

# C Coding Standard

## Rule 1: Any variables must be initialized before using it.



Any variables, including global, static or local variable, must be initialized with an explicit value before using it.

### Reason:

This rule required to avoid using variable with unknown or random value.

**Category:** Compulsory

### Example:

<pre>int xVal = 0; int yVal = 0;  long doMagicAdd(int x, int y) {     long iSum = 0;      if( x &gt; y )     {         iSum = x + y - 1010;     }     else if( x == y )     {         iSum = x + y + 1010;     }     else     {         iSum = x + Y;     }      return iSum; }</pre>	<pre>[ int xVal;   int yVal = 0;    long doMagicAdd(int x, int y)   {     [ long iSum;      if( x &gt; y )     {         iSum = x + y - 1010;     }     else if( x == y )     {         iSum = x + y + 1010;     }     else     {         iSum = x + Y;     }      return iSum; }]</pre>
	

## Rule 2: Compiler/Linker dependent functions should be avoided as possible.

The Compiler/Linker dependent functions will restrict the portability of the code. We should try our best to avoid using such functions, including:

- To access a 3rd party library, we should make sure to follow the same compiler options with which to build this lib. The caller must following the callee invocation convention. Mixed using of `__cdecl`, `__stdcall`, `__fastcall` and `__pascal` is not allowed.
- `#pragma` directive must be documented and explained where it is used, if such usage cannot be avoided.
- Using function calling as parameter should be avoided also, it is implementation-defined. For example: `foo( getValue( ), i++)`

**Reason:**

Some behaviors of the C language are compiler-dependent, therefore these must be understood for how the compiler being used. Examples of issues which need to be understood are: `Stack Usage`, `Parameter Passing` and `how data stored` like `Length`, `Alignment`, `Aliasing`, `Overlay` etc

These rules can improve the code portability.

**Category:** Optional

**Example:** NA

**Rule 3: Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.**

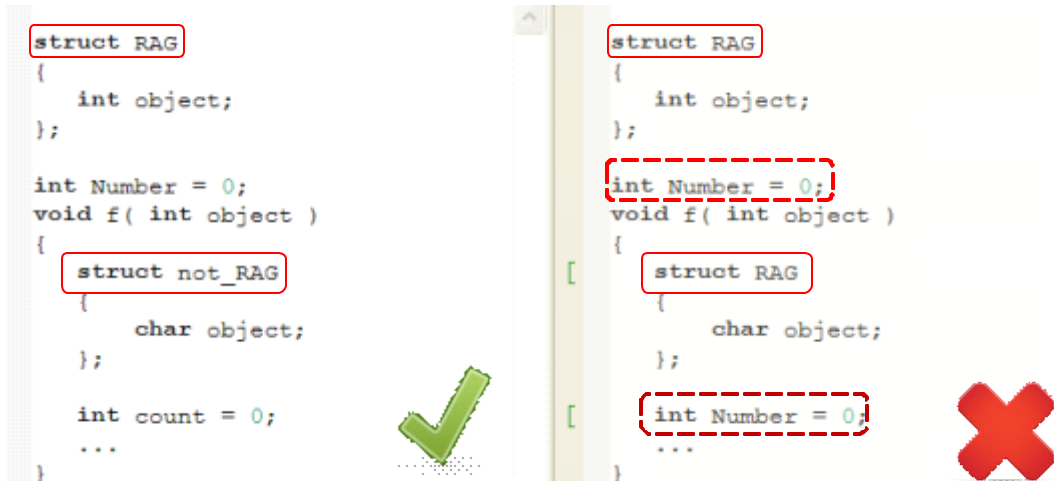
The rule is to disallow the case where an inner definition can hide an outer definition. If the inner definition does not hide the outer definition, then this rule is not violated.

**Reason:**

Hiding identifiers definition in an outer scope leads unexpected mistakes, and such code is some confusing.

**Category:** Compulsory

**Example:**



#### **Rule 4: typedef name must be a unique identifier.**

When defining a type via typedef, the new type identifier (calling "**typedef name**") should be unique at the project scope. No typedef name should be reused either as typedef name or for any other purpose. Where the type definition is made in a header file, and that header file is included in multiple source files, this rule is not a violated.

#### **Reason:**


This rule is to misusing of wrong type definition. Multiple typedef with same name may lead unexpected mistake.

**Category:** Compulsory

**Example:**


```
typedef int int32;

void f(void)
{
    typedef int int32;
    {
        typedef int int32;
        ...
    }
    ....
}
```



```
typedef int int32;

void f(void)
{
    typedef int int32;
    {
        typedef int int32;
        ...
    }
    ....
}
```



**Rule 5: signed and unsigned char type shall be used only for the storage and use of numeric values; the plain char shall be used for character data.**

There are three distinct char types, (plain) char, signed char and unsigned char. signed char and unsigned char shall be used for numeric data and plain char shall be used for character data. The only permissible operators on plain char types are assignment and equality operators (=, ==, !=).

**Reason:**

The sign of the plain char type is implementation-defined and should not be relied upon.

**Category:** Compulsory

**Example:**

<pre>for( signed char i=255; i&gt;0; i-- ) {     ... }  char input = 'a'</pre> 	<pre>for( char i=255; i&gt;0; i-- ) {     ... }  char input = 'a' a = a + 1;</pre> 
--	--

**Rule 6: Bit fields shall only be defined to be of unsigned type or signed type.**



The only allowed type to define bit fields are signed or unsigned type. Plain type (such as int, char, short, long) and enum type are not allowed

**Reason:**

The plain type and enum type are implementation-defined and should not be relied upon.

**Category:** Compulsory

**Example:**

<pre>struct {     signed int f1:2;     signed char f2;     unsigned short f3:4;     unsigned long f4:9; } bit_struct;  union {     signed int f1:2;     signed char f2;     unsigned short f3:4;     unsigned long f4:9; } bit_union;</pre> 	<pre>struct {     int f1:2;     char f2;     short f3:4;     long f4:9; } bit_struct;  union {     int f1:2;     char f2;     short f3:4;     long f4:9; } bit_union;</pre> 
---	---

## Rule 7: Whenever an object or function is declared or defined, its type shall be explicitly stated.

The types of the parameters and return values in the prototype and the definition must match. This requires identical types including typedef names and qualifiers, and not just identical base types.

### Reason:

The use of prototypes enables the compiler to check the integrity of function definitions and calls. Without prototypes the compiler is not obliged to pick up certain errors in function calls.

If such types does not explicitly state, the behavior is compiler-defined. So such case shall be avoided.

**Category:** Compulsory

### Example:



## Rule 8: There shall be no definitions of objects or functions in a header file.

Header files should be used to declare objects, functions, typedefs, macros and constants. Header files shall not contain or produce definitions of objects or functions.

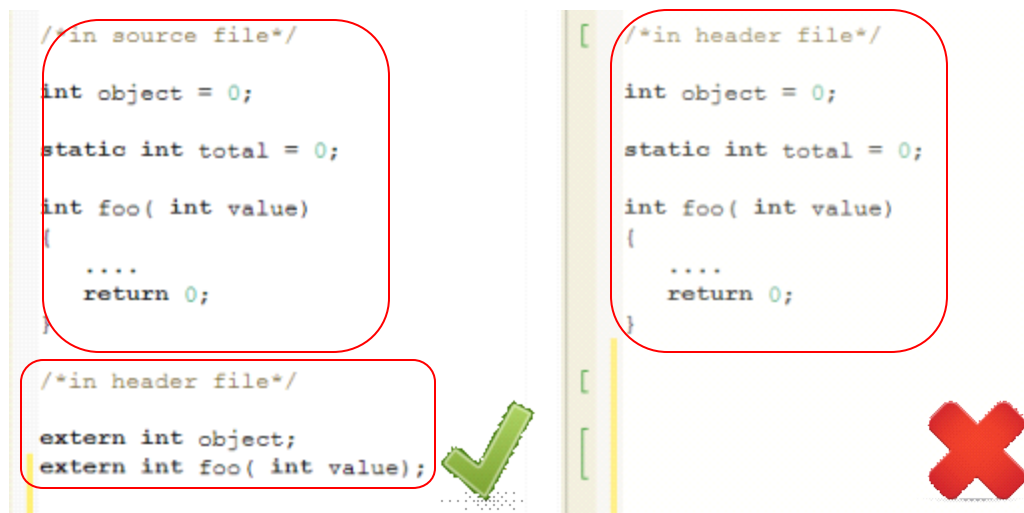
### Reason:

This makes it clear that only C files contain executable source code and that header files only contain declarations. And it can avoid re-definition error possible. It is possible that the compiler/linker will not report error when defining a static const variable in a header file even the header file is included by multiple

source files, but this will increase the size of the firmware; in most cases there will be errors to define a variable or function in a header file.

**Category:** Optional

**Example:**



## **Rule 9: An external object or function shall be declared in one and only one file.**

Normally this will mean declaring an external identifier in a header file, which will be included in any file where the identifier is defined or used. We should avoid using "extern ..." to refer to an external function or variable since only the header file declaration is the standard way to export the external function or variable

### **Reason:**

This rule is to avoid referring an external object or function everywhere, to avoid redefinition error and reduce maintenance cost.

**Category:** Compulsory

**Example:** NA

## Rule 10: All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.

If a variable/function is only to be used by functions within the same source file, to use static to make it only visible in the source file.

If one static variable is only used by one function, moves the static variable into the function;



For local variables, we should define it at the most inner block scope

### Reason:

Use of the static storage-class specifier will ensure that the identifier is only visible in the file in which it is declared and avoids any possibility of confusion with an identical identifier in another file or a library.

**Category:** Compulsory

### Example:

<pre>static int ei = 0; static int check_ei( void ) {     if ( ei &gt; 255 )     {         return 0;     }      return 1; }  void Inc_ei( void ) {     if( check_ei() == 1)     {         ei++;     } }  int get_ei( void ) {     return ei; }</pre>	
<pre>[ int ei = 0; [ int check_ei( void ) {     if ( ei &gt; 255 )     {         return 0;     }      return 1; }  void Inc_ei( void ) {     if( check_ei() == 1)     {         ei++;     } }  int get_ei( void ) {     return ei; }</pre>	



## Rule 11: Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.

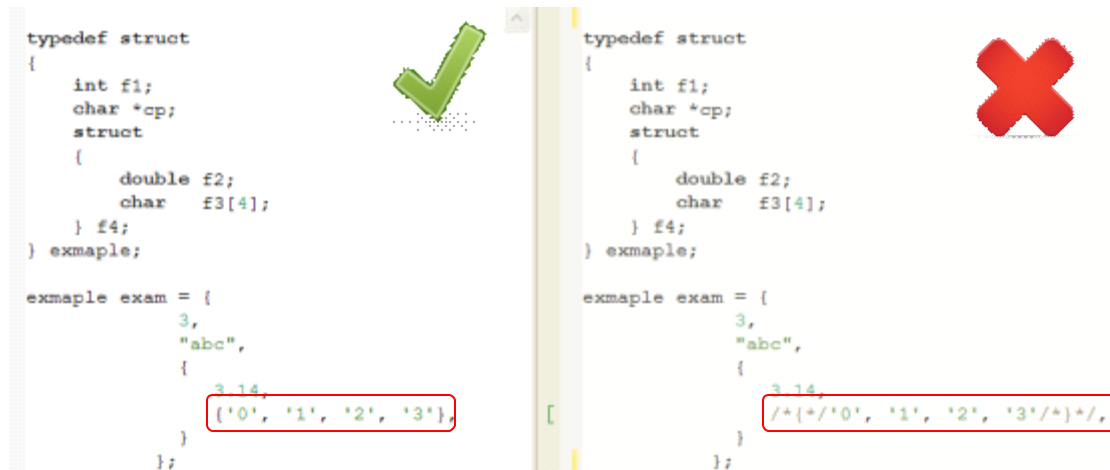
This forces the programmer to explicitly consider and demonstrate the order in which elements of complex data types are initialized.

### Reason:

It can avoid typo.

**Category:** Compulsory

### Example:



## ~~Rule 12: for security special rules, please refer to C Language Common Secure Coding Specification.~~

~~Security is very important; the new added code should comply with C Language Common Secure Coding Specification~~

### ~~Reason:~~

~~The C language Common Secure Coding Specification defines how to write secure code with C.~~

~~**Category:** Optional~~

**Example:** NA

### **Rule 13: The address of auto variables allocated from the stack shall NEVER be returned as a pointer.**

The address of an auto variable allocated from the stack shall NEVER be returned as a pointer since it will be freed by the compiler when exiting its functional area.



#### **Reason:**

This rule is to avoid accessing a free memory since the memory is freed after the function is returned:

- To read the content via the returned address/pointer, the value is uncertain
- To write to the returned address/pointer, the stack of another function is most likely to be corrupted

**Category:** Compulsory

#### **Example:**

<pre>int * fool() {     static int value = 0;     ...     return &amp;value; }  char * foo2() {     char * str = "foo2";     ...     return str; }</pre>		<pre>int * fool() {     int value = 0;     ...     return &amp;value; }  char * foo2() {     char str[] = "foo2";     ...     return str; }</pre>
		

**NOTES:** The bad examples are actually code errors!

**Rule 14: goto statement shall be avoided as possible as we can.**

**Reason:**

The goto statement will change the execution progress of the program - such logic will reduce the readability of the code, also easy to introduce bugs such as resource leakage.

In rare cases, judicious use of goto statement may provide more compact code, and may even improve readability. If a goto must be used, then the following rules shall be observed:

- The goto shall not be used to terminate a controlled loop
- The goto shall not leave the current block
- A label shall be referenced by one and only one goto

**Category:** Optional

**Example:** N/A

**NOTES:** Similarly **setjmp** and **longjmp** should be avoided also

**Rule 15: Upon relinquishing any resources, the resource related variable shall be set as a meaningful invalid value.**

When one buffer is relinquished, the pointer of the buffer shall be set as NULL; when one handle is closed, the handle shall be set as 0.

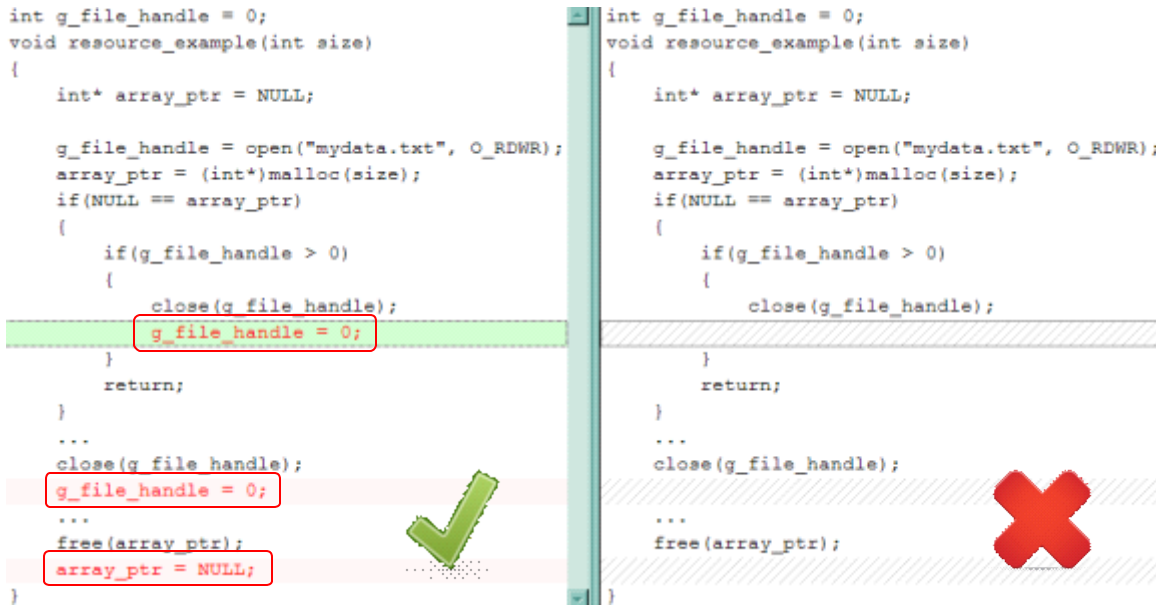
**Reason:**

The meaningful value of one resource related variable is a clear indication to the programmer that the resource is invalid anymore. It is always a good behavior to make sure:

- When accessing one resource, always validate its value
- After relinquishing one resource, always assign an invalid value

**Category:** Compulsory

**Example:**



**NOTES:** There is code error in the bad examples!

## Rule 16: Each switch statement should contain a default case.

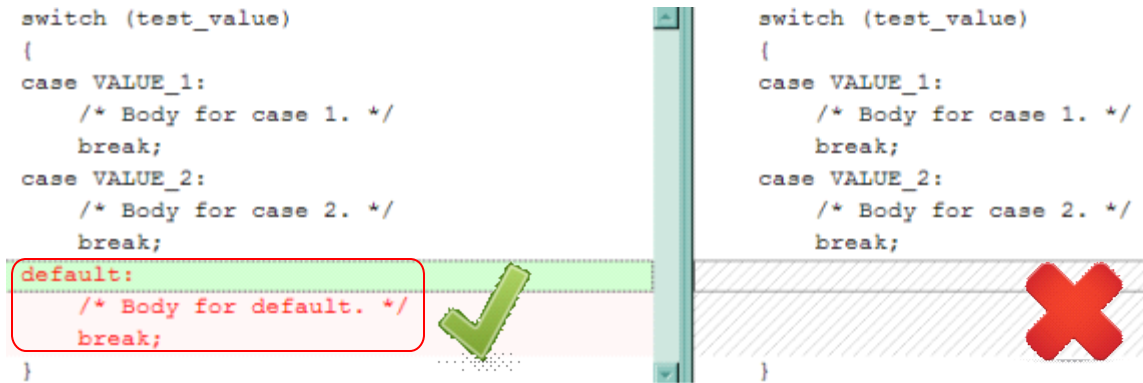
This indicates to the maintainer that the original author did consider cases which may not be covered by the specified case statements.

### Reason:

The default statement reminds the programmer to consider the exception cases and deal with these cases gracefully. Similarly we shall consider the **else** statement of any **if** statement.

**Category:** Compulsory

**Example:**



## Rule 17: The max level of usage for indirect address should not more than 2.

This rule recommends NOT use a pointer for indirect address reference more than 2 levels.

### Reason:

To improve the readability of the code, the dimension of indirect access via pointer shall be limited. There shall be other alternatives to avoid the 3 or more dimensions of pointer usage.

**Category:** Optional

### Example:



## Rule 18: Declaring the magnitude of a single dimensional array in an argument declaration shall NEVER be used.

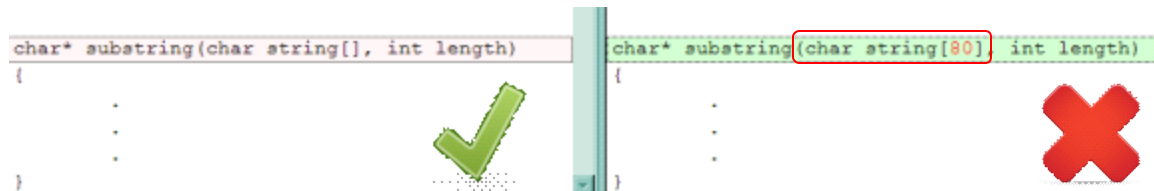
The 'C' language will pass an array argument as a pointer to the first element in the array. In fact, different invocations of the function may pass array arguments with different magnitudes. Therefore, specifying the magnitude of an array in a function argument definition might only serve to hinder software maintenance.

### Reason:

The magnitude of an array may mislead the junior programmer to calculate the size of the array parameter via **sizeof**, but the result is NOT the size of the array, but the size of one pointer.

**Category:** Optional

### Example:



## Rule 19: Defining a structural array instead of multiple arrays for data accessed by the same index.

If there are multiple attributes for one entity with multiple entries, it is recommended to define one structural array to save the entity instead of multiple arrays for each attribute.

### Reason:



Multiple arrays accessed by the same index shall introduce the maintenance issue such as: when adding/deleting one entry, the programmer is required to:

- Add/delete the entry at the correct position of all these arrays
- Change all these arrays without any missing

Even in the creation phase, multiple array definition is an unnecessary burden to the programmer.

**Category:** Optional

### Example:

<pre> struct menu {     int string_id;     int target_app_id; };  struct menu menu_info[] = {     {MSI_CONTACTS, APP_UCL_DIGITAL},     {MSI_SCANNING, APP_SCAN_MENU},     ... }; </pre>		<pre> int menu_string_ids[] = {     MSI_CONTACTS,     MSI_SCANNING,     ... };  int menu_target_app_ids[] = {     APP_UCL_DIGITAL,     APP_SCAN_MENU,     ... }; </pre>	
---	---	---	---

**Rule 20: If a function can fail, the function shall return a status indicating a failure. More specific, if the function returns a pointer, we should return NULL for the error conditions.**

When writing a function, carefully consider all reasons why the function could fail. For every reason that a function could fail, you should write code to check for the fail conditions and exit the function with a return code indicating that the function failed. This method can be used for checking the validity of input data as well as the validity of any intermediate result including the end result.

#### **Reason:**

Both the caller and callee shall take careful consideration to the error conditions:

- The callee function should make sure to report all the possible error conditions
- The caller function should make sure to check the result of the errors then decide the correct behavior
- Especially for the **interfaces** among multiple components, the error checking is a compulsory requirement

**Category:** Compulsory

**Example:** N/A

**Rule 21: Structures shall be passed by reference.**

Passing the address of a structure is usually faster because only a single pointer is copied to the argument area. Passing the structure by value requires the entire structure to be copied.



Similarly any function shall avoid returning a structural value; instead we shall define a temporary variable in the caller and pass its address for the callee to update its value.

**Reason:**

The execution speed is the reason of this rule.

**Category:** Optional

**Example:**

<pre>void structure_example1(const struct menu* menu_item_ptr) {     ... }  void structure_example2(struct menu* menu_item_ptr) {     ... }</pre>	<pre>void structure_example1(struct menu menu_item) {     ... }  struct menu structure_example2() {     struct menu menu_item = {0};     ...     return menu_item; }</pre>
	

**Rule 22: If a 'case' with code within a 'switch' statement does not contain a normal break, a comment shall be used to highlight this and indicate it is intentional.**

This is a must for maintainability. It tells the maintenance personnel that this was intended and not an oversight. It also makes the exception to the expected usage stand out better. When consistently applied, it will help to highlight logic errors in code inspections.

**Reason:** N/A.

**Category:** Compulsory

**Example:**



```

switch (c)
{
case 'A':
    /* body for A */
    break;
case 'B':
    /* body for B */
    /* flow through case C */
case 'C':
    /* body for C */
    break;
default:
    ...
    break;
}
switch (digit)
{
    /* same action taken for cases 3, 5, 7 */
case 3:
case 5:
case 7:
    ...
    break;
default:
    ...
    break;
}

```



```

switch (c)
{
case 'A':
    /* body for A */
    break;
case 'B':
    /* body for B */
case 'C':
    /* body for C */
    break;
default:
    ...
    break;
}
switch (digit)
{
case 3:
case 5:
case 7:
    ...
    break;
default:
    ...
    break;
}

```



## Rule 23: Header files shall use a defined constant to allow for multiple inclusions.

The key drivers here are reusability, maintainability, and, more importantly, portability. This practice reduces the number of things a programmer must keep in mind when using routines supplied by others or maintaining code. For example, if Miles writes a set of string manipulation routines and Nancy writes a set of linked list routines, then Pete can use both sets of routines without having to worry that the header for Nancy's routines includes the header for Miles' set of routines.

### Reason:

The rule will allow for multiple inclusions for the header file

**Category:** Compulsory

### Example:

/* File Name: example1.h */	/* File Name: example1.h */
#ifndef EXAMPLE1_H #define EXAMPLE1_H	
.	
.	
#endif	
/* File Name: example2.h */	/* File Name: example2.h */
#ifndef EXAMPLE2_H #define EXAMPLE2_H	#ifndef Example2_H #define Example2_H
.	.
.	.
#endif	#endif
/* File Name: example3.h */	/* File Name: example3.h */
#ifndef EXAMPLE3_H #define EXAMPLE3_H	#ifndef MY_EXAMPLE_H #define MY_EXAMPLE_H
.	.
.	.
#endif	#endif

## Rule 24: In function-like macros, parameters shall be fully parenthesized.

Every time a parameter is used it must be enclosed in parenthesis. This promotes readability as well as preventing possible errors.

### Reason:

The rule will allow errors happening during compiler macro expansion.

**Category:** Compulsory

### Example:

#define PRT_DEBUG(a, b) printf("ERROR: %s:%d, %s\n", \	#define PRT_DEBUG(a, b) printf("ERROR: %s:%d, %s\n", \
(a), __LINE__, (b));	a, __LINE__, b);
#define MULTIPLY(a, b) (a) * (b)	#define MULTIPLY(a, b) a * b

# C Coding Style

This section describes the rules for both format and content, which can help to make the codes more better for reading and understanding, and also can help to avoid some unintentional and simple typo.

## 1 Comments

### Rule 25: Header of a file

At the beginning of each file, it should include these sections as comments

- Copyright
- File Name
- Function Names List
- Originator and Origin Date
- The content description(Purpose, Synopsis)
- Global data description
- Change History

**Reason:** Good for understanding and tracing.

**Category:** Compulsory

```

/*****
*
*      C MODULE CLUSTER FUNCTIONS
*      COPYRIGHT 1998-2012 MOTOROLA SOLUTIONS
*
*      MOTOROLA SOLUTIONS CONFIDENTIAL PROPRIETARY
*
*****/
*
* FILE NAME       : group_call_handler.c
* FUNCTION NAMES  : group_call_handler
*                  : group_call_handler_ext
* ORIGINATOR      : xxx.aaa
* DATE OF ORIGIN  : 07/97
*
*----- PURPOSE -----
*
* This file contains all the packet handler function for the group call classification.
*
*----- SYNOPSIS -----
*
*----- GLOBAL DATA DESCRIPTION -----
*
*----- REVISIONS -----
* Date      Name      Prob#      Description
*-----
* 01/07/97   xxx       CCMPDxxxx Initial Version
* 01/21/99   yyyy      CCMPDxxxx Add .... function.
* 04/04/12   zzzzzz     CCMPDxxxx Fix .... issue.
*****/
/
```



## Rule 26: Header of a function

At the beginning of each function definition, it should include these sections as comments:

- Copyright
- Function Name
- Originator and Origin Date
- Function Description
- Parameter Description
- Global data Referred
- Function Logic Description
- Change History

**Reason:** Good for understanding and tracing.

**Category:** Compulsory

```

/*****
 *
 *      STATIC SUBFUNCTION
 *      COPYRIGHT 1997- 1999 MOTOROLA
 *
 *      MOTOROLA CONFIDENTIAL PROPRIETARY
 *
 *****/

* FUNCTION NAME   : get_max_ch_num
* ORIGINATOR     : xxx.yyyy
* DATE OF ORIGIN  : 11/19/97
*
*----- PURPOSE -----
* This function will handle ....
*
*----- SYNOPSIS -----
*
* ASSUMPTIONS/PRECONDITIONS: assume the select ch is not equal to 0...
* POSTCONDITIONS: msg_header is reformatted as necessary
*
* PARAMETER DESCRIPTION:
*
* PARAMETERS PASSED IN: ch_packet_type - type of SAH packet received
*                      attach_request - type of SAH request received
*
* PARAMETERS RETURNED: None
*
*----- GLOBAL DATA DESCRIPTION -----
*
* sig_control_msg_ptr - this points to the most recently received input buffer
*
*----- REVISIONS -----
*


| Date     | Name    | Problem | Description                      |
|----------|---------|---------|----------------------------------|
| 11/19/97 | xxx.yyy |         | initial creation                 |
| 10/21/99 | aaabbb  | xyz852  | Updated DOL per review comments. |


*----- DESCRIPTION OF LOGIC -----
* IF return value from CALL ch_is_valid with argument
*   ch_packet_type EQUALS FALSE
*   SET attach_action (local) to the return value of ch_system_request with
*   argument attach_request
*   IF attach_action DOES NOT EQUAL NO_CH_ACTION
*     REFORMAT sig_control_msg_ptr header to SIG_CH_ATTACH_ACTION_REQ,
*     SLIO_INTRA_TASK, and NULL with attach_action field equal to attach_action
*   ENDIF
* ENDIF
*****/
/
```

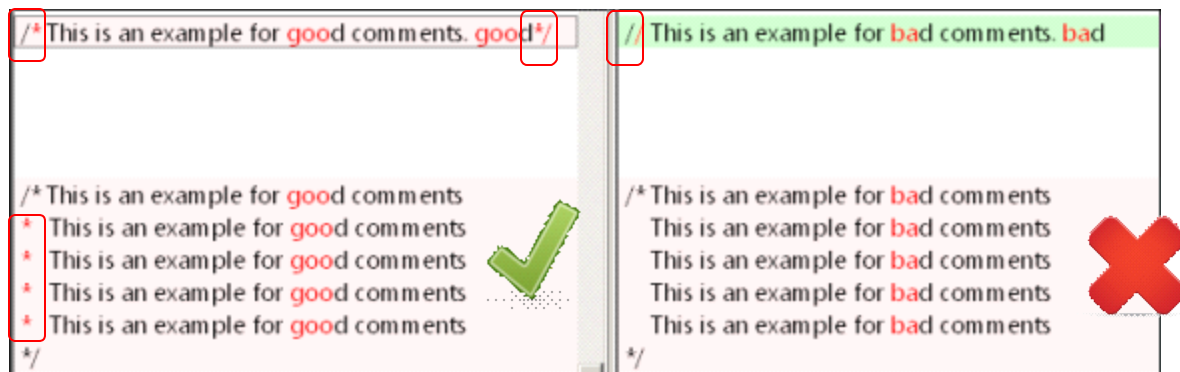
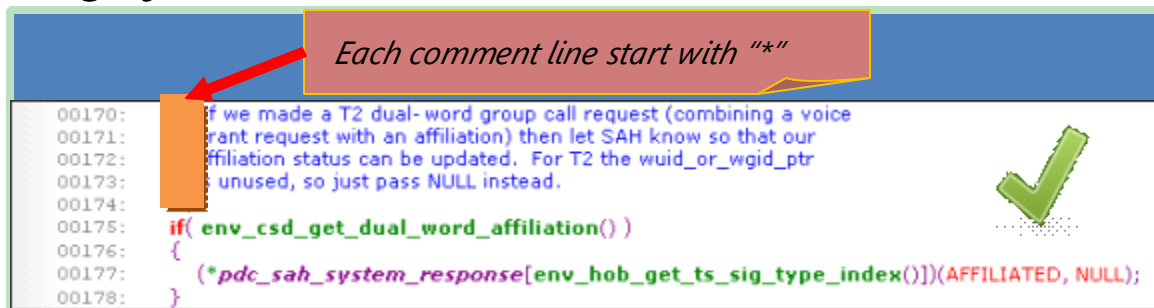


## Rule 27: Source code shall only use `/* ... */` style comments and avoid the nesting of comments; each comment line should start with `'*'`.

`///  
*/` style comment is from C89 but some other C compilers may not support such comment style.



**Reason:** some compilers support the `///  
*/` style of comments as an extension to C89. The use of `//` in preprocessor directives (e.g. `#define`) can vary. Also the mixing of `/* ... */` and `//` is not consistent. This is more than a style issue, since different (pre C89) compilers may behave differently.

**Category:** Compulsory



## Rule 28: Good enough comments

- Comments should be consistent with the code;
- If the code is self-commented, no extra comment need;
- Don't insert comments inner a code line
- Add a comment to indicate the end of a coding block, like "End of ...".  
Especially at the end of a function or a long loop/switch block like "if, for, while, switch-case".



<pre> DEVICE_MODE_TYPE check_device_home(     unsigned int device_zone_id,     unsigned device_sys_id ) {     unsigned int bs_zone_id = 0; /*current operating base station's zone ID*/     unsigned int bs_sys_id = 0; /*current operating base station's system ID*/     DEVICE_MODE_TYPE current_dev_mode = 0; /*current device's operating mode*/     unsigned char rtn = LOCAL_BS;      bs_zone_id = get_bs_zone_id();     bs_sys_id = get_bs_sys_id();     current_dev_mode = get_curr_dev_mode();      /*According to the requirement R100, the zone_id, sys_id and the device operatin     /*mode will be applied to determine the mode of the operating base station*/     if( ZONE_MODE == current_dev_mode )     {         if( (device_zone_id == bs_zone_id) &amp;&amp; (device_sys_id == bs_sys_id) )         {             rtn = ZONE_BS;         }     }     else if( SYS_MODE == current_dev_mode )     {         if( device_sys_id == bs_sys_id )         {             rtn = SYSTEM_BS;         }     }     /*end of if( WACH_ROAMING == current_coverage_type)*/     return rtn; }/*end of check_to_home()*/ </pre> 	<pre> DEVICE_MODE_TYPE check_device_home(     unsigned int device_zone_id,     unsigned device_sys_id ) {     unsigned int bs_zone_id = 0;     unsigned int bs_sys_id = 0;     DEVICE_MODE_TYPE current_dev_mode = 0;     unsigned char rtn = LOCAL_BS;      bs_zone_id = get_bs_zone_id();     bs_sys_id = get_bs_sys_id();     current_dev_mode = get_curr_dev_mode();      if( ZONE_MODE == current_dev_mode )     {         if( (device_zone_id == bs_zone_id) &amp;&amp; (device_sys_id == bs_sys_id) )         {             rtn = ZONE_BS;         }     }     else if( SYS_MODE == current_dev_mode )     {         if( device_sys_id == bs_sys_id )         {             rtn = SYSTEM_BS;         }     }     return rtn; } </pre> 
--	--

## 2 Variable

**Rule 29: Local variable definitions shall be one per line. Exceptions to this rule are loop counters, temporary variables, and initializes to the same value**

**Reason:** To make sure the logic meaning of the variable shown to the programmer and reader, this rule will improve the readability of the code and help the reader to quickly capture the meaning and usage of the variable.

**Category:** Optional

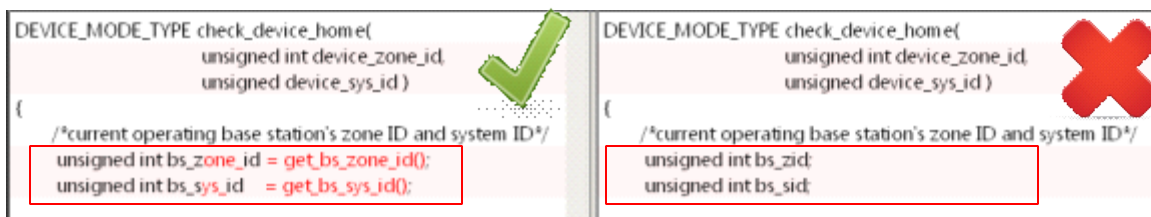
<pre> DEVICE_MODE_TYPE check_device_home(     unsigned int device_zone_id,     unsigned device_sys_id ) {     /*current operating base station's zone ID and system ID*/     unsigned int bs_zone_id = get_bs_zone_id();     unsigned int bs_sys_id = get_bs_sys_id();      /*current device's operating mode*/     DEVICE_MODE_TYPE current_dev_mode = get_curr_dev_mode();      unsigned int i, j, k;      i = 0;     j = 0;     k = 0; } </pre> 	<pre> DEVICE_MODE_TYPE check_device_home(     unsigned int device_zone_id,     unsigned device_sys_id ) {     /*current operating base station's zone ID and system ID*/     unsigned int bs_zone_id, bs_sys_id;      /*current device's operating mode*/     DEVICE_MODE_TYPE current_dev_mode = get_curr_dev_mode();      unsigned int i, j, k;      bs_zone_id = get_bs_zone_id();     bs_sys_id = get_bs_sys_id();      i = 0, j = 0, k = 0; } </pre> 
--	---

### Rule 30: Variable names shall be used which are meaningful to those reviewing the code.

This promotes readability and maintainability since the variables are self documenting. Note, it may not be possible to give all variables a meaningful name. Suggestions for variable names in which a meaningful name may not apply are shown below.

**Reason:** The good name of one identifier will help the reviewer to understand the logic and design of the code.

**Category:** Compulsory



### Rule 31: Global variable names shall use a convention which clearly differentiates them from local variables



**Category:** Optional

### Rule 32: To define a '#defined' macro, or a constant, or an enumerated list, must use all UPPERCASE characters.

All '#defined' constants shall be in UPPERCASE. This does not apply to Macros. This convention will make the recognition of constants easier and support future maintenance efforts. Similarly if an identifier is not a macro, should not use all UPPERCASE characters.

UPPERCASE is a dedicated indication to a macro or enum, we should follow the common practice as possible as we can.



**Category:** Compulsory

<pre>#define MAX_RES_COUNT 100 #define MIN_RES_COUNT 1  const int MAX_FIND_RETRY_NUM 100 const int MIN_FIND_RETRY_NUM 100  typedef enum { LOCAL_BS = 0, SYSTEM_BS, ZONE_BS } DEVICE_MODE_TYPE;</pre> 	<pre>#define MAX_RES_count 100 #define MIN_RES_count 1  const int MAX_find_retry_NUM 100 const int MIN_fine_retry_NUM 100  typedef enum { local_BS = 0, syste_BS, zone_BS } DEVICE_MODE_TYPE;</pre> 
--	---

### Rule 33: NULL shall only be used for pointer comparisons and initialization.

NULL shall never be used to terminate a string. The symbol NULL is defined as a null pointer. 'C' guarantees that no valid pointer will ever have a value of zero. The size of the pointer is highly machine dependent, and is not interchangeable with zero. In the INTEL 80x86 architecture (real-mode) NULL may be '0' or '0L', depending on what memory model the program is using.

**Category:** Recommended

<pre>unsigned int *message_ptr = NULL;  unsigned int *renew_system_msg_ptr ( unsigned int *msg_ptr) {     if( NULL != msg_ptr)     {         renew( msg_ptr-&gt;system_msg_ptr );     }     else     {         msg_ptr-&gt;system_msg_ptr = NULL;     }      return (msg_ptr-&gt;system_msg_ptr); }</pre> 	<pre>unsigned int *message_ptr = 0;  unsigned int *renew_system_msg_ptr ( unsigned int *msg_ptr) {     if( 0 != msg_ptr)     {         renew( msg_ptr-&gt;system_msg_ptr );     }     else     {         msg_ptr-&gt;system_msg_ptr = 0;     }      return (msg_ptr-&gt;system_msg_ptr); }</pre> 
---	--



## 3 Indentation



**Rule 34: Use indentation to show the logical structure of the code for improving the readability of the codes.**



- Spaces should be used to indent code, rather than tabs.
- Indent 4 spaces at a time. (Research has shown that 4 spaces is the optimum indent for readability and maintainability.)
- Switch-case block also need to add indentation.
- Comments should be aligned with its code block.

**Category:** Compulsory

<pre> unsigned int * renew_system_msg_ptr ( unsigned int * msg_ptr) {     if( NULL != msg_ptr)     {         renew( msg_ptr-&gt;system_msg_ptr );     }     else     {         msg_ptr-&gt;system_msg_ptr = NULL;     }     return (msg_ptr-&gt;system_msg_ptr); } </pre> 	<pre> unsigned int * renew_system_msg_ptr ( unsigned int * msg_ptr) {     if( NULL != msg_ptr)     {         renew( msg_ptr-&gt;system_msg_ptr );     }     else     {         msg_ptr-&gt;system_msg_ptr = NULL;     }     return (msg_ptr-&gt;system_msg_ptr); } </pre> 
---	---



<pre> DEVICE_MODE_TYPE check_device_home(     unsigned int device_zone_id,     unsigned device_sys_id ) {     /*current operating base station's zone ID and system ID*/     unsigned int bs_zone_id = get_bs_zone_id();     unsigned int bs_sys_id = get_bs_sys_id();      /*current device's operating mode*/     DEVICE_MODE_TYPE current_dev_mode = get_curr_dev_mode(); } </pre> 	<pre> DEVICE_MODE_TYPE check_device_home(     unsigned int device_zone_id,     unsigned device_sys_id ) {     /*current operating base station's zone ID and system ID*/     unsigned int bs_zone_id = get_bs_zone_id();     unsigned int bs_sys_id = get_bs_sys_id();      /*current device's operating mode*/     DEVICE_MODE_TYPE current_dev_mode = get_curr_dev_mode(); } </pre> 
---	---

## 4 Blank line



Inserting some blank lines inside the codes will be good for understanding and will significantly improve the readability of the code.

**Category:** Optional

### Rule 35: Blank line to make code like a 'paragraph'

<pre> unsigned int renew_dev_zone_id( unsigned int device_zone_id ) {     /*current operating base station's zone ID and system ID*/     unsigned int bs_zone_id = get_bs_zone_id();      if( (0 != bs_zone_id) &amp;&amp; (0 != device_zone_id) )     {         device_zone_id = bs_zone_id;     }     else     {         device_zone_id = FIX_ZONE_ID;     }      return device_zone_id; }/*end of renew_dev_zone_id*/ </pre> 	<pre> unsigned int renew_dev_zone_id( unsigned int device_zone_id ) {     /*current operating base station's zone ID and system ID*/     unsigned int bs_zone_id = get_bs_zone_id();      if( (0 != bs_zone_id) &amp;&amp; (0 != device_zone_id) )     {         device_zone_id = bs_zone_id;     }     else     {         device_zone_id = FIX_ZONE_ID;     }      return device_zone_id; }/*end of renew_dev_zone_id*/ </pre> 
---	---

## Rule 36: Blank line should be inserted between the comments and its above code block

<pre>DEVICE_MODE_TYPE check_device_home(     unsigned int device_zone_id,     unsigned device_sys_id ) {     /*current operating base station's zone ID and system ID*/     unsigned int bs_zone_id = get_bs_zone_id();     unsigned int bs_sys_id = get_bs_sys_id();     /*current device's operating mode*/     DEVICE_MODE_TYPE current_dev_mode = get_curr_dev_mode(); }</pre> 	<pre>DEVICE_MODE_TYPE check_device_home(     unsigned int device_zone_id,     unsigned device_sys_id ) {     /*current operating base station's zone ID and system ID*/     unsigned int bs_zone_id = get_bs_zone_id();     unsigned int bs_sys_id = get_bs_sys_id();     /*current device's operating mode*/     DEVICE_MODE_TYPE current_dev_mode = get_curr_dev_mode(); }</pre> 
--	--



## 5 General Coding format

### Rule 37: Logical parts of a conditional expression shall be grouped with parenthesis even if not logically required

This promotes readability by explicitly indicating the logical groups being compared.

**Reason:** Good for understand and avoid typo. And it can reflect the intention of the programmer.



**Category:** Compulsory

<pre>unsigned int renew_dev_zone_id( unsigned int device_zone_id ) {     /*current operating base station's zone ID and system ID*/     unsigned int bs_zone_id = get_bs_zone_id();     if ( 0 != bs_zone_id &amp;&amp; 0 != device_zone_id )     {         device_zone_id = bs_zone_id;     } }</pre> 	<pre>unsigned int renew_dev_zone_id( unsigned int device_zone_id ) {     /*current operating base station's zone ID and system ID*/     unsigned int bs_zone_id = get_bs_zone_id();     if 0 != bs_zone_id &amp;&amp; 0 != device_zone_id     {         device_zone_id = bs_zone_id;     } }</pre> 
--	--

### Rule 38: All conditional statements shall use { and } to identify the body of code associated with the condition, even if the body has only a single line or no line of code in it.

This promotes readability by explicitly identifying the code block associated with a conditional expression. This also reduces the risk of error if a programmer has to add statements to a conditional code block which beforehand had none or only a single line of code associated with it. When interpreting this rule else if shall be considered as one conditional statement.

**Category:** Compulsory



<pre>unsigned int renew_dev_zone_id(unsigned int device_zone_id) {     /*current operating base station's zone ID and system ID*/     unsigned int bs_zone_id = get_bs_zone_id();      if( ( 0 != bs_zone_id ) &amp;&amp; ( 0 != device_zone_id ) )     {         device_zone_id = bs_zone_id;     }     else     {         device_zone_id = FIX_ZONE_ID;     } }</pre> 	<pre>unsigned int renew_dev_zone_id(unsigned int device_zone_id) {     /*current operating base station's zone ID and system ID*/     unsigned int bs_zone_id = get_bs_zone_id();      if( ( 0 != bs_zone_id ) &amp;&amp; ( 0 != device_zone_id ) )     {         device_zone_id = bs_zone_id;     }     else device_zone_id = FIX_ZONE_ID; }</pre> 
---	---

### Rule 39: Use positive logic rather than negative logic whenever practical

The use of many logical NOT "!" within an expression makes the expression difficult to understand and maintain

**Reason:** Good for understand.

**Category:** Optional

<pre>unsigned int renew_dev_zone_id(unsigned int device_zone_id) {     /*current operating base station's zone ID and system ID*/     unsigned int bs_zone_id = get_bs_zone_id();      if( ( 0 != bs_zone_id ) &amp;&amp; ( 0 != device_zone_id ) )     {         device_zone_id = bs_zone_id;     } }</pre> 	<pre>unsigned int renew_dev_zone_id(unsigned int device_zone_id) {     /*current operating base station's zone ID and system ID*/     unsigned int bs_zone_id = get_bs_zone_id();      if( !( 0 == bs_zone_id ) ) &amp;&amp; ( !( 0 == device_zone_id ) ) )     {         device_zone_id = bs_zone_id;     } }</pre> 
--	--

### Rule 40: Line Length

The source code should not contain lines that are longer than 100 characters long. If a line is longer than 100 characters, please break it up into multiple lines by logically.

**Reason:** Good for understand.



**Category:** Optional

***5.1 Add space between the type operator and pointer operator.***

### Rule 41: A constant should be at the left side in 'if' logic condition statement.

**Reason:** To prevent typo error.



**Category:** Optional

<pre>unsigned int renew_dev_zone_id( unsigned int device_zone_id ) {     /*current operating base station's zone ID and system ID*/     unsigned int bs_zone_id = get_bs_zone_id();      if( ( 0 != bs_zone_id ) &amp;&amp; ( 0 != device_zone_id ) )     {         device_zone_id = bs_zone_id;     }     else</pre> 	<pre>unsigned int renew_dev_zone_id( unsigned int device_zone_id ) {     /*current operating base station's zone ID and system ID*/     unsigned int bs_zone_id = get_bs_zone_id();      if( ( bs_zone_id != 0 ) &amp;&amp; ( device_zone_id != 0 ) )     {         device_zone_id = bs_zone_id;     }     else</pre> 
---	---

## Rule 42: {} should occupy one line and be aligned.

The brace sign should occupy one line exclusively. “{” and “}” should align correspondingly to each other.

**Category:** Compulsory

<pre>unsigned int renew_dev_zone_id( unsigned int device_zone_id ) {     /*current operating base station's zone ID and system ID*/     unsigned int bs_zone_id = get_bs_zone_id();      if( ( 0 != bs_zone_id ) &amp;&amp; ( 0 != device_zone_id ) )     {         device_zone_id = bs_zone_id;     }     else     {         device_zone_id = FIX_ZONE_ID;     } }</pre> 	<pre>unsigned int renew_dev_zone_id( unsigned int device_zone_id ) {     /*current operating base station's zone ID and system ID*/     unsigned int bs_zone_id = get_bs_zone_id();      if( ( 0 != bs_zone_id ) &amp;&amp; ( 0 != device_zone_id ) )     {         device_zone_id = bs_zone_id;     }     else ( device_zone_id = FIX_ZONE_ID; ) }</pre> 
---	---

## ***5.2 Only one statement per line is allowed.***