

# Chapter 3: Using the WICED Real Time Operating System (RTOS)

After completing chapter 3 you will have a fundamental understanding of the role of the WICED RTOS in building WICED applications. You will be able to use the WICED RTOS abstraction layer to create and use threads, semaphores, mutexes, queues, and timers.

<b>3.1</b>	<b>AN INTRODUCTION TO RTOS .....</b>	<b>2</b>
<b>3.2</b>	<b>WICED RTOS ABSTRACTION LAYER .....</b>	<b>3</b>
<b>3.3</b>	<b>PROBLEMS WITH RTOS .....</b>	<b>4</b>
<b>3.4</b>	<b>THREADS .....</b>	<b>5</b>
<b>3.5</b>	<b>SEMAPHORE.....</b>	<b>7</b>
<b>3.6</b>	<b>MUTEX.....</b>	<b>8</b>
<b>3.7</b>	<b>QUEUE .....</b>	<b>9</b>
<b>3.8</b>	<b>EXERCISE(S) .....</b>	<b>10</b>
	EXERCISE - 3.1 SEMAPHORE .....	10
	EXERCISE - 3.2 (ADVANCED) MUTEX .....	11
	EXERCISE - 3.3 (ADVANCED) QUEUES .....	12

## 3.1 An Introduction to RTOS

The purpose of an RTOS is to reduce the complexity of writing embedded firmware with 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 a connected network, reading and writing data to an external file system, and reading and writing data from 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 (also called **threads**) and develop them in a somewhat independent fashion.

The RTOS maintains a list of threads that are idle, halted or running and which one 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 has the ability to 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. Preemptive multitasking is the scheme that is used in Windows, Linux etc.

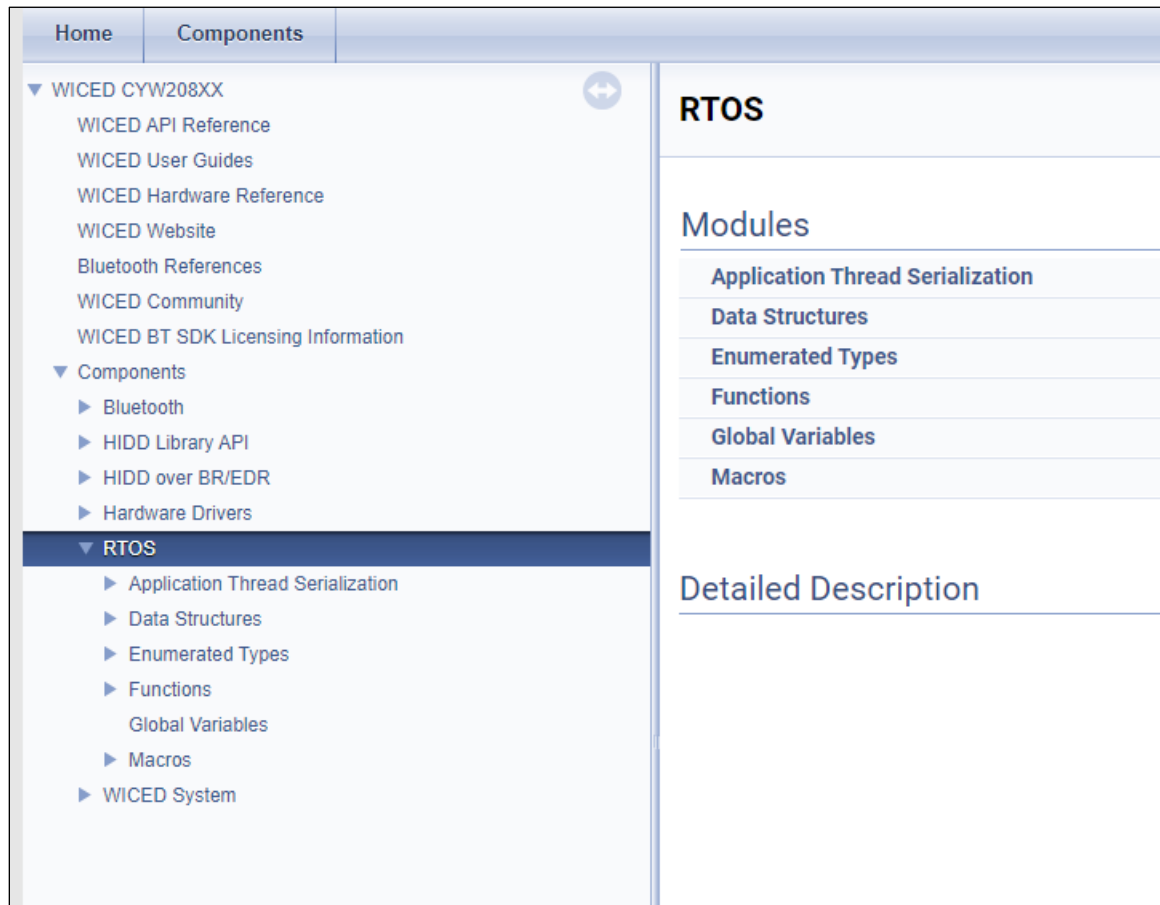
In co-operative multitasking each process has to be a good citizen and yield control back to the RTOS. There are a number of mechanisms for yielding control such as `rtos_delay`, semaphores, mutexes, and queues (which we will discuss later in this document).

The WICED RTOS is preemptive. However, higher priority tasks will always run at the expense of lower priority tasks, so it is still 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. As an example, the watch dog timer thread is very low priority (it runs in the idle task) so if your tasks don't yield you will likely see watchdog resets. It is good practice to have some form of yield control mechanism in every thread to prevent such situations.

## 3.2 WICED RTOS Abstraction Layer

Embedded WICED Bluetooth Devices have the [ThreadX](#) RTOS built into the device ROM and the license is included for anyone using WICED chips.

The WICED SDK has a built-in abstraction layer that provides a higher-level interface to the fundamental RTOS functions. You can find the documentation for the WICED RTOS APIs under the **API Guide > Components > RTOS**.



### 3.3 Problems with RTOSes

There are three serious bugs that can easily be created when using an RTOS 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.

But all hope is not lost. All RTOSes give you mechanisms to deal with these problems, specifically semaphores, mutexes, and queues. These functions generally all work the same way. The basic process is:

1. Include the *wiced\_rtos.h* header file so that you have access to the RTOS functions.
2. Declare a pointer of the right type (e.g. *wiced\_mutex\_t\**)
3. Call the appropriate create function to allocate memory and return the pointer.
4. Call the appropriate RTOS initialize function (e.g. *wiced\_rtos\_init\_mutex*). Provide it with the pointer that was created in step 2.
5. Access the pointer using one of the access functions (e.g. *wiced\_rtos\_lock\_mutex*).
6. If you don't need it anymore, free up the pointer with the appropriate de-init function (e.g. *wiced\_rtos\_deinit\_mutex*).

All these functions need to have access to the pointer, so I generally declare these "shared" resources as static global variables within the file that they are used.

## 3.4 Threads

As we discussed earlier, threads are at the heart of an RTOS. It is easy to create a new thread by calling the function `wiced_rtos_create_thread` and then `wiced_rtos_init_thread` with the following arguments:

- `wiced_thread_t* thread` – A pointer to a thread handle data structure returned by the `wiced_rtos_create_thread` function. This handle is used to identify the thread for other thread functions.
- `uint8_t priority` – This is the priority of the thread.
  - Priorities can be from 0 to 7 where 7 is the highest priority. User applications should typically use middle priorities of ~4.
  - If the scheduler knows that two threads are eligible to run, it will run the thread with the higher priority.
- `char *name` – A name for the thread. This name is only used by the debugger. You can give it any name or just use `NULL` if you don't want a specific name.
- `wiced_thread_function_t *thread` – A function pointer to the function that is the thread.
- `uint32_t stack_size` – How many bytes should be in the thread's stack.
  - You should be careful here as running out of stack can cause erratic, difficult to debug behavior. Using 1024 is overkill but will work for any of the exercises we do in this class. If you want to see how much a given thread uses, we'll show you how you can do that below.
- `void *arg` – A generic argument which will be passed to the thread.
  - If you don't need to pass an argument to the thread, just use `NULL`.

As an example, if you want to create a thread that runs the function `mySpecialThread`, the initialization might look something like this:

```
#define THREAD_PRIORITY    (4)
#define THREAD_STACK_SIZE (1024)
.
.
wiced_thread_t* mySpecialThreadHandle; /* Typically defined as a global */
.
.
/* Typically inside the BTM_ENABLED_EVT */
mySpecialThreadHandle = wiced_rtos_create_thread();
wiced_rtos_init_thread(mySpecialThreadHandle, THREAD_PRIORITY,
"mySpecialThreadName", mySpecialThread, THREAD_STACK_SIZE, NULL);
```

The thread function must match type `wiced_thread_function_t`. It must take a single argument of type `uint32_t` and must have a `void` return.

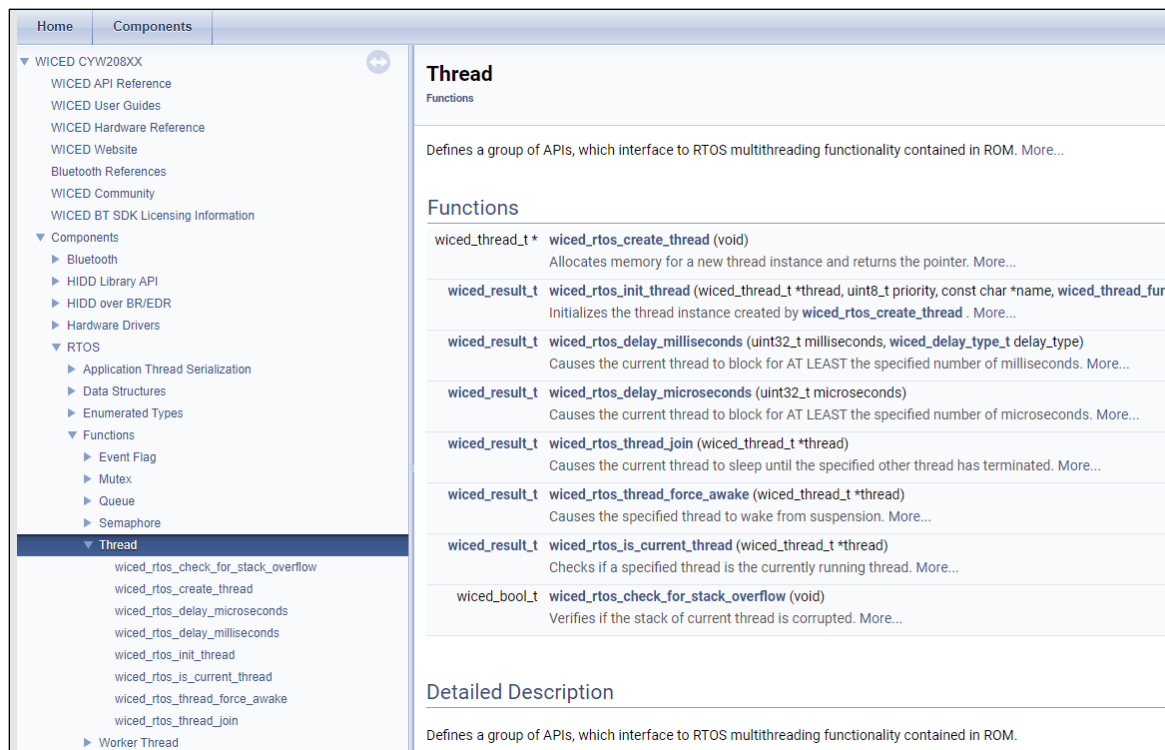
The body of a thread looks just like the `main` function of a typical C application. Often a thread will run forever so it will have an initialization section and a `while(1)` loop that repeats forever. For example:

```
void mySpecialThread(uint32_t arg)
{
    /* Do any required one-time initialization here */
    #define MY_THREAD_DELAY (100)

    while(1)
    {
        processData();
        wiced_rtos_delay_milliseconds(MY_THREAD_DELAY, ALLOW_THREAD_TO_SLEEP);
    }
}
```

**Note:** You should usually put a `wiced_rtos_delay_milliseconds` of some amount in every thread with the delay type of `ALLOW_THREAD_TO_SLEEP` so that other threads get a chance to run. The exception is if you have some other thread control function such as a semaphore or queue that is guaranteed to cause the thread to periodically pause.

The functions available to manipulate a thread are in the **Component > RTOS > Functions > Thread** section of the WICED API reference.



The screenshot shows the WICED API reference interface. On the left is a navigation pane with a tree view containing sections like 'WICED CYW208XX', 'WICED API Reference', 'Components', 'RTOS', and 'Functions'. The 'Thread' function category is selected and expanded, showing a list of functions including `wiced_rtos_check_for_stack_overflow`, `wiced_rtos_create_thread`, `wiced_rtos_delay_microseconds`, `wiced_rtos_delay_milliseconds`, `wiced_rtos_init_thread`, `wiced_rtos_is_current_thread`, `wiced_rtos_thread_force_awake`, `wiced_rtos_thread_join`, and 'Worker Thread'. The main content area on the right is titled 'Thread' and 'Functions'. It contains a description: 'Defines a group of APIs, which interface to RTOS multithreading functionality contained in ROM. More...'. Below this is a list of functions with their signatures and brief descriptions, each with a 'More...' link. The functions listed are:

- `wiced_thread_t* wiced_rtos_create_thread (void)`: Allocates memory for a new thread instance and returns the pointer. More...
- `wiced_result_t wiced_rtos_init_thread (wiced_thread_t *thread, uint8_t priority, const char *name, wiced_thread_fun_t thread_func)`: Initializes the thread instance created by `wiced_rtos_create_thread`. More...
- `wiced_result_t wiced_rtos_delay_milliseconds (uint32_t milliseconds, wiced_delay_type_t delay_type)`: Causes the current thread to block for AT LEAST the specified number of milliseconds. More...
- `wiced_result_t wiced_rtos_delay_microseconds (uint32_t microseconds)`: Causes the current thread to block for AT LEAST the specified number of microseconds. More...
- `wiced_result_t wiced_rtos_thread_join (wiced_thread_t *thread)`: Causes the current thread to sleep until the specified other thread has terminated. More...
- `wiced_result_t wiced_rtos_thread_force_awake (wiced_thread_t *thread)`: Causes the specified thread to wake from suspension. More...
- `wiced_result_t wiced_rtos_is_current_thread (wiced_thread_t *thread)`: Checks if a specified thread is the currently running thread. More...
- `wiced_bool_t wiced_rtos_check_for_stack_overflow (void)`: Verifies if the stack of current thread is corrupted. More...

At the bottom, there is a 'Detailed Description' section with the same text: 'Defines a group of APIs, which interface to RTOS multithreading functionality contained in ROM.'

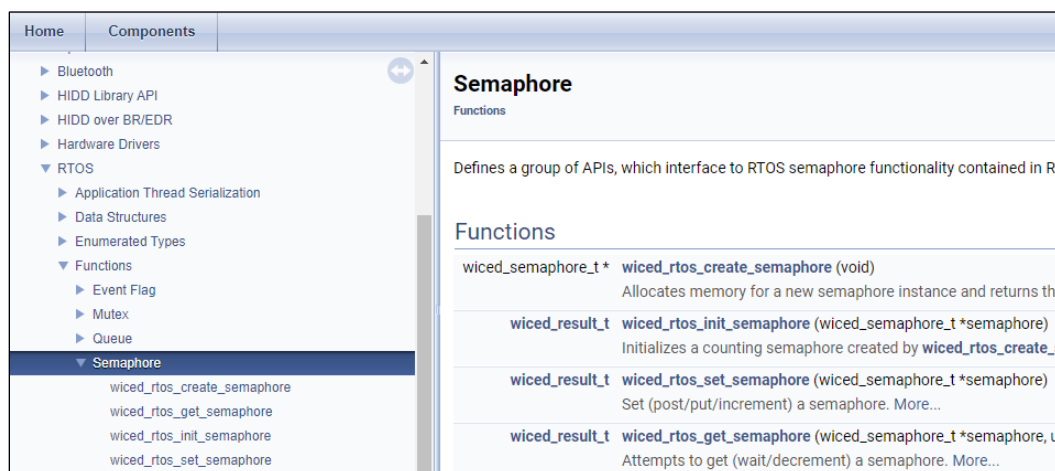
## 3.5 Semaphore

You will use this in [Exercise - 3.1](#).

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. In the BT\_20819A1 SDK, semaphores are implemented as a simple unsigned integer. When you "set" a semaphore it increments the value of the semaphore. When you "get" 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 `sendData` thread and a `collectDataThread`. The `sendData` thread will "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.

The get function requires a timeout parameter. This allows the thread to continue after a specified amount of time even if the semaphore doesn't get set. This can be useful in some cases to prevent a thread from stalling permanently if the semaphore is never set due to an error condition. The timeout is specified in milliseconds. If you want the thread to wait indefinitely for the semaphore to be set rather than timing out after a specific delay, use `WICED_WAIT_FOREVER` for the timeout.

The semaphore functions are available in the documentation under **Components > RTOS > Functions > Semaphore**.



You should always create and initialize a semaphore before starting any threads that use it. Otherwise, you may see unpredictable behavior. For example:

```
wiced_semaphore_t* mySemaphore; /* Typically defined as a global */
.
.
mySemaphore = wiced_rtos_create_semaphore(); /* Typically inside the
BTM_ENABLED_EVT */
wiced_rtos_init_semaphore( mySemaphore );
```

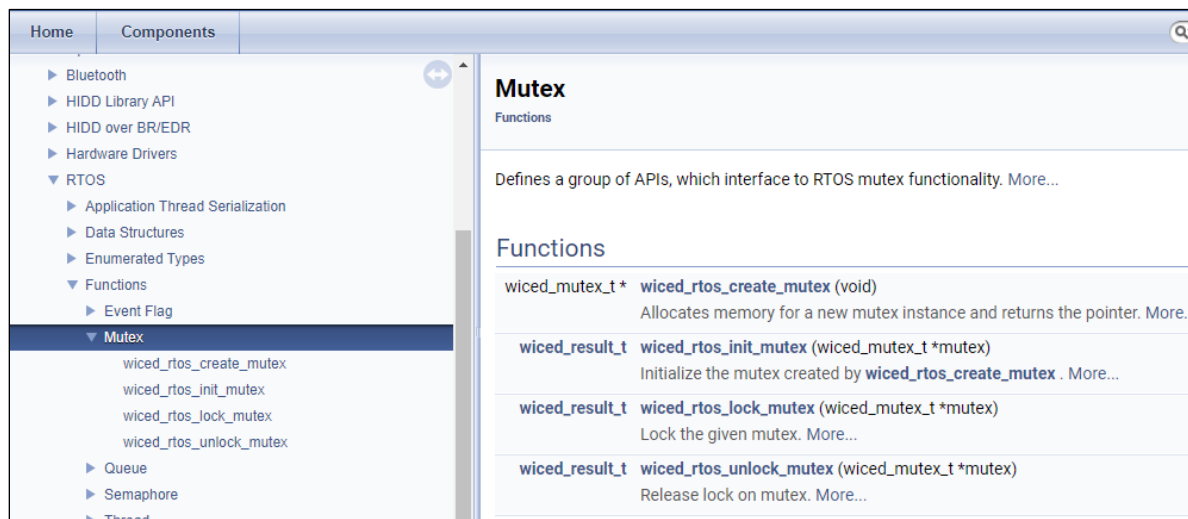
It is generally not a good idea to use a semaphore get inside an interrupt callback or a timer callback with a non-zero timeout since it may lock up your program waiting for a set that never occurs.

## 3.6 Mutex

You will use this in [Exercise - 3.2](#).

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. In the exercises for this chapter you will create two different threads that blink the same LED at different rates. Without a mutex, you will see strange behavior. With a mutex, the threads are each given exclusive access to the LED.

The mutex functions are available in the documentation under **Components > RTOS > Functions > Mutex**.



You should always create and initialize a mutex before starting any threads that use it. Otherwise, you may see unpredictable behavior. For example:

```
wiced_mutex_t* myMutex; /* Typically defined as a global */  
.  
.  
myMutex = wiced_rtos_create_mutex(); /* Typically inside the BTM_ENABLED_EVT */  
wiced_rtos_init_mutex( myMutex );
```

Note that a mutex can only be unlocked by the same thread that locked it.



## 3.7 Queue

You will use this in [Exercise - 3.3](#).

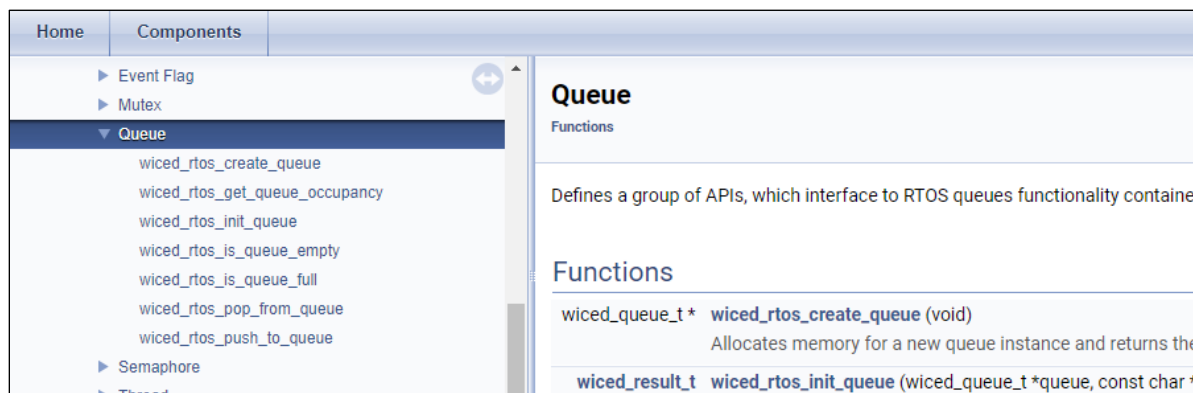
A queue is a thread-safe mechanism to send data to another thread. The queue is a FIFO - you read from the front and you write to the back. If you try to read a queue that is empty your thread will suspend until something is written into it. The payload in a queue (size of each entry) and the size of the queue (number of entries) is user configurable at queue creation time.

The `wiced_rtos_push_to_queue` requires a timeout parameter. This comes into play if the queue is full when you try to push into it. The timeout allows the thread to continue after a specified amount of time even if the queue stays full. This can be useful in some cases to prevent a thread from stalling permanently if the queue stays full due to an error condition. The timeout is specified in milliseconds. If you want the thread to wait indefinitely for room in the queue rather than timing out after a specific delay, use `WICED_WAIT_FOREVER` for the timeout. If you want the thread to continue immediately if there isn't room in the queue, then use `WICED_NO_WAIT`. Note that if the function times out, then the value is not added to the queue.

Likewise, the `wiced_rtos_pop_from_queue` function requires a timeout parameter to specify how long the thread should wait if the queue is empty. If you want the thread to wait indefinitely for a value in the queue rather than continuing execution after a specific delay then use `WICED_WAIT_FOREVER`. If you want the application to continue immediately if there isn't anything in the queue then use `WICED_NO_WAIT`.

There are also functions to check to see if the queue is full or empty and to determine the number of entries in the queue.

The queue functions are available in the documentation under **Components > RTOS > Functions > Queues**.



You should always create and initialize a queue before starting any threads that use it. Otherwise, you may see unpredictable behavior. For example:

```
wiced_queue_t* myQueue; /* Typically defined as a global */
.
.
myQueue = wiced_rtos_create_queue(); /* Typically inside the BTM_ENABLED_EVT */
wiced_rtos_init_queue( myQueue, "myQueue", sizeof(uint32_t), 5 );
```

## 3.8 Exercise(s)

### Exercise - 3.1 Semaphore

Create a program where an interrupt looks for a button press then sets a semaphore to communicate to the toggle LED thread. This material is covered in [3.5](#)

☐  
☐

1. Create a new application called **ch03\_ex01\_semaphore** using the ch03 template.
2. Create a new semaphore pointer as a global variable, then create and initialize the semaphore when the Bluetooth stack is enabled.

**Hint** You need both a create function call and an initialize function call.

**Hint** Be sure to create and initialize the semaphore before starting the LED thread or the interrupt (added in the next step) since they use the semaphore.

☐

3. Use the provided interrupt callback function to look for a button press and set the semaphore.

**Hint** Refer to the interrupt exercise from the peripherals chapter.

☐

4. Get the semaphore inside the LED thread so that it waits for the semaphore forever and then toggles the LED rather than blinking constantly.

**Hint** Use `WICED_WAIT_FOREVER` so that the thread will wait until the button is pressed. The definition for this can be found at the top of *wiced\_rtos.h*.

Questions to answer:

☐

1. Do you need `wiced_rtos_delay_milliseconds` in the LED thread? Why or why not?

## Exercise - 3.2 (Advanced) Mutex

An LED may behave strangely if two threads try to blink it at the same time. In this exercise we will use a mutex to lock access. This material is covered in [3.6](#)

- ☐ 1. Create a new application called **ch03\_ex02\_mutex** using the ch03\_ex02\_mutex template.
- ☐ 2. The application has 2 threads – one thread blinks the LED at a rate of 2 Hz and the other thread blinks the same LED at a rate of 5 Hz when the button is being pressed.
- ☐ 3. Program the application as it is to your kit. What happens when you hold down the button? Does the LED blink at 5 Hz?
- ☐ 4. Look at the TODO comments to add a mutex to the two threads so that each thread prevents the other from blinking the LED when it needs access.
- ☐ 5. Program the application to your kit. Now what happens when you hold down the button?

Questions to answer:

- ☐ 1. Before you added the mutex, how did the LED behave when you pressed the button?
- ☐ 2. What changed when you added the mutex?
- ☐ 3. What happens if you forget to unlock the mutex in one of the threads? Why?

### Exercise - 3.3 (Advanced) Queues

Use a queue to send a message to indicate the number of times to blink an LED. This material is covered in [3.7](#)



1. Create a new application called **ch03\_ex03\_queue** using the ch03 template.
2. The queue API uses memory from the buffer pools that are defined in *app\_bt\_cfg.c*.

By default, there are insufficient pools to support queues and so you need to modify the value of `wiced_bt_cfg_settings.max_number_of_buffer_pools` to allocate more memory. You should add a buffer pool for each queue that your application will create (in this case, increase from 4 to 5).



3. In *app.c*, create a global pointer to a queue and, when the stack gets enabled, create and initialize the queue

**Hint** Use a message size of 4 bytes (room for one `uint32_t`) and a queue length of 10 so you can push messages while the thread is blinking without causing the queue to overflow.



4. Add a static variable to the interrupt callback that increments each time the button is pressed.

Push the value onto the queue to give the LED thread access to it.



5. In the LED thread, pop the value from the queue to determine how many times to blink the LED.

**Hint** Add a longer delay (e.g. 1 second) after the LED blinks the specified number of times so that you can tell the button press sequences apart.



6. Program your application to the board.

Press the button a few times to see how the number of blinks is increased with each press. Note that you can press the button while it is currently blinking, and the new press will be added to the queue (provided that the queue is large enough).