# FreeRTOS Tutorial

From Embedded Systems Learning Academy

## Contents



# Introduction

This article is about learning FreeRTOS. The FreeRTOS **sample** project running on **SJ One Board** is used as reference, but any FreeRTOS project on any controller can benefit from this article.

Notes about the FreeRTOS Sample Project:

- The Development Package ZIP File contains the FreeRTOS sample project, however, the latest copy can always be downloaded from: https://sourceforge.net/projects/armdevpkg/files
- This article assumes you know how to compile and load a sample project to the SJ-One Board.

## Screencast

I created some screencasts to quickly go through FreeRTOS, however, I **HIGHLY** encourage you to read this article in full first.

- **Tasks and Queues**

**FreeRTOS tasks and Queues (http://www.youtube.com/watch?v=8lIpI30Tj-g)**
**FreeRTOS Queues (http://www.youtube.com/watch?v=yHfDO_jiIFw)**

- **Semaphores :**

  **FreeRTOS Mutex (http://www.youtube.com/watch?v=PjDHn_G078k)**
  **FreeRTOS Binary Semaphore (http://www.youtube.com/watch?v=grXuVMttVuU)**
  **FreeRTOS Interrupt Processing using Binary Semaphore (http://www.youtube.com/watch?v=06TH2NgrKkA)**

- **C++ Wrapper for FreeRTOS tasks (http://www.youtube.com/watch?v=4gawgXminv4)**

  Use this tutorial after mastering the basic FreeRTOS concepts
  The source code is included in **SJSU_Dev** development package.

- Other FreeRTOS Modules:

  **FreeRTOS Event Groups (https://www.youtube.com/watch?v=31g6_uJ4kdE)**
  **FreeRTOS Queue Set (https://www.youtube.com/watch?v=Rf-yobhAFy0)**
  **FreeRTOS Trace Analyzer (https://www.youtube.com/watch?v=-FX2LBN1bhI)**

# What is an OS

An Embedded Operating System like FreeRTOS is nothing but software that provides multitasking facilities. FreeRTOS allows to run multiple tasks and has a simple scheduler to switch between tasks. Here are some of the FreeRTOS features:

- Priority-based multitasking capability
- Queues to communicate between multiple tasks
- Semaphores to manage resource sharing between multiple tasks
- Utilities to view CPU utilization, stack utilization etc.

# FreeRTOS main() and Tasks

## The MAIN Function

The main function in FreeRTOS based project is nothing but a function that creates tasks. FreeRTOS will let you multi-task based on your tasks and their priority. Remember that a "task" is simply a "function" name of type: `void my_task(void* p)`

## What is Stack Memory?

Before you create a task, you need to know what is stack memory. Every variable you declare uses memory on the stack. This memory is generally preferred over heap allocated memory that comes from `malloc` or `new` operators.

What uses Stack Memory?

- Local Variables of the task.
- Function calls (function parameters + function return address)
- Local variables of functions your task calls.

How is stack managed?

- Every time you declare a variable, stack pointer moves down.
  - If you declare an int, your assembly code will generate: `SUB SP, 1`
  - If you declare `char mem[128]` your assembly code will generate: `SUB SP, 32` assuming 32-bit machine
- That is why you should use curly braces and limit the scope of variables as much as possible. This way, every time a variable goes out of scope, you will see something like: `ADD SP, ##` indicating memory released.
- This is why stack memory is preferred over heap because stack uses just two ADD and SUB instructions to manage memory whereas heap uses rather expensive `malloc` operations. Furthermore, `malloc` fragments your memory and in a smaller system, fragmented memory may result in `malloc` returning NULL pointers.

Let's start with examples on how to estimate your stack memory:

```
void hello_world_task(void* p)
{
    char mem[128];
    while(1) {
    }
}
//The task above uses 128 bytes of stack.
```

```
void hello_world_task(void* p)
{
    char mem[128];
    int  int_mem[128]; // 4 bytes per int
    while(1) {
    }
}
// The task above uses 128+ (128*4) bytes of stack.
```

```
void hello_world_task(void* p)
{
    char mem[128];
    while(1) {
        foo(); // Assume foo uses 128 bytes of stack.
    }
}
// The task above uses 128+128 bytes of stack.
```

```
void hello_world_task(void* p)
{
    char mem[128];
    while(1) {
        if(...) {
            char mem_one[128];
        }
        else(...) {
            char mem_two[256];
        }
    }
}
/* The task above uses 128 + 256 bytes of stack.
 * Note that it is not 128+128+256, because only one branch statement
 * will execute and in the worst case, branch two's code will end up
 * using 128+256 bytes of stack.
 */
```

```
void hello_world_task(void* p)
{
    char *mem = char* malloc(128);
    while(1) {
    }
}
// The task above uses just 4 bytes of stack (to hold mem pointer)
// The actual memory comes from HEAP, which is not part of the stack.
```

## Task Stack Size

The stack size of a task depends on the memory consumed by its local variables and function call depth. Please note that if your task (or function) uses printf, it consumes around 1024 bytes of stack. **At minimum however, you would need at least 512 bytes + your estimated stack space above**. If you don't allocate enough stack space, your CPU will run to an exception and/or freeze.

You should definitely read the following article to study the memory layout:

- Stack Heap Walkthrough

## Controlling Tasks

In FreeRTOS, you have precise control of when tasks will use the CPU. The rules are simple:

- Task with highest priority will run first, and never give up the CPU until it sleeps
- If 2 or more tasks with the same priority do not give up the CPU (they don't sleep), then FreeRTOS will share the CPU between them (time slice).

Here are some of the ways you can give up the CPU:

- **`vTaskDelay()`** This simply puts the task to "sleep"; you decide how much you want to sleep.
- **`xQueueSend()`** If the Queue you are sending to is full, this task will sleep (block).
- **`xQueueReceive()`** If the Queue you are reading from is empty, this task will sleep (block).

- **`xSemaphoreTake()`** You will sleep if the semaphore is taken by somebody else.

Remember that each function given above takes a parameter that decides how long you are willing to sleep. You could use this parameter as a timeout. For example, your logic may be: "I'm going to wait 1 second to receive something from the queue, otherwise I will <do whatever>".

### Simple Task

Below is a simple task example that prints a message once a second. Note that `vTaskStartScheduler()` never returns and FreeRTOS will begin servicing the tasks at this point. Also note that every task must have an **infinite loop and NEVER EXIT**.

```
void hello_world_task(void* p)
{
    while(1) {
        puts("Hello World!");
        vTaskDelay(1000);
    }
}

int main()
{
    xTaskCreate(hello_world_task, (signed char*)"task_name", STACK_BYTES(2048), 0, 1, 0);
    vTaskStartScheduler();

    return -1;
}
```

### Terminal Task

The FreeRTOS sample project creates "terminal" task that allows you to interact with the serial port. You can type "help" and press enter in **Hercules** program to see the commands supported by the terminal task. You can, of course, add more commands to it. You should now pause reading this article and now look at how the **terminal** task works. Here is a screenshot of terminal task interaction: **TODO: Add screenshot**

## FreeRTOS Inter-task Communication

### Queue Communication

You can communicate between tasks by using Queues or Semaphores. Let's create an example to communicate between two tasks based on the Software Framework in the sample project. Here's some sample code:

```
// Global Queue Handle
QueueHandle_t qh = 0;

void task_tx(void* p)
{
    int myInt = 0;
    while(1)
    {
```

```
        myInt++;
        if(!xQueueSend(qh, &myInt, 500)) {
            puts("Failed to send item to queue within 500ms");
        }
        vTaskDelay(1000);
    }
}

void task_rx(void* p)
{
    int myInt = 0;
    while(1)
    {
        if(!xQueueReceive(qh, &myInt, 1000)) {
            puts("Failed to receive item within 1000 ms");
        }
        else {
            printf("Received: %u\n", myInt);
        }
    }
}

int main()
{
    qh = xQueueCreate(1, sizeof(int));

    xTaskCreate(task_tx, (signed char*)"t1", STACK_BYTES(2048), 0, 1, 0);
    xTaskCreate(task_rx, (signed char*)"t2", STACK_BYTES(2048), 0, 1, 0);
    vTaskStartScheduler();

    return -1;
}
```

### Explanation

Note the following items:

- In **main()**, we create the Queue before creating tasks, otherwise sending to un-initialized Queue will crash the system.
- In **task_tx()**, we send one item every second, and if the queue is full, we print a failure message.
- In **task_rx()**, we receive one item, and we do not use vTaskDelay(). This is because if there is nothing in the queue, FreeRTOS will sleep(or block) this task from running. The timeout itself in xQueueReceive() allows us to sleep for 1000ms but wake-up if an item is available in the queue earlier.
- If the priority of the receiving queue(task_rx()) is higher, FreeRTOS will switch tasks the moment xQueueSend() happens, and the next line inside task_tx() will not execute since CPU will be switched over to task_rx().

### Practical Example

A practical example of a queue may be to start another task to start doing its work while the primary task continues doing its own work independently. In the following example, we demonstrate how a terminal task can kick-off another task to begin playing an mp3 song while it operates independently to handle the next command from a terminal (user input).

```
void terminal_task(void* p)
```

```
{
     // Assume you got a user-command to play an mp3:
     xQueueSend(song_name_queue, "song_name.mp3", 0);

     ...
}

void mp3_play_task(void* p)
{
    char song_name[32];
    while(1) {
        if(xQueueReceive(song_name_queue, &song_name[0], portMAX_DELAY)) {
            // Start to play the song.
        }
    }
}
```

## Semaphores

Semaphores are meant to limit access to resources, but there are many applications. There are also many types of semaphores and the text below discusses some of them and their application.

### Mutex

One of the best example of a mutex is to guard a resource or a door with a key. For instance, let's say you have an SPI BUS, and only one task should use it at a time. Mutex provides mutual exclusion with priority inversion mechanism (http://en.wikipedia.org/wiki/Priority_inversion). Mutex will only allow ONE task to get past **xSemaphoreGet()** operation and other tasks will be put to sleep if they reach this function at the same time.

```
// In main(), initialize your Mutex:
SemaphoreHandle_t spi_bus_lock = xSemaphoreCreateMutex();

void task_one()
{
    while(1) {
        if(xSemaphoreGet(spi_bus_lock, 1000)) {
            // Use Guarded Resource

            // Give Semaphore back:
            xSemaphoreGive(spi_bus_lock);
        }
    }
}
void task_two()
{
    while(1) {
        if(xSemaphoreGet(spi_bus_lock, 1000)) {
            // Use Guarded Resource

            // Give Semaphore back:
            xSemaphoreGive(spi_bus_lock);
        }
    }
}
```

In the code above, only ONE task will enter its xSemaphoreGet() branch. If both tasks execute the statement at the same time, one will get the mutex, the other task will sleep until the mutex is returned by the task that was able to obtain it in the first place.

### Binary Semaphore

Binary semaphore can also be used like a mutex, but binary semaphore doesn't provide priority inversion mechanism. Binary semaphores are better suited for helper tasks for interrupts. For example, if you have an interrupt and you don't want to do a lot of processing inside the interrupt, you can use a helper task. To accomplish this, you can perform a semaphore give operation inside the interrupt, and a dedicated ask will sleep or block on **xSemaphoreGet()** operation.

```
// Somewhere in main() :
SemaphoreHandle_t event_signal;
vSemaphoreCreateBinary( event_signal ); // Create the semaphore
xSemaphoreTake(event_signal, 0);        // Take semaphore after creating it.


void System_Interrupt()
{
    xSemaphoreGiveFromISR(event_signal);
}

void system_interrupt_task()
{
    while(1) {
        if(xSemaphoreTake(event_signal, 9999999)) {
            // Process the interrupt
        }
    }
}
```

The above code shows example of a deferred interrupt processing. The idea is that you don't want to process the interrupt inside System_Interrupt() because you'd be in a critical section with system interrupts globally disabled, therefore, you can potentially lock up the system or destroy real-time processing if the interrupt processing takes too long.

Another way to use binary semaphore is to wake up one task from another task by giving the semaphore. So the semaphore will essentially act like a signal that may indicate: "Something happened, now go do the work in another task". In the sample project, binary semaphore is used to indicate when an I2C read operation is done, and the interrupt gives this semaphore. **Note that when you create a binary semaphore in FreeRTOS, it is ready to be taken, so you may want to take the semaphore after you create it such that the task waiting on this semaphore will block until given by somebody.**

### Counting Semaphore

Counting semaphores are suited for applications in which more than one user is allowed access to a resource. For example, if you have a parking garage of 10 cars, you can allow 10 semaphore access. Each car entering a garage will take 1 semaphore until 10 cars take 10 semaphores and no more cars will be allowed access to the garage.

# FAQ

- **I wrote my own version of Queue, I don't want to use FreeRTOS queue since it is not efficient**

  Be careful here. FreeRTOS queues may be more expensive, but they provide benefits your queue may not provide. FreeRTOS's queues can switch tasks upon Queue send and receive, and your tasks will be managed better and sleep as appropriate whereas your own version likely doesn't integrate well with FreeRTOS. For example, FreeRTOS may switch context inside of xQueueSend() if it finds that someone with higher priority was waiting for an item in this queue.

- **What if I send an item on FreeRTOS queue and my item goes out of scope?**

  FreeRTOS copies value of the item you send, so this is perfectly okay.

- **If I use deferred interrupt processing, but still want to process interrupt quickly, what can I do?**

  You can set the priority of the deferred interrupt task as highest, and as soon as interrupt gives the binary semaphore and exit, FreeRTOS will switch context to your interrupt task.

- **I have a lot of little tasks but end up using a lot of stack memory for each task. What can I do?**

  First, think about consolidating tasks. If your tasks do a bunch of things every second, then combine the processing into a single task. If you have a number of things happening periodically, consider using a FreeRTOS timer. FreeRTOS timers use common stack but provide independent timers.

- **How much stack space am I really using?**

  Use FreeRTOS's: `vTaskList()` function to get a measure of the stack space. The sample projects are specifically modified such that this function will report stack free in bytes, along with CPU utilization of each task.

- **I'm using a delay function and CPU utilization is very high**

  Your delay function is probably a busy-wait loop. Use FreeRTOS's vTaskDelay(), which is actually smart enough to put the task to sleep and wake it up precisely when the timeout is done. If a task uses little processing every second and you use vTaskDelay(1000); your CPU utilization will be near zero percent.

- **I don't understand the difference between vTaskDelay() and vTaskDelayUntil()**

  vTaskDelay() will delay by the defined amount, and if you wanted a precise periodic processing of one second, this function might not work as you'd expect. In the example below, even if you wanted sensor update of once per 1000ms, your actual rate would be anywhere from 1005 to 1050ms because update_sensor() function's processing time can vary. In this example, if we switch over to vTaskDelayUntil(), we will be updating sensors exactly once per 1000ms.

```
void my_task(void* p)
{
    while(1) {
        update_sensors(); // Assume this function can take 5-50ms
        vTaskDelay(1000);
```

```
    }
}
```

# Going Beyond

The basic tasks show you around FreeRTOS centric system and this is just the starting point. For example, if you are creating a relay switch that gets turned on based on the light level, you may want to create a task, get the light reading from the light sensor, and control the relay. Logically thinking, you may want to monitor the light reading once a second, so don't forget to use vTaskDelay().

As your project gets more complex, you will determine by experience that which tasks can be put to lower priority and which tasks need high priority. By prioritizing the tasks, you can create a deterministic system and guarantee real-time operation of critical tasks while completing "background" processing using lower priority tasks.

Valuable Reads:

- Read 2012 SJ One Hello World Sample Project to get newbie oriented details about how to read sensor values of your SJ One Board.
- Read FreeRTOS API (http://www.freertos.org/a00106.html) to unleash the power of this OS.