Chapter 2: Understanding RTOS

After completing this chapter, you will understand what an RTOS is and how to use one. We will cover usage of FreeRTOS and Mbed OS, FreeRTOS, in addition to the RTOS abstraction layer.

2.1 INTRODUCTION TO RTOS 2
2.1.1 General 2
2.1.2 Tasks/Threads 2
2.1.3 Potential Pitfalls 2
2.1.4 Queue 3
2.1.5 Semaphore 3
2.1.6 Mutex 3
2.1.7 Events/Notifications 3
2.1.8 Timers 3
2.1.9 Interrupts 4
2.2 RTOS ABSTRACTION LAYER 4
2.3 FREERTOS 5
2.3.1 Getting Started 5
2.3.2 FreeRTOSConfig.h 6
2.3.3 Configuration Settings 6
2.3.4 Kernel Interface Functions 8
2.3.5 Interrupts 9
2.3.6 Tasks 10
2.3.7 Queues 11
2.3.8 Semaphores and Mutexes 12
2.3.9 Task Notifications 14
2.3.10 References 15

2.1 Introduction to RTOS

2.1.1 General

The purpose of an RTOS is to reduce the complexity of writing embedded firmware that has multiple asynchronous, response-time-critical tasks that have overlapping resource requirements. For example, you might have a device that is reading and writing data to/from a connected network, an external filesystem, and peripherals. Making sure that you deal with the timing requirement of responding to network requests while continuing to support the peripherals can be complex and, therefore, error prone. By using an RTOS, you can separate the system functions into separate tasks (called **threads or tasks**) and develop them in a somewhat independent fashion.

2.1.2 Tasks/Threads

A thread/task is a specific execution context. The RTOS maintains a list of threads that are idle, halted, or running, as well as which task needs to run next (based on priority) and at what time. This function in the RTOS is called the scheduler. There are two major schemes for managing which threads/tasks/processes are active in operating systems: preemptive and co-operative.

In preemptive multitasking, the CPU completely controls which task is running and can stop and start them as required. In this scheme, the scheduler uses CPU protected modes to wrest control from active tasks, halt them, and move onto the next task. Higher priority tasks will be prioritized to run before lower priority tasks. That is, if a task is running and a higher priority task requests a turn, the RTOS will suspend the lower priority task and switch to the higher priority task. Preemptive multitasking is the scheme that is used in Windows, Linux etc.

In co-operative multitasking, each process must be a good citizen and yield control back to the RTOS. There are several mechanisms for yielding control such as RTOS delay functions, semaphores, mutexes, and queues (which we will discuss later in this document).

Most embedded RTOSes are configured in a hybrid mode. It is preemptive for higher priority tasks and “round robin” for tasks of equal priority. Higher priority tasks will always run at the expense of lower priority tasks, so it is important to yield control to give lower priority tasks a turn. If not, tasks that don't yield control will prevent lower or equal priority tasks from running at all. It is good practice to have some form of yield control mechanism in every thread to prevent such situations.

2.1.3 Potential Pitfalls

All of this sounds great, but there are three serious bugs which can easily be created in these types of systems and these bugs can be very hard to find. These bugs are all caused by side effects of interactions between the threads. The big three are:

- Cyclic dependencies which can cause deadlocks.
- Resource conflicts when sharing memory and sharing peripherals which can cause erratic non-deterministic behavior.
- Difficulties in executing inter-process communication.

To handle those problems, every RTOS gives you mechanisms to deal with these problems, specifically semaphores, mutexes, and queues which we will discuss in a minute.

2.1.4 Queue

A queue is a thread-safe mechanism to send data to another thread. The queue is a FIFO; you read from the front and you write to the back. If you try to read a queue that is empty your thread will suspend until something is written into it. The payload in a queue (size of each entry) and the size of the queue (number of entries) is user configurable at queue creation time. To add data to a queue, you will do a "push" or "send" operation. To pull data from a queue, you use either "pop" or "receive".

2.1.5 Semaphore

A semaphore is a signaling mechanism between threads. The name semaphore (originally sailing ship signal flags) was applied to computers by Dijkstra in a paper about synchronizing sequential processes. Semaphores can either be binary or counting. In the case of a counting semaphore, they are usually implemented as a simple unsigned integer. When you "set" or "give" a counting semaphore it increments the value of the semaphore. When you "get" or "take" a semaphore it decrements the value, but if the value is 0 the thread will SUSPEND itself until the semaphore is set. So, you can use a semaphore to signal between threads that something is ready. For instance, you could have a "collectDataThread" that reads data from a sensor and a "sendData" thread that sends the data up to the cloud. The sendData thread would "get" the semaphore which will suspend the thread UNTIL the collectDataThread "sets" the semaphore when it has new data available that needs to be sent.

With a binary semaphore, no matter how many times the set/give function is called, there can only be one get/take event until the semaphore is cleared. With a counting semaphore, there can be as many get/take events as there have been set/give events. These can both useful in different situations.

2.1.6 Mutex

Mutex is an abbreviation for "Mutual Exclusion". A mutex is a lock on a specific resource - if you request a mutex on a resource that is already locked by another thread, then your thread will go to sleep until the lock is released. Without a mutex, you may see strange behavior if two threads try to access the same shared resource at the same time or when one thread starts to use a shared resource but is then preempted by another thread that uses the same resource before the first thread is done. Shared resources can be things like a UART or an I2C interface, a shared block of memory, etc.

2.1.7 Events/Notifications

Events and notifications can be likened to that of a notice board, where multiple tasks have visibility to it. Tasks can check the values in the event or notification objects to determine if a certain event/notification happened and take appropriate action. Once an event is no longer required in the system, then it can be cleared.

2.1.8 Timers

An RTOS timer allows you to schedule a function to run at a specified interval - e.g. send your data to the cloud every 10 seconds. When you setup a timer you specify the function you want run and how often you want it run.

Note that there is a single execution of the function every time the timer expires rather than a continually executing thread, so the function should NOT have an infinite loop – it should just run and exit each time the timer calls it.

Since the timer is a function, not a thread, make sure you don't exit the main application thread if your project has no other active threads because that will crash the system.

2.1.9 Interrupts

An interrupt is a signal to the processor that an event has occurred, and that immediate attention is required. An interrupt is handled with an interrupt service routine (ISR). An interrupt is a function, not a thread, so it should do what it needs to do and then exit (i.e. no infinite loop). It is also important not to call any long blocking functions inside of an ISR since it will prevent other tasks from running until it completes.

Interrupts have priorities just like threads. A higher priority interrupt can cause an ISR for a lower priority interrupt to suspend until the higher priority ISR completes. This is called a nested interrupt.

2.2 RTOS Abstraction Layer

In order to help make middleware applications as portable as possible within ModusToolbox, we have developed a thin RTOS Abstraction library for PSoC 6. This library provides a unified API around the actual RTOS. This allows middleware libraries to be written independent of the RTOS actually selected for the application. The abstraction layer provides access to all the standard RTOS resources described above.

While the primary purpose of the library is for middleware, the abstraction layer can just as easily be used by the application code. However, since the application code generally knows what RTOS is being used, this is typically an unnecessary overhead.

The RTOS abstraction layer functions generally all work the same way. The basic process is:

1. Include the *cyabs_rtos.h* header file so that you have access to the RTOS functions.
2. Declare a variable of the right type (for example, `cy_mutex_t`)
3. Call the appropriate create or initialize function (for example, `cy_rtos_init_mutex`). Provide it with a reference to the variable that was created in step 2.
4. Access the RTOS object using one of the access functions (for example, `cy_rtos_set_mutex`).
5. If you don't need it anymore, free up the pointer with the appropriate de-init function (for example, `cy_rtos_deinit_mutex`).

Note: All these functions need a pointer, so it is generally best to declare these "shared" resources as static global variables within the file that they are used. You can find the full documentation for the RTOS Abstraction layer on GitHub at:

https://github.com/cypresssemiconductorco/abstraction-rtos

2.3 FreeRTOS

Pulled from the FreeRTOS website:

> "Developed in partnership with the world's leading chip companies over a 15-year period, and now downloaded every 175 seconds, FreeRTOS is a market-leading real-time operating system (RTOS) for microcontrollers and small microprocessors. Distributed freely under the MIT open source license, FreeRTOS includes a kernel and a growing set of libraries suitable for use across all industry sectors. FreeRTOS is built with an emphasis on reliability and ease of use."

Basically, FreeRTOS has been around for a long time and lots of customers have used it and are already comfortable with it. All APIs are documented online at https://www.freertos.org/a00106.html.

2.3.1 Getting Started

To create a FreeRTOS application you need to do the following:

1. Create an application to start with. This is the application that you will add FreeRTOS into.
2. Add the FreeRTOS library to the application. In ModusToolbox, you can do this with the Library Manager (we'll discuss this in Chapter 5a). This will also add the dependencies *clib-support* and *abstraction-rtos*.
3. Add FREERTOS to the COMPONENTS variable in the Makefile.
4. Copy the *FreeRTOSConfig.h* file from the *../mtb_shared/freertos/<version>/Source/portable/COMPONENT_<core>* directory to your project. This is a template containing settings to customize how FreeRTOS behaves. More on that in a minute.
5. Open the copied *FreeRTOSConfig.h* file. Remove the #warning line and make any other setting changes required for your application.
6. In the *main.c* file:
    - Include the required header files:

```
#include "cyhal.h"
#include "cybsp.h"
#include "FreeRTOS.h"
#include "task.h"
```

    - Create functions for any tasks that your application requires. For example:

```
void MyFunction(void *arg)
{
    (void)arg;

    /* Initialize hardware */

    for (;;)
    {
        /* Do something */
        MyActions();
```

```
            /* Allow other tasks to execute */
            vTaskDelay(500);
        }
    }
```

- Setup and start any RTOS objects such as queues and semaphores.
- Create a task for each of your functions and start the RTOS scheduler:

```
int main(void)
{
   /* Initialize the device and board peripherals */
    if (CY_RSLT_SUCCESS != cybsp_init())
    {
        /* Unable to initialize BSP so HALT */
        CY_ASSERT(0);
    }

    if (pdPASS == xTaskCreate(MyFunction,  // Task function
                              "Task Name", // Task name
                              1024,        // Task stack size
                              NULL,        // Parameters passed to task
                              5,           // Task priority
                              NULL)        // Task handle
        )
    {
        vTaskStartScheduler();
    }

    /* vTaskStartScheduler should never return.
       If we get here, then there was a serious error */
    CY_ASSERT(0);
}
```

2.3.2 FreeRTOSConfig.h

This file is where you configure how FreeRTOS will work. It also contains the system specific configuration. A predefined template you can modify is in the *../mtb_shared/freertos/Source/portable/COMPONENT_<core>* directory. When you build this template will generate a warning. You should make a copy of this file, save it in your project directory and remove the warning. This file controls the APIs you can call as well as other FreeRTOS settings. For many applications, the template will work just fine as-is.

2.3.3 Configuration Settings

The FreeRTOS Customization documentation describes the configuration options available in the FreeRTOSConfig.h file. The following configuration values are specific to a PSoC 6 MCU FreeRTOS port:

configCPU_CLOCK_HZ

This parameter passes the frequency of the MCU core clock required for FreeRTOS system timer configuration. The pre-configured *FreeRTOSConfig.h* file provided with the FreeRTOS library sets the `configCPU_CLOCK_HZ` value as `SystemCoreClock`, which is calculated by the PSoC 6 MCU system startup code. You should not need to modify this parameter.

configMAX_SYSCALL_INTERRUPT_PRIORITY

This parameter sets the highest interrupt priority from which to call interrupt-safe FreeRTOS functions. Calling FreeRTOS functions from an interrupt with a priority higher than `configMAX_SYSCALL_INTERRUPT_PRIORITY` causes FreeRTOS to generate an exception. To avoid this exception, you can do one of the following:

- Reduce all interrupt priorities to `configMAX_SYSCALL_INTERRUPT_PRIORITY` or lower.
- Trigger an interrupt with a priority less or equal to `configMAX_SYSCALL_INTERRUPT_PRIORITY` and call the FreeRTOS functions from this interrupt handler.
- Call the FreeRTOS functions from the `traceTASKSWITCHEDOUT` macro. See this post on FreeRTOS support forums.

Note: Code that writes to the flash block (including the Flash/Bluetooth Low Energy (LE)/Emulated EEPROM/DFU) uses system calls and thus the system pipe (IPC) interrupt to do so. If the IPC interrupt priority is less than or equal to `configMAX_SYSCALL_INTERRUPT_PRIORITY` and another task is running at the same or higher priority than the IPC or if a critical section is entered during the flash write this can cause issues with both blocking and partially blocking flash writes. See the PSoC 6 PDL Flash Driver documentation for more details.

configHEAP_ALLOCATION_SCHEME

This parameter specifies the memory management scheme. The FreeRTOS kernel requires RAM for each created task, queue, mutex, software timer, semaphore, or event group. To manage used memory, FreeRTOS implements several different memory management schemes so that you can choose the most suitable for your application.

Each scheme is documented in the FreeRTOS Memory Management topic. The memory management implementation files are in the freertos/Source/portable/MemMang directory.

The available memory management schemes:

- `HEAP_ALLOCATION_TYPE1` (heap_1) - The simplest, does not permit memory to be freed.
- `HEAP_ALLOCATION_TYPE2` (heap_2) - Permits memory to be freed, but not does coalesce adjacent free blocks.
- `HEAP_ALLOCATION_TYPE3` (heap_3) - Simply wraps a standard malloc and free for task safety.
- `HEAP_ALLOCATION_TYPE4` (heap_4) - Coalesces adjacent free blocks to avoid fragmentation. Includes the absolute address-placement option.
- `HEAP_ALLOCATION_TYPE5` (heap_5) - As per heap_4, with the ability to span the heap across multiple non-adjacent memory areas.
- `NO_HEAP_ALLOCATION` - Disables the memory management, used for applications with static memory allocation.

In the heap_3 memory scheme, your linker file must specify a sufficient size of heap and stack, and your firmware must implement and use `malloc` and `free` to allocate and release memory.

In the other memory schemes, the RTOS itself allocates a stack and heap. For these schemes, the stack defined in the Board Support Package (BSP) linker file is used only by the `main` function and the functions it calls. We'll talk more about BSPs in Chapter 5a.

configTOTAL_HEAP_SIZE

This parameter specifies a total amount of RAM available for the FreRTOS heap. This parameter ignored when heap_3 memory scheme is used.

configMINIMAL_STACK_SIZE

This parameter specifies the size of the stack used by the idle task. It is not recommended to reduce the default parameter value.

2.3.4 Kernel Interface Functions

While the API lists a dozen kernel functions, there are only 6 needed in day to day use, and 4 of them are paired sets. They are broken down by what type of task scheduler you're using. The FreeRTOSConfig.h file controls the scheduler type; see the FreeRTOS Customization documentation.

Cooperative

- `taskYIELD` – Lets the kernel schedule someone else. Note this is not like a delay which blocks the task for a specified period of time. Instead it allows tasks at the same or higher priority to run. Lower priority tasks will never run because of a yield.

Note: This can be used in a preemptive system if there are specific places where you want to let other tasks run, instead of waiting for the next system tick to kick you out. For example, you may want to yield before starting a critical section.

If you want to allow any task to run, even a lower priority task, you can use vTaskDelay. We'll cover that in just a minute.

Preemptive

There are critical section functions that are useful in a preemptive system to prevent a task from being preempted for some critical operation.

Note: FreeRTOS API functions must not be called in a critical section.

- `taskENTER_CRITICAL/taskEXIT_CRITICAL` – Defines a block of code that must be performed atomically. Must not be called from an ISR.
- `taskENTER_CRITICAL_FROM_ISR/taskEXIT_CRITICAL_FROM_ISR` – Defines a block of code that must be performed atomically. Must only be called from an ISR.

In general, you should stay in the same scope for the enter critical section/exit critical section. In some cases, you may want to call other APIs and if those APIs do not rely on interrupts you're fine. That said, like interrupts, critical sections should be kept short whenever possible.

2.3.5 Interrupts

Interrupts will do just as you would expect; stop the current task so that they can be serviced. In fact, this is how the preemptive/time slice systems are implemented, with a system tick interrupt at the lowest interrupt priority. Here is an ISR from a CapSense code example:

```
static void capsense_timer_callback(TimerHandle_t xTimer)
{
    capsense_command_t cmd = CAPSENSE_SCAN;
    BaseType_t xYieldRequired;

    (void)xTimer;

    /* Send command to start CapSense scan */
    xQueueSendFromISR(capsense_cmd_q, &cmd, &xYieldRequired);

   /* Yield current task if a higher priority task is now unblocked */
   portYIELD_FROM_ISR(xYieldRequired);
}
```

You may be wondering what `portYIELD_FROM_ISR` does. Well let's imagine task 1 with a priority of 2 is executing when the ISR comes in. Provided interrupts are enabled and task 1 is not in a critical section, the system will switch to the ISR and run it.

Notice that the ISR calls the function `xQueueSendFromISR` that may unblock a task. Also notice the function has the "FromISR" suffix so that it is safe to call inside the ISR. If this unblocks a task with a higher priority than the one which was executing when the ISR fired, `xYieldRequired` will be set to true. Continuing the example, assume `xQueueSendFromISR` unblocks task 2 and it has a priority of 3, so `xYieldRequired` is true.

When `portYIELD_FROM_ISR` is called with a value of true, this queues up a scheduler change as the next action once the ISR is complete. From the user's point of view, the system processing is task 1 → ISR → task 2. Once task 2 is done (or blocked again), the scheduler will go back to task 1.

If `portYIELD_FROM_ISR` is false, the task will be executed once the ISR exits since nothing else of a higher priority asked for a turn. From the user's perspective it looks like task 1 → ISR → task 1. Failing to call `portYIELD_FROM_ISR` has the same behavior as calling it with false.

With our example if you failed to call `portYIELD_FROM_ISR`, the behavior the user will observe is task 1 → ISR → task 1 → task 2. A scheduler event like an ISR or system tick is required before task 1 is suspended, so this would delay when the higher priority task (task 2 in this case) gets to run.

2.3.6 Tasks

Documentation on task creation can be found at: https://www.freertos.org/a00019.html

To create a task, you use the xTaskCreate API. Its protype is:

```
BaseType_t xTaskCreate( TaskFunction_t pxTaskCode,
                        const char * const pcName,
                        const configSTACK_DEPTH_TYPE usStackDepth,
                        void * const pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t * const pxCreatedTask )
```

Of particular interest to us is the priority. Zero is the lowest and generally reserved for the idle task which may or may not feed a watchdog timer (WDT). In a preemptive system there are (generally) two basic rules at play:

- Task with highest priority will run first, and never give up the CPU until the task is blocked
- If 2 or more tasks with the same priority do not give up the CPU (they don't block), then FreeRTOS will share the CPU between them. *FreeRTOSConfig.h* determines how this sharing happens.

You can make a task give up the CPU (that is, block it) by calling:

vTaskDelay	This simply blocks the task so that something else can run; you decide how long you want it to block (in ms).
xQueueSend	If the Queue you are sending to is full, this task will be blocked until there is space in the queue.
xQueueReceive	If the Queue you are reading from is empty, this task will be blocked until there is data in the queue.
xSemaphoreTake	The task will be blocked if the semaphore has not been given.

There are other ways for tasks to become blocked as well, but these are the most common.

2.3.7 Queues

Documentation on queues can be found at https://www.freertos.org/a00018.html

You can communicate between tasks by using Queues or Semaphores. In order to use the queue functions, you must include *queue.h* in your source code. Let's create an example to communicate between two tasks. Here's some sample code:

```
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "stdio.h"

// Global Queue Handle
static QueueHandle_t qh = 0;

void task_tx(void *p)
{
    int myInt = 0;
    while (1)
    {
        myInt++;
        if (!xQueueSend(qh, &myInt, pdMS_TO_TICKS(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, pdMS_TO_TICKS(10000)))
        {
            puts("Failed to receive item within 1000 ms");
        }
        else
        {
            printf("Received: %u\n", myInt);
        }
    }
}

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

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

    CY_ASSERT(0); // Should never get here
}
```

Notes

- In `main`, we create the Queue before creating tasks, because sending to an un-initialized Queue will crash the system.
- In `task_tx`, we send one item every second. If the queue is full the task will block for 500ms (set by the timeout value). The value is specified in ticks so the macro pdMS_TO_TICKS is required to covert to milliseconds. Once the queue is no longer full, the item will be sent. If the queue stays full for 500ms, the thread unblocks due to the timeout and 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 block the task from running. `xQueueReceive` will unblock the task as soon as an item shows up in the queue. If nothing shows up in 1000ms, the task will unblock anyway - that is set by the value of 1000 for the timeout in the `xQueueReceive` function call.
- 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`.

2.3.8 Semaphores and Mutexes

Documentation on semaphores and mutexes can be found at: https://www.freertos.org/a00113.html

In FreeRTOS, Mutex functions are grouped with Semaphores since a mutex is just a special case of a semaphore. In fact, there are functions to create a mutex, but you just use semaphore functions to lock and unlock a mutex. Let me show you with an example. Note that you need to include *semphr.h* in your source code to use semaphores (or mutexes).

Mutexes

Semaphores in FreeRTOS can be used as mutexes with a priority inversion mechanism such that a lower priority task that holds a mutex for a shared resource will prevent a higher priority task from running if that higher priority task requires the same mutex. One of the best examples 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.

In order to support the Mutex functions, `configUSE_MUTEXES` must be set to 1 in the FreeRTOSConfig.h file.

```
#define configUSE_MUTEXES      1
```

A mutex will only allow ONE task to get past an `xSemaphoreTake` operation and other tasks will be put to sleep if they reach this function at the same time. In this example, the two tasks use a semaphore as a mutex.

```
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"

static SemaphoreHandle_t spi_bus_lock=0;

void task_one()
```

```
{
    while (1)
    {
        if (xSemaphoreTake(spi_bus_lock, 1000))
        {
            // Use Guarded Resource

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

void task_two()
{
    while (1)
    {
        if (xSemaphoreTake(spi_bus_lock, 1000))
        {
            // Use Guarded Resource

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

int main(void)
{
    spi_bus_lock = xSemaphoreCreateMutex();
    xTaskCreate(task_one, (char *)"t1", 2048, 0, 1, 0);
    xTaskCreate(task_two, (char *)"t2", 2048, 0, 1, 0);
    vTaskStartScheduler();
    CY_ASSERT(0);
}
```

Note that the semaphore was created using the function `xSemaphoreCreateMutex` which takes care of setting up the semaphore with priority inversion.

Binary Semaphore

A binary semaphore can also be used like a mutex, but binary semaphores doesn't provide a 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 task will block on the corresponding take operation.

```
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"

#define LONG_TIME 0xffff

static SemaphoreHandle_t event_signal = 0;

void System_Interrupt()
{
    static BaseType_t xHigherPriorityTaskWoken;
    xSemaphoreGiveFromISR(event_signal, &xHigherPriorityTaskWoken);
}
```

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

int main(void)
{
    vSemaphoreCreateBinary(event_signal); // Create the semaphore
    xSemaphoreTake(event_signal, 0);      // Take semaphore after creating it.
    // create your task, assuming it has an interrupt at some point
    vTaskStartScheduler();
    CY_ASSERT(0);
}
```

The above code shows an example of 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.

In the example, the ISR function `System_Interrupt` just gives the semaphore (with the FromISR designation) and then exits. The actual processing happens inside the function `system_interrupt_task`, which waits for the semaphore and then does the processing. This way, other critical tasks that have a higher priority than `system_interrupt_task` can run first.

Another way to use binary semaphore is to wake up one task from another task by giving the semaphore. The semaphore will essentially act like a signal that may indicate: "Something happened, now go do the work in another task".

Again, remember that when you create a 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

As you leaned earlier, a counting semaphore keeps track of how many times it was given rather than being a binary event. With a counting semaphore, you specify the depth and the initial count when you create the semaphore. Its function prototype looks like this:

```
SemaphoreHandle_t xSemaphoreCreateCounting( UBaseType_t uxMaxCount,
                                            UBaseType_t uxInitialCount )
```

2.3.9 Task Notifications

Every FreeRTOS task has a 32-bit notification value. An RTOS task notification is an event sent directly to a task that can unblock the receiving task, and optionally update the receiving task's notification value. Unblocking an RTOS task with a direct notification is 45% faster and uses less RAM than unblocking a task with a binary semaphore.

Notifications are sent using the `xTaskNotify` and `xTaskNotifyGive` functions and received using `xTaskNotifyWait` and `ulTaskNotifyTake` functions. A task that calls one of the receiving functions will block for a specified amount of time until another task calls one of the sending functions, specifying the blocked receiving task.

Task notifications can update the receiving task's notification value in several ways:

- Set the receiving task's notification value with or without overwriting previous notification values
- Set one or more bits in the receiving task's notification value
- Increment the receiving task's notification value

If you want to use a notification as a binary semaphore you should use the `xTaskNotifyGive` and the `ulTaskNotifyTake` functions. If you want to use a notification to deliver more data than a simple binary semaphore provides you should use the `xTaskNotify` and the `xTaskNotifyWait` functions.

Note The functions mentioned thus-far for sending task notifications are not ISR safe! If you want to send a notification from an ISR you will need to call the ISR safe equivalent of the function you want: `xTaskNotifyFromISR` or `vTaskNotifyGiveFromISR`

There is a complete description of, as well as a plethora of usage examples for notifications, available from FreeRTOS's website.

2.3.10 References

The information above comes for a variety of sources: Some of the following sources have more recommendations/information.

- https://www.freertos.org/a00106.html
- https://github.com/cypresssemiconductorco/freertos
- http://socialledge.com/sjsu/index.php/FreeRTOS_Tutorial