Coding Standard, Testing and Style Guide

On this page:

Coding Standard / MISRA Compliance

The core FreeRTOS source files (those that are common to all ports, but not the port layer) conform to the MISRA coding standard guidelines. Compliance is checked using pc-lint with the linked lint configuration files. As the standard is many pages long, and is available for purchase from MISRA for a very small fee, we have not replicated all the rules here.

Deviations from the MISRA standard are listed below:

- Two API functions have more than one exit point. A deviation was permitted in these two cases for reasons of critical efficiency.
- When creating tasks, the source code manipulates memory addresses to locate the start and end addresses of the stack allocated to the created task. The code has to work for all the architectures to which FreeRTOS has been ported – which includes architectures with 8, 16, 20, 24 and 32-bit buses. This inevitably requires some pointer arithmetic. When pointer arithmetic is used, the arithmetic result is programatically checked for correctness.
- The trace macros are, by default, empty, so do not generate any code. Therefore, MISRA compliance checking is performed with dummy macro definitions.
- MISRA rules are turned off on a line by line basis, as deemed appropriate (that is, complying with the rule is deemed to create less appropriate code for a deeply embedded system than carefully not complying). Each such occurrence is accompanied by a justification using the special pc-lint MISRA comment mark-up syntax.

FreeRTOS builds with many different compilers, some of which are more advanced than others. For that reason FreeRTOS does not use any of the features or syntax that have been introduced to the C language by or since the C99 standard. The one exception to this is the use of the `stdint.h` header file. The `FreeRTOS/Source/include` directory contains a file called `stdint.readme` that can be renamed `stdint.h` to provide the minimum stdint type definitions necessary to build FreeRTOS – should your compiler not provide its own.

Testing

This section describes the tests performed on common code (the code located in the `FreeRTOS/Source` directory, that is built by all FreeRTOS kernel ports), and the tests performed on the portable layer code (the code located in subdirectories of the `FreeRTOS/Source/portable` directory).

- **Common code**

  The standard demo/test files attempt to provide 'branch' test coverage (in most cases this actually achieves 'condition' coverage because the kernel's coding style deliberately keeps conditions simple specifically for this purpose), whereby tests ensure both 'true' and 'false' paths through each decision are exercised. 'branch' coverage is measured using GCOV by defining the mtCOVERAGE_TEST_MARKER() macro to a NOP (no operation) instruction in the 'else' path of each 'if()' condition if the 'else' path would otherwise be empty. mtCOVERAGE_TEST_MARKER() is only

Port layer code is tested using 'reg test' tasks, and, for ports that support interrupt nesting, the 'interrupt queue' tasks.

The 'reg test' tasks create multiple (normally two) tasks that first fill all the CPU registers with known values, then continuously check that every register maintains its expected known value as the other tests execute continuously (soak test). Each reg test task uses unique values.

The 'interrupt queue' tasks perform tests on interrupts of different priorities that nest at least three deep. Macros are used to insert artificial delays into pertinent points within the code to ensure the desired test coverage is achieved.

It is worth noting that the thoroughness of these tests have been responsible for finding bugs in silicon on multiple occasions.

Naming Conventions

The RTOS kernel and demo application source code use the following conventions:

- Variables
  - Variables of type *uint32_t* are prefixed *ul*, where the 'u' denotes 'unsigned' and the 'l' denotes 'long'.
  - Variables of type *uint16_t* are prefixed *us*, where the 'u' denotes 'unsigned' and the 's' denotes 'short'.
  - Variables of type *uint8_t* are prefixed *uc*, where the 'u' denotes 'unsigned' and the 'c' denotes 'char'.
  - Variables of non stdint types are prefixed *x*. Examples include BaseType_t and TickType_t, which are portable layer defined typedefs for the natural or most efficient type for the architecture and the type used to hold the RTOS tick count respectively.
  - Unsigned variables of non stdint types have an additional prefix *u*. For example variables of type UBaseType_t (unsigned BaseType_t) are prefixed *ux*.
  - Variables of type *size_t* are also prefixed *x*.>
  - Enumerated variables are prefixed *e*
  - Pointers have an additional prefixed *p*, for example a pointer to a uint16_t will have prefix *pus*.
  - In line with MISRA guides, unqualified standard *char* types are only permitted to hold ASCII characters and are prefixed *c*.
  - In line with MISRA guides, variables of type *char ** are only permitted to hold pointers to ASCII strings and are prefixed *pc*.
- Functions
  - File scope static (private) functions are prefixed with *prv*.
  - API functions are prefixed with their return type, as per the convention defined for variables, with the addition of the prefix *v* for *void*.
  - API function names start with the name of the file in which they are defined. For example vTaskDelete is defined in tasks.c, and has a void return type.
- Macros
  - Macros are pre-fixed with the file in which they are defined. The pre-fix is lower case. For example, configUSE_PREEMPTION is defined in FreeRTOSConfig.h.
  - Other than the pre-fix, macros are written in all upper case, and use an underscore to separate words.

Data Types

Only stdint.h types and the RTOS's own typedefs are used, with the following exceptions:

- char

  In line with MISRA guides, unqualified char types are permitted, but only when they are used to hold ASCII characters.

- char *

  In line with MISRA guides, unqualified character pointers are permitted, but only when they are used to point to ASCII strings. This removes the need to suppress benign compiler warnings when standard

There are four types that are defined for each port. These are:

- TickType_t

  If configUSE_16_BIT_TICKS is set to non-zero (true), then TickType_t is defined to be an unsigned 16-bit type. If configUSE_16_BIT_TICKS is set to zero (false), then TickType_t is defined to be an unsigned 32-bit type. See the customisation section of the API documentation for full information.

  32-bit architectures should always set configUSE_16_BIT_TICKS to 0.

- BaseType_t

  This is defined to be the most efficient, natural, type for the architecture. For example, on a 32-bit architecture BaseType_t will be defined to be a 32-bit type. On a 16-bit architecture BaseType_t will be defined to be a 16-bit type. If BaseType_t is define to char then particular care must be taken to ensure signed chars are used for function return values that can be negative to indicate an error.

- UBaseType_t

  This is an unsigned BaseType_t.

- StackType_t

  Defined to the type used by the architecture for items stored on the stack. Normally this would be a 16-bit type on 16-bit architectures and a 32-bit type on 32-bit architectures, although there are some exceptions. Used internally by FreeRTOS.

Style Guide

- Indentation

  Tab characters are used to indent. One tab equals four spaces.

- Comments

  Comments never pass column 80, unless they follow, and describe, a parameter.

  C++ style double slash (//) comments are not used.

- Layout

  The FreeRTOS source code lay out is designed to be as easy to view and read as possible. The code snippets below show first the file layout, then the C code formatting.

```
/* Library includes come first… */
#include <stdlib.h>
/* …followed by FreeRTOS includes… */
#include "FreeRTOS.h"
/* …followed by other includes. */
#include "HardwareSpecifics.h"
/* #defines comes next, bracketed where possible. */
#define A_DEFINITION    ( 1 )

/*
 * Static (file private) function prototypes appear next, with comments
 * in this style - with each line starting with a '*'.
 */
static void prvAFunction( uint32_t ulParameter );

/* File scope variables are the last thing before the function definitions.
Comments for variables are in this style (without each line starting with
a '*'). */
static BaseType_t xMyVariable;

/* The following separate is used after the closing bracket of each function,
```

```
void vAFunction( void )
{
    /* Function definition goes here - note the separator after the closing
    curly bracket. */
}
/*-----------------------------------------------------------*/

static UBaseType_t prvNextFunction( void )
{
    /* Function definition goes here. */
}
/*-----------------------------------------------------------*/
```

File Layout

```
/* Function names are always written on a single line, including the return
type.  As always, there is no space before the opening parenthesis.  There
is a space after an opening parenthesis.  There is a space before a closing
parenthesis.  There is a space after each comma.  Parameters are given
verbose, descriptive names (unlike this example!).  The opening and closing
curly brackets appear on their own lines, lined up underneath each other. */
void vAnExampleFunction( long lParameter1, unsigned short usParameter2 )
{
/* Variable declarations are not indented. */
uint8_t ucByte;

    /* Code is indented.  Curly brackets are always on their own lines
    and lined up underneath each other. */
    for( ucByte = 0U; ucByte < fileBUFFER_LENGTH; ucByte++ )
    {
        /* Indent again. */
    }
}

/* For, while, do and if constructs follow a similar pattern.  There is no
space before the opening parenthesis.  There is a space after an opening
parenthesis.  There is a space before a closing parenthesis.  There is a
space after each semicolon (if there are any).  There are spaces before and
after each operator.  No reliance is placed on operator precedence -
parenthesis are always used to make precedence explicit.  Magic numbers,
other than zero, are always replaced with a constant or #defined constant.
The opening and closing curly brackets appear on their own lines. */
for( ucByte = 0U; ucByte < fileBUFFER_LENGTH; ucByte++ )
{
}

while( ucByte < fileBUFFER_LENGTH )
{
}

/*  There must be no reliance on operator precedence - every condition in a
multi-condition decision must uniquely be bracketed, as must all
sub-expressions. */
if( ( ucByte < fileBUFFER_LENGTH ) && ( ucByte != 0U ) )
{
    /* Example of no reliance on operator precedence! */
    ulResult = ( ( ulValue1 + ulValue2 ) - ulValue3 ) * ulValue4;
}

/* Conditional compilations are laid out and indented as per any
other code. */
```

```
}
#endif

A space is placed after an opening square bracket, and before a closing
square bracket.
ucBuffer[ 0 ] = 0U;
ucBuffer[ fileBUFFER_LENGTH - 1U ] = 0U;

Formatting of C Constructs
```

[Back to the top] [About FreeRTOS] [Privacy] [Sitemap] [Report an error on this page]