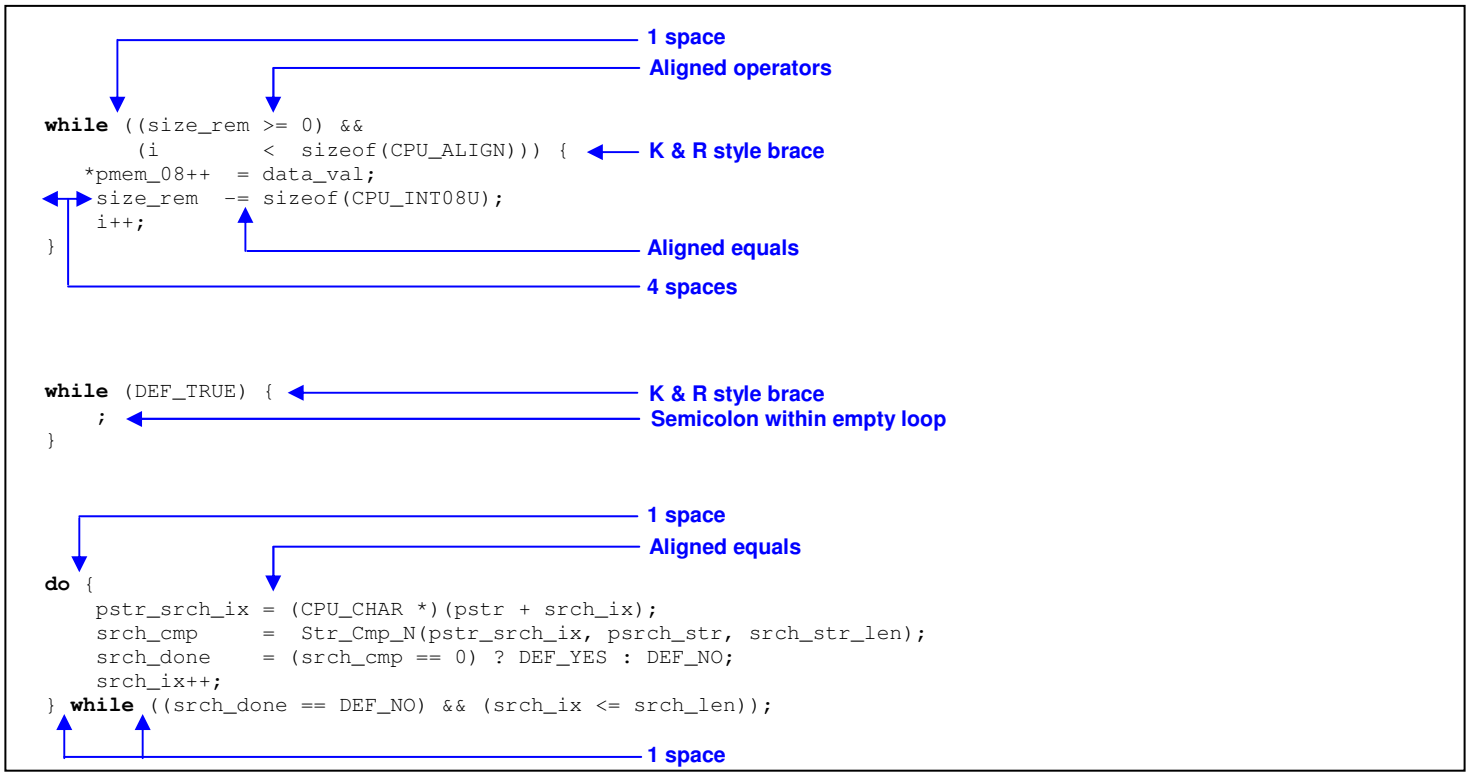
```
if (a > b)
while (x > 0)
for (i = 0; i < 10; i++)
} else {
switch (x)
return (y)
```

Expressions within parentheses are written with no space after the opening parenthesis and no space before the closing parenthesis:

```
x = (a + b) * c;
```

2.06 `while` and `do…while` Loops

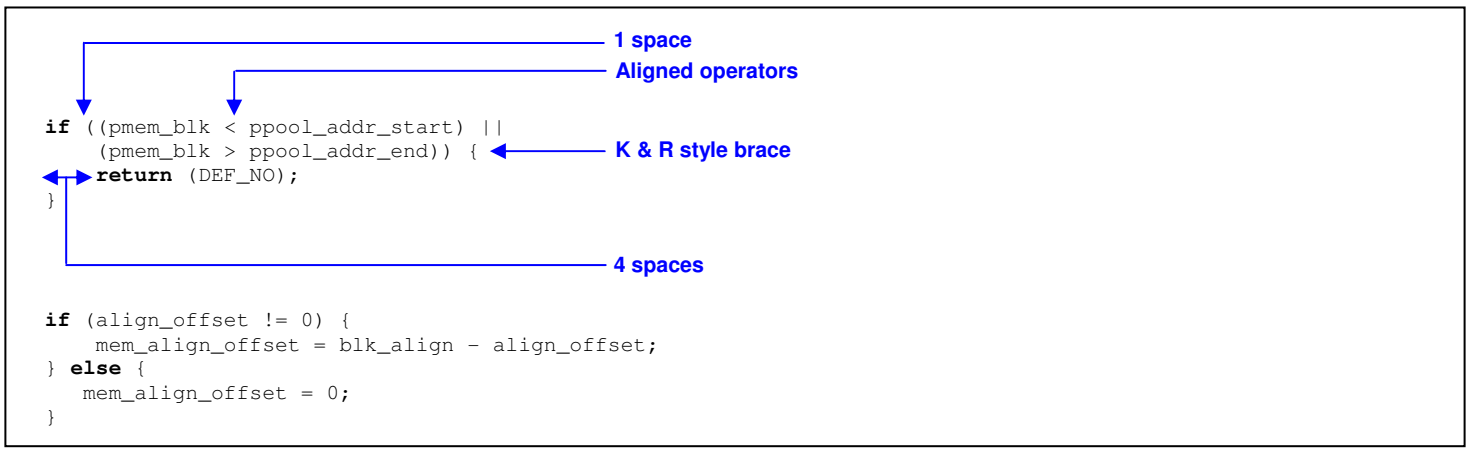
`while` and `do…while` loops shall always be formatted as shown in Figure 2-4. The body must be enclosed in braces (MISRA 59), even for an empty loop.



Listing 2-4. `while` and `do…while` Loop Formats.

2.07 `if` and `if...else` Statements

`if` and `if…else` statements shall always be formatted as shown in Figure 2-5. The body must be enclosed in braces (MISRA 59).



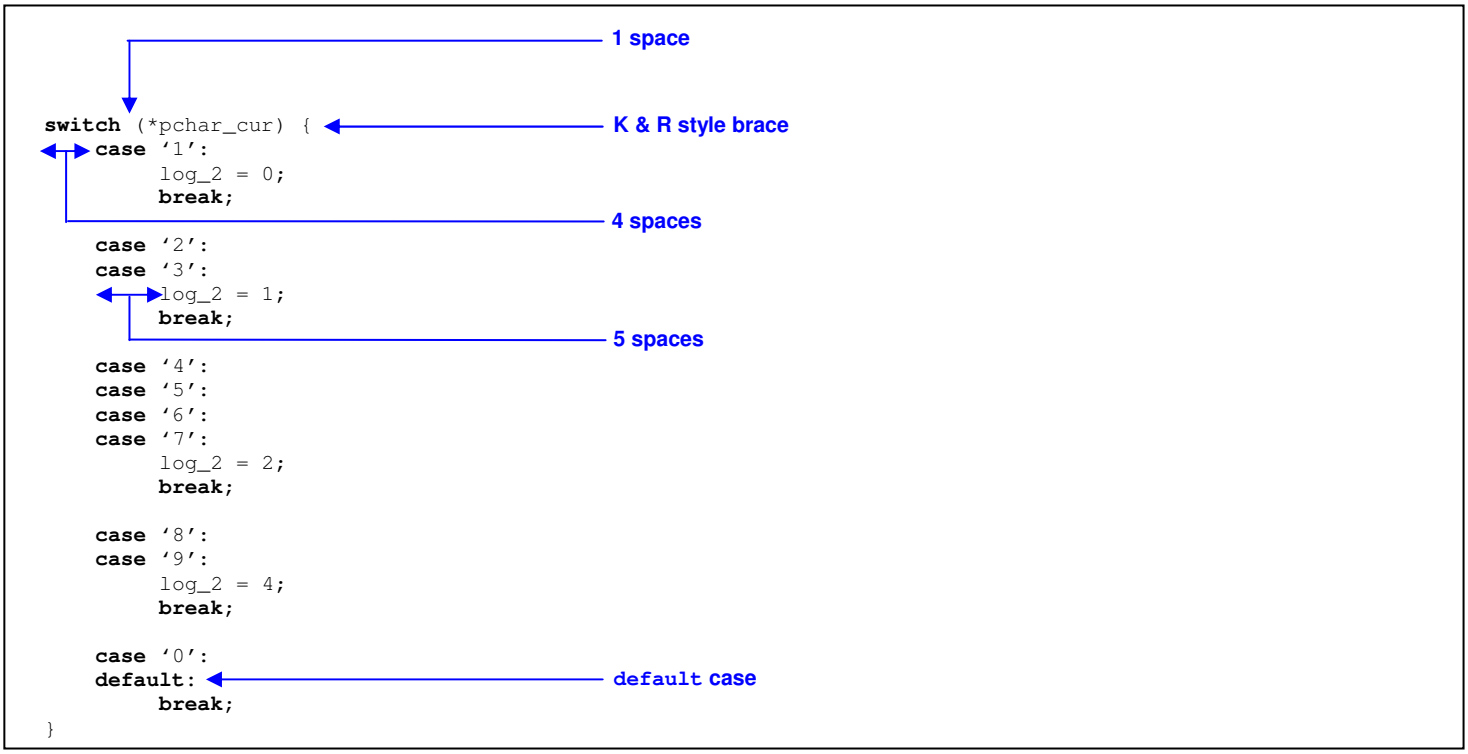

```
if ((pmem_blk < ppool_addr_start) ||
    (pmem_blk > ppool_addr_end)) {
   return (DEF_NO);
}


if (align_offset != 0) {
    mem_align_offset = blk_align - align_offset;
} else {
   mem_align_offset = 0;
}
```

Listing 2-5. `if` and `if…else` Statement Formats.

2.08 `switch` Statements

`switch` conditions shall always be formatted as shown in Figure 2-6.




```
switch (*pchar_cur) {
    case '1':
         log_2 = 0;
         break;

    case '2':
    case '3':
         log_2 = 1;
         break;

    case '4':
    case '5':
    case '6':
    case '7':
         log_2 = 2;
         break;

    case '8':
    case '9':
         log_2 = 4;
         break;

    case '0':
    default:
         break;
}
```

Listing 2-6. `switch` Statement Format.

Every non-empty case shall be terminated by a `break` statement (MISRA 61) or a `return`. A `switch` statement shall always have at least one case (MISRA 64) in addition to a final `default` case (MISRA 62). A switch statement should not be used for a Boolean expression (MISRA 63).

2.09 #defineS

`#define`s constants should be aligned, when possible, and suffixed, where appropriate. Example `#define`s are shown in Figure 2-7.

```
                                                          Right-align rightmost digit of constants;
                                                          column 60 is a recommended alignment
                                                          column.
#define  DEF_BIT_00                                 0x00
#define  DEF_BIT_08                               0x0100
#define  DEF_BIT_16                           0x00010000

#define  DEF_INT_08U_MIN_VAL                         0u
#define  DEF_INT_08U_MAX_VAL                       255u   <-- Use type suffix, where appropriate.

#define  DEF_INT_08S_MIN_VAL                      -128
#define  DEF_INT_08S_MAX_VAL                       127

#define  DEF_TIME_NBR_HR_PER_DAY                    24uL
                                                          Align operators.
#define  DEF_TIME_NBR_MIN_PER_HR                    60uL
#define  DEF_TIME_NBR_MIN_PER_DAY                  (DEF_TIME_NBR_MIN_PER_HR  * DEF_TIME_NBR_HR_PER_DAY)

#define  DEF_TIME_NBR_SEC_PER_MIN                   60uL
#define  DEF_TIME_NBR_SEC_PER_HR                   (DEF_TIME_NBR_SEC_PER_MIN * DEF_TIME_NBR_MIN_PER_HR)
#define  DEF_TIME_NBR_SEC_PER_DAY                  (DEF_TIME_NBR_SEC_PER_HR  * DEF_TIME_NBR_HR_PER_DAY)
```

Listing 2-7. Example `#define`s.

2.10 break and continue

When possible, `break` (outside of `switch` statements) and `continue` should be avoided (MISRA 58, 57). A 'flag' variable, modified in the loop body and checked in the loop condition, is preferred to `break`ing from the loop. For example, Listing 2-8a is recommended over Listing 2-8b.

```
found = DEF_NO;

while (found == DEF_NO) {
    .
    .
    .


    if (...) {
       found = DEF_YES;
    }
}
```

(a)

```
while (DEF_TRUE) {
    .
    .
    .


    if (...) {
       break;
    }
}
```

(b)

Listing 2-8. Loop Flag.
The use of a loop flag (a, left) is preferred to `break`ing from the loop (b, right).

2.11 Labels and `goto`

Labels should not be used (except in `switch` statements) (MISRA 55).

`goto` should NOT be used (MISRA 56).

2.12 Code Lines

In general, there should only be one action per line of code.

```
DispSegTblIx =    0;
DispDigMsk   = 0x80;
```

rather than

```
DispSegTblIx = 0; DispDigMsk = 0x80;
```

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

rather than

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

2.13 Explicit Comparisons

Explicit comparisons should always be made. Listing 2-9(a) compares the Boolean variables directly against `DEF_YES`, resulting in more obvious (and demonstrably correct) behavior than Listing 2-9(b).

```
addr_conflict = ((sender_protocol_verifd == DEF_YES) &&
                 (sender_hw_verifd       != DEF_YES)) ? DEF_YES : DEF_NO;
```
(a)

```
addr_conflict = sender_protocol_verifd && !sender_hw_verifd;
```
(b)

Listing 2-9. Explicit Comparison Example.
Listing (a, top) uses explicit tests and is preferred to Listing (b, bottom).

2.14 Configuration

Configuration `#define`s should always be explicit; preprocessor tests using configuration `#define`s should always test values explicitly. For example, `LIB_MEM_CFG_POOL_EN` should be configured `DEF_DISABLED` or `DEF_ENABLED`, as directed in Listing 2-10(a). In the `CONFIGURATION ERRORS` section of the module header file (Listing 2-10(b)), the definition of `LIB_MEM_CFG_POOL_EN` by explicitly testing against `DEF_DISABLED` and `DEF_ENABLED`, making it impossible to configure a different value. In the module's code file (Listing 2-10(c)), the inclusion of the functions `Mem_PoolSegCalcTotSize()` and `Mem_PoolSegAlloc()` is based on an explicit test of `LIB_MEM_CFG_POOL_EN`.

```
/*
*********************************************************************************************************
*                                       MEMORY POOL CONFIGURATION
*
* Note(s) : (1) Configure LIB_MEM_CFG_POOL_EN to enable/disable memory pool functions.
*********************************************************************************************************
*/
                                                        /* Configure memory pool feature (see Note #1) :        */
#define  LIB_MEM_CFG_POOL_EN    DEF_DISABLED
                                                        /*   DEF_DISABLED     Memory pool(s) DISABLED           */
                                                        /*   DEF_ENABLED      Memory pool(s) ENABLED            */
```

(a)

```
/*
*********************************************************************************************************
*                                        CONFIGURATION ERRORS
*********************************************************************************************************
*/

#ifndef  LIB_MEM_CFG_POOL_EN
#error  "LIB_MEM_CFG_POOL_EN               not #define'd in 'app_cfg.h'"
#error  "                             [MUST be  DEF_DISABLED]           "
#error  "                             [     ||  DEF_ENABLED ]           "

#elif  ((LIB_MEM_CFG_POOL_EN != DEF_DISABLED) && \
         (LIB_MEM_CFG_POOL_EN != DEF_ENABLED ))
#error  "LIB_MEM_CFG_POOL_EN          illegally #define'd in 'app_cfg.h'"
#error  "                             [MUST be  DEF_DISABLED]           "
#error  "                             [     ||  DEF_ENABLED ]           "

#endif
```

(b)

```
/*
*********************************************************************************************************
*                                       LOCAL FUNCTION PROTOTYPES
*********************************************************************************************************
*/

#if (LIB_MEM_CFG_POOL_EN == DEF_ENABLED)                        /* ---------- MEM POOL FNCTS ---------- */

static  CPU_SIZE_T    Mem_PoolSegCalcTotSize(void        *pmem_addr,
                                             CPU_SIZE_T   blk_nbr,
                                             CPU_SIZE_T   blk_size,
                                             CPU_SIZE_T   blk_align);

static  void         *Mem_PoolSegAlloc      (MEM_POOL    *pmem_pool,
                                             CPU_SIZE_T   size,
                                             CPU_SIZE_T   align);
#endif
```

(c)

Listing 2-10. Explicit Configuration/Configuration Test Example.

Chapter 3

Commenting

Comments allow a programmer to communicate details about implementation and guide a reader through code.

3.01 Style

Only C-style comments should be used:

```
/* This is a C-style comment. */
```

C++-style comments should NEVER be used:

```
// This is a C++-style comment.
```

Multi-line comments with a single comment terminator should NOT be used:

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

Comments should not be nested (MISRA 9).

3.02 What to Put in a Comment

Make every comment count. Get to the point. Do not state what a decent programmer would read from the code. Explain what the code does from a 'high-level' standpoint.

Do NOT use …

- **'Emotions' or profanity**. For example, do NOT use comments such as "Let's make this one big happy structure!"
- **The product name**. Consistently use a generic referent for the module ("OS" rather than "µC/OS-II", "network protocol suite" rather than "µC/TCP-IP").

DO use …

- **Structured sentences** (as much as possible).
- **Uppercase words** to emphasize meaning.
- **Acronyms, abbreviations and mnemonics**, as long as the audience will understand the meaning.

3.03 Where to Place Comments

Minimize comments embedded among statements. NEVER start comments immediately above code blocks, as shown in Listing 2-1. This makes the code difficult to follow because the comments distract the visual scanning of the code.

```
void  ClkUpdateTime (void)
{                                                       /* DO NOT comment like this!         */
    /* 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++;
    }
}
```

Listing 3-1. Avoid Comments Interleaved with Code.

As much as possible, use trailing comments, all starting at the same column. The terminating comment characters should line up.

```
void ClkUpdateTime (void)
{
    if (ClkSec >= CLK_MAX_SEC) {                    /* Update the seconds                */
        ClkSec = 0;
        if (ClkMin >= CLK_MAX_MIN) {                /* Update the minutes                */
            ClkMin = 0;
            if (ClkHour >= CLK_MAX_HOURS) {         /* Update the hours                  */
                ClkHour = 0;
            } else {
                ClkHour++;
            }
        } else {
            ClkMin++;
        }
    } else {
        ClkSec++;
    }
}
```

Listing 3-2 Use Trailing Comments Whenever Possible.

3.04 Special Comments

Special comments indicate the presence of known bugs, legacy implementation and future issues to address. The comment markers shown in Listing 2-3 are recommended.

```
/* ???? Question(s) regarding implementation or design specification.  */

/* $$$$ Future function that needs to be implemented.                  */

/* @@@@ Old code to leave as-is because ....                          */

/* #### Technical issue not (satisfactorily) resolved.                */
```

Listing 3-3. Special Comments.

3.05 Notes

The header of each function has a `Note(s)` section that may contain an ordered, hierarchical list of notes (see Section 6.03). If an extended explanation or gloss on a line or block of code is necessary, the text should be placed into a note and a specific reference made in the comment for the line or block. If the note so referred to is in the comment header of the same function, the reference can be formatted:

```
        x = 20;                 /* See Note #1a3B.                                  */
```

If the note is in the comment header of a different function, section, or file; the reference should be formatted:

```
        x = 20;                 /* See 'Mem_Copy()  Note #1a3B'.                    */
        y = 0x00;               /* See 'lib_mem.h  MEMORY MACRO's  Note #2b1'.      */
```

3.06 Commenting Out Code

To avoid nesting comments, use `#if 0` and `#endif` to 'comment out' large portions of code.

Commented-out code should only be released if an explanation is provided (MISRA 10).

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

Listing 3-4. 'Commenting Out' Code.

Chapter 4

Naming Convention

Table 3-1 summarizes the naming conventions detailed in Sections 4.04 through 4.08.

Category	Format	Example

`#define` constant,
`#define` macro,
`typedef`,
`enum` tag
	All uppercase.
	Words separated by an underscore ('_').
		`DEF_OCTET_NBR_BITS`
		`DEF_BIT_IS_SET()`
		`CLK_DATE_TIME`

Local (function-scope) variable	All lowercase. Words separated by an underscore ('_').	`i` `p_tcb`
File-scope variable, Global variable	Prefixed with the module identifier followed by an underscore ('_'). Each component word starts with an initial capital. File-scope variables should be declared static.	`Clk_TimeStamp` `Clk_TimeZoneOffset`
Local function, Global function	Prefixed with the module identifier followed by an underscore ('_'). Each component word starts with an initial capital. Global variables should be declared static.	`Clk_IsLeapYear()` `Clk_SetTS()`
`struct` members	Each component word starts with an initial capital.	`CLK_DATE_TIME.DayOfWeek`

Table 4-1. Summary of Naming Convention.

4.01 Length

Identifiers should not have more than 31 significant characters (MISRA 11).

4.02 Uniqueness

Module names should be unique, and each file-scope or global identifier should be unique (MISRA 12). Local (function-scope) variables should not use the same name as (or 'hide') a file-scope or global identifier.

4.03 Appropriateness

For all categories of identifier, names should be chosen appropriate to the usage context. Names of pets, children or household items should never be used.

4.04 `#define`s, `typedef`s and `enum` Tags

`#define`s, `typedef`s and `enum` tags should be prefixed by the module identifier followed by an underscore ('_'). The identifier itself should be all uppercase, with component words separated by an underscore. For example:

```
DEF_OCTET_NBR_BITS
DEF_BIT_IS_SET()
CLK_DATE_TIME
```

4.05 Local (Function-Scope) Variables

Local (function-scope) variables should be all lowercase, with component words separated by an underscore.

```
CPU_SIZE_T    size_tot;          Total size
CPU_SIZE_T    size_tot_ptrs;     Total size of memory pool pointers
CPU_SIZE_T    i;                 Loop counter variable
```

Standard loop counter variables (`i`, `j`, `k` …) should be used.

If a variable is a pointer to another variable or a memory location, it should be prefixed by a '`p_`':

```
MEM_POOL      *p_mem_pool_heap;    Pointer to heap memory pool struct.
void         **p_pool_ptr;         Pointer to pool pointer.
CPU_INT08U    *p_mem_addr_ptrs;    Pointer to memory address pointers.
```

4.06 File-scope and Global Variables

File-scope and global variables should be prefixed by the module identifier followed by an underscore ('_'). The identifier itself should be camel-case, with each component word starting with an initial capital.

```
MEM_POOL      Mem_PoolHeap;                          Total size
CPU_INT08U    Mem_Heap[LIB_MEM_CFG_HEAP_SIZE];       Size of memory pool pointers
```

If a component word is an acronym, abbreviation or mnemonic that should be all-caps (see section 3.09), then that word should be in uppercase and followed by an underscore ('_').

If a variable is a pointer to another variable or a memory location, it should be suffixed by '`Ptr`'.

4.07 File-scope and Global Functions

File-scope and global functions should be prefixed by the module identifier followed by an underscore ('_'). The identifier itself should be camel-case, with each component word starting with an initial capital.

```
static  CPU_BOOLEAN  Clk_DateTimeTest (CLK_DATE_TIME  *date_time);
static  void         Clk_ComputeDOW   (CLK_DATE_TIME  *date_time);
static  CPU_BOOLEAN  Clk_IsLeapYear   (CPU_INT16U      year);
```

If a component word is an acronym, abbreviation or mnemonic that should be all-caps (see section 3.09), then that word should be in uppercase and followed by an underscore ('_').

4.08 `struct` Members

File-scope and global functions should be prefixed by the module identifier followed by an underscore ('_'). The identifier itself should be camel-case, with each component word starting with an initial capital.

```
typedef  struct  clk_date_time {
    CPU_INT16U     Year;
    CPU_INT08U     Month;
    CPU_INT08U     Day;
    CPU_INT08U     DayOfWeek;
    CPU_INT08U     Hour;
    CPU_INT08U     Minute;
    CPU_INT08U     Second;
    CLK_TZ_OFFSET  TZ_Offset;
}  CLK_DATE_TIME;
```

If a component word is an acronym, abbreviation or mnemonic that should be all-caps (see section 3.09), then that word should be in uppercase and followed by an underscore ('_'):

```
CLK_DATE_TIME.TZ_Offset
```

4.09 Acronyms, Abbreviations and Mnemonics

The Acronym, Abbreviation and Mnemonics (AAM) Dictionary (AN-2001) should be consulted when forming identifiers. The standard AAM should ALWAYS be used, even if the full word could be accommodated. For example, `Init` should be used rather than `Initialize`. Table 3-2 summarizes some common AAMs; AN-2002 contains a more extensive table.

The AAM included in the right-hand column is for all-lower-case usage. When the AAM is included in a camel-case variable or function name, it is typically modified only by the capitalization of its first letter. For the entries requiring different treatment, a camel-case identifier is included in parenthesis.

Word/Phrase	AAM
argument	`arg`
buffer	`buf`
clear	`clr`
clock	`clk`
compare	`cmp`
configuration	`cfg`
context	`ctx`
delay	`dly`
device	`dev`
display	`disp`
error	`err`
function	`fnct`
hexadecimal	`hex`
initialize	`init`
mailbox	`mbox`
manager	`mgr`
maximum	`max`
message	`msg`
minimum	`min`
Operating System	`os (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`

Table 3-2. Common AAMs.

4.10 'Module-Object-Operation' Format

Identifiers should be prefixed by a module identifier. This prefix makes it easy to locate identifier declarations in medium to large projects. For example, the functions in a file named *kbd.c* and functions in a file named *video.c* could be declared as follows:

kbd.c:

```
Kbd_GetChar()
Kbd_GetLine()
Kbd_GetFnctKey()
```

video.c:

```
Video_GetAttrib()
Video_PutChar()
Video_PutStr()
Video_SetAttrib()
```

It is not necessary to use the whole file or module name as the prefix. For example, the functions in a file named *keyboard.c* could have functions starting with `Kbd` instead of `Keyboard`.

As much as possible, use 'module-object-operation' format with AAMs. When creating identifiers, specify the name of the module (or sub-system) first, followed by the object and then the operation as shown below.

```
OS_SemPost()
OS_SemPend()
```

Here, the module name is `OS` (Operating System), the object is `Sem` (Semaphore) and the operation that can be performed on the object is `Post` or `Pend`. Though 'module-action-object' composition (e.g., `OSPostSem()`) may seem more 'natural', we prefer 'object-action' ordering because functions performing actions on the same object are grouped together. For example:

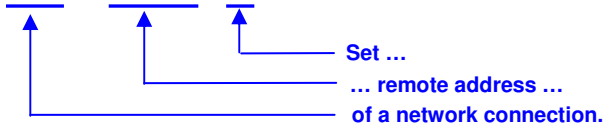
```
OS_SemAccept()
OS_SemCreate()
OS_SemDel()
OS_SemPost()
OS_SemPend()
OS_SemQuery()
```

or

```
NetConn_CloseFromApp()
NetConn_CloseFromTransport()
NetConn_AddrLocalGet()
NetConn_AddrLocalSet()
NetConn_AddrRemoteGet()
NetConn_AddrRemoteSet()
```

The second list of identifiers is excerpted from a network protocol stack's network connection management module. The module identifier includes the stack's identifier (`Net` or network) followed by the module's primary object (`Conn` or connection). The last four functions illustrate how additional sub-objects with a final action form a complete identifier:

```
NetConn_AddrRemoteSet()
```



4.11 Units

An identifier may be suffixed with a units flag to make obvious the magnitude of a variable or `#define` or a function return value. For example:

```
NetOS_TCP_RxQ_TimeoutGet_ms()
NetOS_TCP_TxQ_TimeoutGet_ms()
NET_TMR_TASK_PERIOD_SEC
NET_TMR_TASK_PERIOD_mS
NET_TMR_TASK_PERIOD_uS
NET_TMR_TASK_PERIOD_nS
```

Chapter 5

Data Types

5.01 Portable Data Types

Standard C data types MUST be avoided because their size is not portable (MISRA 13). Instead, the data types in Listing 5-1 should be declared (within a µC/CPU port) based on the target processor and compiler used.

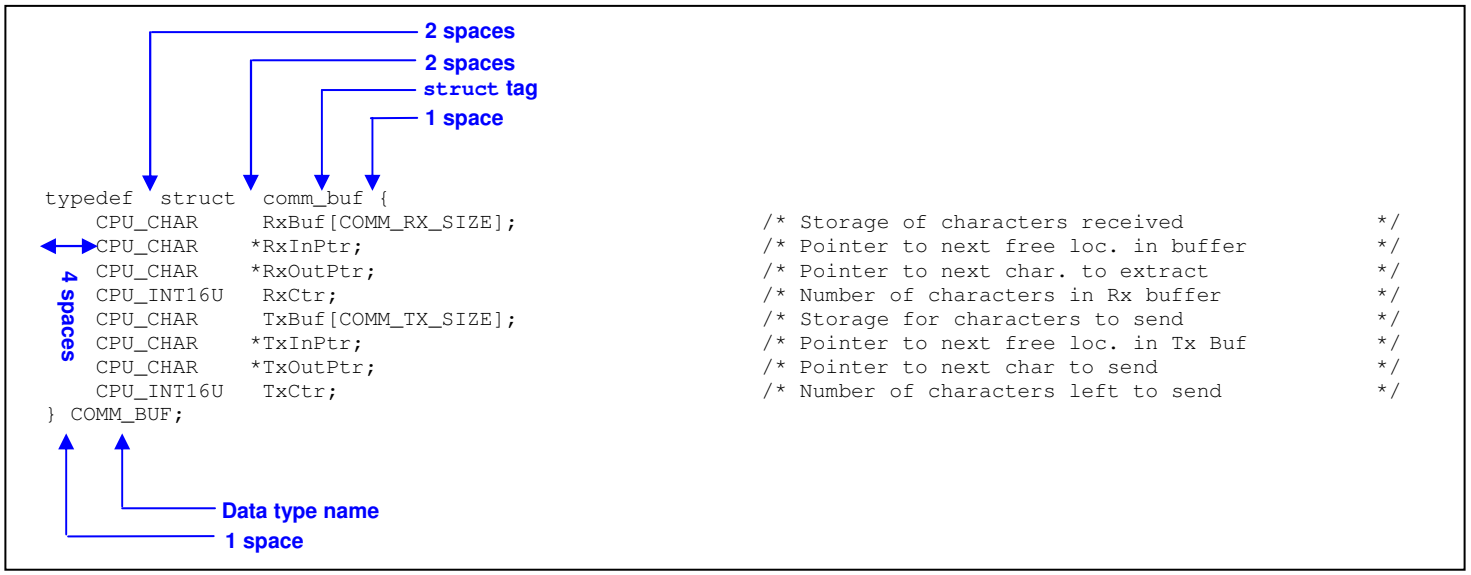
```
typedef  unsigned  char       CPU_CHAR;               /*  8-bit character                         */
typedef  unsigned  char       CPU_BOOLEAN;            /*  8-bit boolean or logical                */
typedef  unsigned  char       CPU_INT08U;             /*  8-bit unsigned integer                  */
typedef    signed  char       CPU_INT08S;             /*  8-bit   signed integer                  */
typedef  unsigned  short      CPU_INT16U;             /* 16-bit unsigned integer                  */
typedef    signed  short      CPU_INT16S;             /* 16-bit   signed integer                  */
typedef  unsigned  int        CPU_INT32U;             /* 32-bit unsigned integer                  */
typedef    signed  int        CPU_INT32S;             /* 32-bit   signed integer                  */
typedef  unsigned  long long  CPU_INT64U;             /* 64-bit unsigned integer                  */
typedef    signed  long long  CPU_INT64S;             /* 64-bit   signed integer                  */

typedef            float      CPU_FP32;               /* 32-bit floating point                    */
typedef            double     CPU_FP64;               /* 64-bit floating point                    */
```

Listing 5-1. Portable (µC/CPU) Data Types.

5.02 `struct` and `union` Types

All structures and unions MUST be typed as shown in Listing 4-2. The data type name MUST be written using all uppercase characters. A `struct` tag MUST be provided, identical to the data type name, except using all lowercase characters. The data types of each member are indented 4 spaces, and the structure member names are also aligned vertically.




```
typedef  struct  comm_buf {
    CPU_CHAR     RxBuf[COMM_RX_SIZE];          /* Storage of characters received              */
    CPU_CHAR    *RxInPtr;                      /* Pointer to next free loc. in buffer         */
    CPU_CHAR    *RxOutPtr;                     /* Pointer to next char. to extract            */
    CPU_INT16U   RxCtr;                        /* Number of characters in Rx buffer           */
    CPU_CHAR     TxBuf[COMM_TX_SIZE];          /* Storage for characters to send              */
    CPU_CHAR    *TxInPtr;                      /* Pointer to next free loc. in Tx Buf         */
    CPU_CHAR    *TxOutPtr;                     /* Pointer to next char to send                */
    CPU_INT16U   TxCtr;                        /* Number of characters left to send           */
} COMM_BUF;
```

Listing 5-2. Proper Structure Data Type

5.03 Forward Declarations

If a structure includes a pointer to a structure of the same type, or several structures have interdependent links, forward declarations can be used.

```
typedef  struct  list_node  LIST_NODE;

struct  list_node {
    LIST_NODE  *NextPtr;
    LIST_NODE  *PrevPtr;
    void       *Data;
};
```

Listing 5-3. Forward Declaration

5.04 Scope

If a data type is only to be used in the implementation file, then it must be declared in the `LOCAL DATA TYPES` section of the implementation file. If the data type is global, if must be placed in the `DATA TYPES` section of the module's header file.

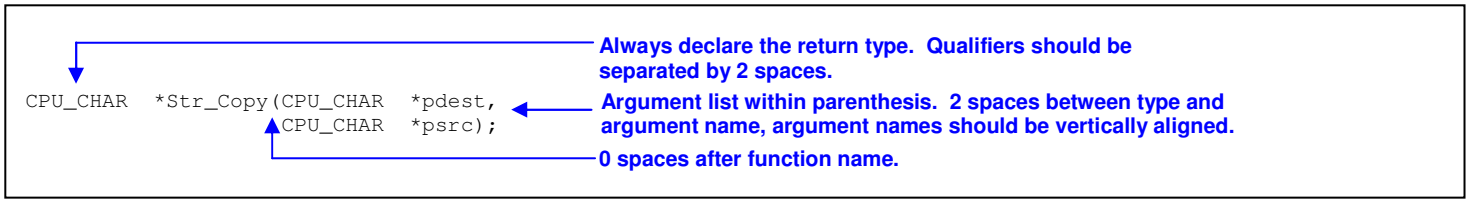
Chapter 6

Functions

6.01 Function Declarations

Prototype functions declarations must be provided for all functions (MISRA 71). Identifiers must be given for all parameters (MISRA 73). The parameter types and identifiers must be identical in both the prototype and definition (MISRA 72, 74). The function must have an explicit return type (MISRA 75). Functions with no parameters must have the parameter type void (MISRA 76).

An example prototype is shown in Figure 6-1.



Listing 6-1. Example Prototype Function Declaration.

Functions should be declared at file-scope (with the qualifier `static`, in the code file) when not referred to from other code files.

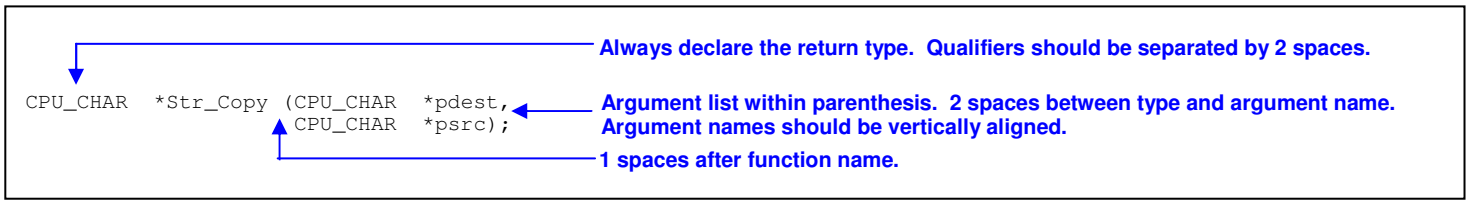
6.02 Function Definitions

Whenever possible, functions should be kept short; one standard metric, 'less than one page', gives a rough quantification of 'short'.

All local variables should be declared at the beginning of the function, followed by two blank lines.

The type of the return expression shall match declared return type (MISRA 83); an explicit cast may be necessary. The return expression should be enclosed in parentheses.

Function definitions should be separated by at least 2 spaces.



Listing 5-2. Example Prototype Function Declaration.

6.03 Function Headers

A function header, as shown in Figure 5-3, includes five mandatory sections:

- **`Description`**. Describe what the function does. This may include an ordered, hierarchical list:
  The first level should use `(1)`, `(2)`, `(3)` ...
  The second level should use `(a)`, `(b)`, `(c)` …
  The third level should use `(1)`, `(2)`, `(3)` …
  The fourth level should use `(A)`, `(B)`, `(C)` …
  … and so on.

- **`Argument(s)`**. List the function arguments by name and the purpose of the function. The argument name should match the name in both the prototype and the declaration. If the argument should hold one value from a certain set of values, then those values should be listed and described. If there are no arguments, specify "`none`.".

- **`Return(s)`**. Give the return value of the function. If nothing is returned, specify "`none`.".

- **`Caller(s)`**. List the caller(s) of the function. If the function is used internally in the module from various locations, specify "`Various`.". If the function is called by the application, specify "`Application`.".

- **`Note(s)`**. List any function notes. This should be an ordered, hierarchical list:
  The first level should use `(1)`, `(2)`, `(3)` ...
  The second level should use `(a)`, `(b)`, `(c)` …
  The third level should use `(1)`, `(2)`, `(3)` …
  The fourth level should use `(A)`, `(B)`, `(C)` …
  … and so on.

  These notes can be referred to from comments (including other sections of the same function header) as described in Section 3.05.

```
/*
*********************************************************************************************************
*                                            Str_Copy()
*
* Description : Copy source string to destination string buffer.
*
* Argument(s) : pdest       Pointer to destination string buffer to receive source string copy.
*
*               psrc        Pointer to source      string to copy into destination string buffer.
*
* Return(s)   : Pointer to destination string, if NO errors.
*
*               Pointer to NULL,               otherwise.
*
* Caller(s)   : Application.
*
* Note(s)     : (1) Destination buffer size NOT validated; buffer overruns MUST be prevented by caller.
*
*                   (a) Destination buffer size MUST be large enough to accommodate the entire source
*                       string size including the terminating NULL character.
*
*               (2) String copy terminates when :
*
*                   (a) Destination/Source string pointer(s) are passed NULL pointers.
*                       (1) No string copy performed; NULL pointer returned.
*
*                   (b) Destination/Source string pointer(s) points to NULL.
*                       (1) String buffer(s) overlap with NULL address.
*                       (2) Source string copied into destination string buffer up to but NOT beyond
*                           or including the NULL address; destination string buffer properly
*                           terminated with NULL character.
*
*                   (c) Source string's terminating NULL character found.
*                       (1) Entire source string copied into destination string buffer.
*********************************************************************************************************
*/
CPU_CHAR  *Str_Copy (CPU_CHAR  *pdest,
                     CPU_CHAR  *psrc)
{
    CPU_CHAR  *pstr;
    CPU_CHAR  *pstr_next;


    if (pdest == (CPU_CHAR *)0) {
        return  ((CPU_CHAR *)0);
    }
    if (psrc  == (CPU_CHAR *)0) {
        return  ((CPU_CHAR *)0);
    }

    pstr      = pdest;
    pstr_next = pstr;
    pstr_next++;
    while (( pstr_next != (CPU_CHAR *)0) &&
           ( psrc      != (CPU_CHAR *)0) &&
           (*psrc      != (CPU_CHAR  )0)) {
       *pstr = *psrc;
        pstr++;
        pstr_next++;
        psrc++;
    }

   *pstr = (CPU_CHAR)0;

    return (pdest);
}
```

Comment block to describe function; this format should always be used.

Always declare the return type. Qualifiers should be separated by 2 spaces.

Argument list within parenthesis. 2 spaces between type and argument name, argument names should be vertically aligned.

1 space after function name.

All local variables declared at beginning of function, followed by 2 blank lines.

```
                                /* Rtn NULL if str ptr(s) NULL.                 */
```

Keep local variable declaration separate from initial value. In other words, do not declare and initialize a variable at the same time.

```
                                /* Copy str until NULL ptr(s) ...               */

                                /* ... or NULL char found.                      */

                                /* Append NULL char.                            */
```

Col 65 — Whenever possible, use the same starting and ending columns for all comments, even between functions and modules. A starting column of 65 and ending column of 121 are recommended. — Col 121

Return value within parentheses.

Listing 5-3. Example Function Definition.

6.04 Function Calls

In a function invocation, there should be no spaces between a function name and its open parenthesis. There should be no space between the open and close parentheses when a function without arguments is invoked:

```
DispInit();
```

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

```
DispStr(x, y, s);
```

The arguments for functions with many parameters can be separated by line breaks, and aligned:

```
Clk_DateTime_Make(&date_time,
                  2008,
                  10,
                  7,
                  6,
                  59,
                  59,
                  0);
```

The type of the parameters passed to a function should be compatible with the expected types in the function prototype (MISRA 77). If an explicit cast is necessary, or an implicit cast would be performed, the parameters should be cast and aligned:

```
Mem_PoolCreate((MEM_POOL   *)&App_Pool,
               (void       *)&App_MemBlk,
               (CPU_SIZE_T  ) 1000,
               (CPU_SIZE_T  ) 10,
               (CPU_SIZE_T  ) 4,
               (CPU_SIZE_T *)&octets_reqd,
               (LIB_ERR    *)&lib_err);
```

Return parameter should be checked (MISRA 86); if ignored, indicate with `void` cast.

Appendix A

Further Information

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

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