

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 Mbed OS, FreeRTOS, and the WICED Bluetooth 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	CYPRESS ABSTRACTION LAYER	4
2.3	MBED OS	5
2.3.1	GENERAL.....	5
2.3.2	THREAD & THISTHREAD	5
2.3.3	QUEUE & EVENT QUEUE.....	5
2.3.4	SEMAPHORE.....	5
2.3.5	MUTEX	5
2.3.6	EVENT & EVENTFLAGS & USER ALLOCATED EVENT & CONDITIONVARIABLE.....	5
2.3.7	MEMORYPOOL	5
2.4	FREERTOS	5
2.4.1	GENERAL.....	5
2.4.2	THREAD & THISTHREAD	5
2.4.3	QUEUE & EVENT QUEUE.....	5
2.4.4	SEMAPHORE.....	5
2.4.5	MUTEX	6
2.4.6	EVENT & EVENTFLAGS & USER ALLOCATED EVENT & CONDITIONVARIABLE.....	6
2.4.7	MEMORYPOOL	6
2.5	WICED BLUETOOTH	6
2.5.1	THREAD.....	7
2.5.2	QUEUE.....	9
2.5.3	SEMAPHORE	11
2.5.4	MUTEX	12
2.5.5	STACK USAGE	13

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 a connected network, reading and writing data to an external filesystem, 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 (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 and 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 a number of 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. Therefore, 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.

But all hope is not lost. 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.

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" a counting 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 "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.

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 Cypress Abstraction Layer

In order to help make middleware applications as portable as possible within ModusToolbox, Cypress has 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 (e.g. `cy_mutex_t`)
3. Call the appropriate create or initialize function (e.g. `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 (e.g. `cy_rtos_set_mutex()`).
5. If you don't need it anymore, free up the pointer with the appropriate de-init function (e.g. `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 Mbed OS

2.3.1 General

Kernel Interface Functions

2.3.2 Thread & ThisThread

2.3.3 Queue & Event Queue

2.3.4 semaphore

2.3.5 Mutex

2.3.6 Event & EventFlags & User allocated Event & ConditionVariable

2.3.7 MemoryPool

2.4 FreeRTOS

2.4.1 General

Kernel Interface Functions

2.4.2 Thread & ThisThread

2.4.3 Queue & Event Queue

2.4.4 semaphore

2.4.5 Mutex

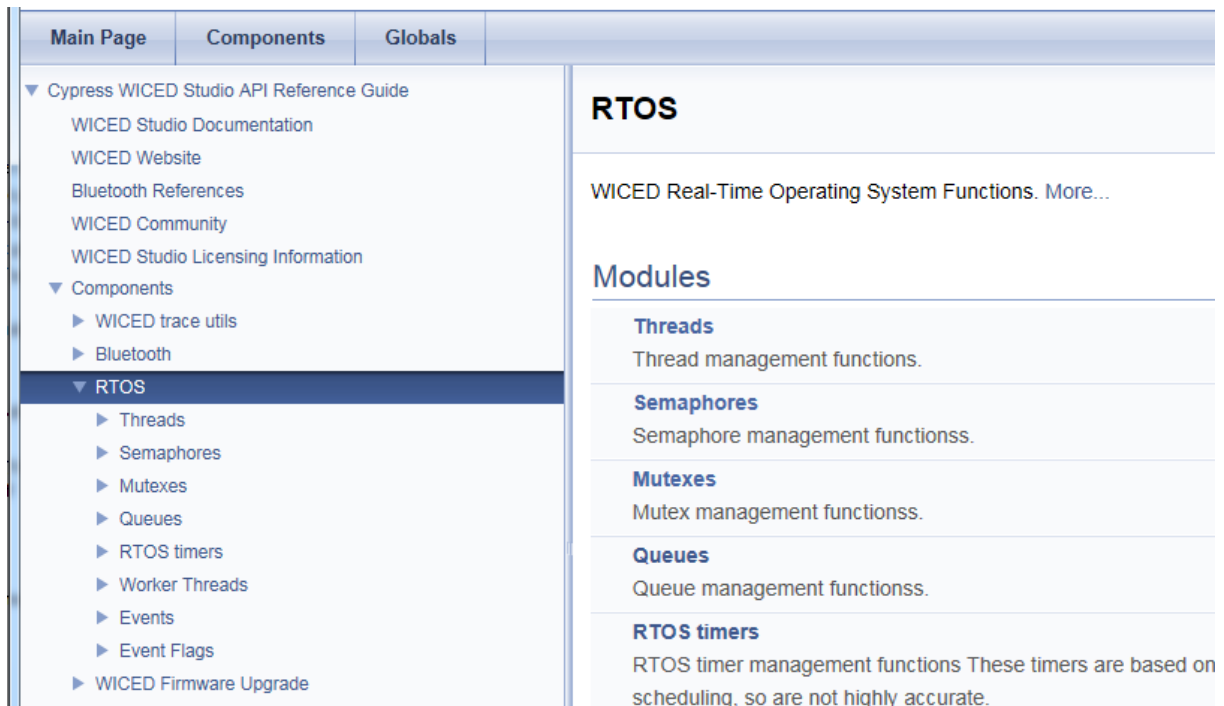
2.4.6 Event & EventFlags & User allocated Event & ConditionVariable

2.4.7 MemoryPool

2.5 WICED Bluetooth

WICED devices support multiple RTOSs, but [ThreadX](#) by [Express Logic](#) is built into the device ROM and the license is included for anyone using WICED chips so that is by far the best choice.

In order to simplify using multiple RTOSs, the WICED SDK has a built-in abstraction layer that provides a unified interface to the fundamental RTOS functions. You can find the documentation for the WICED RTOS APIs under the API Guide > Components > RTOS.



The RTOS abstraction layer 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 (e.g. `wiced_rtos_create_mutex()`).

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.

2.5.1 Thread

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.

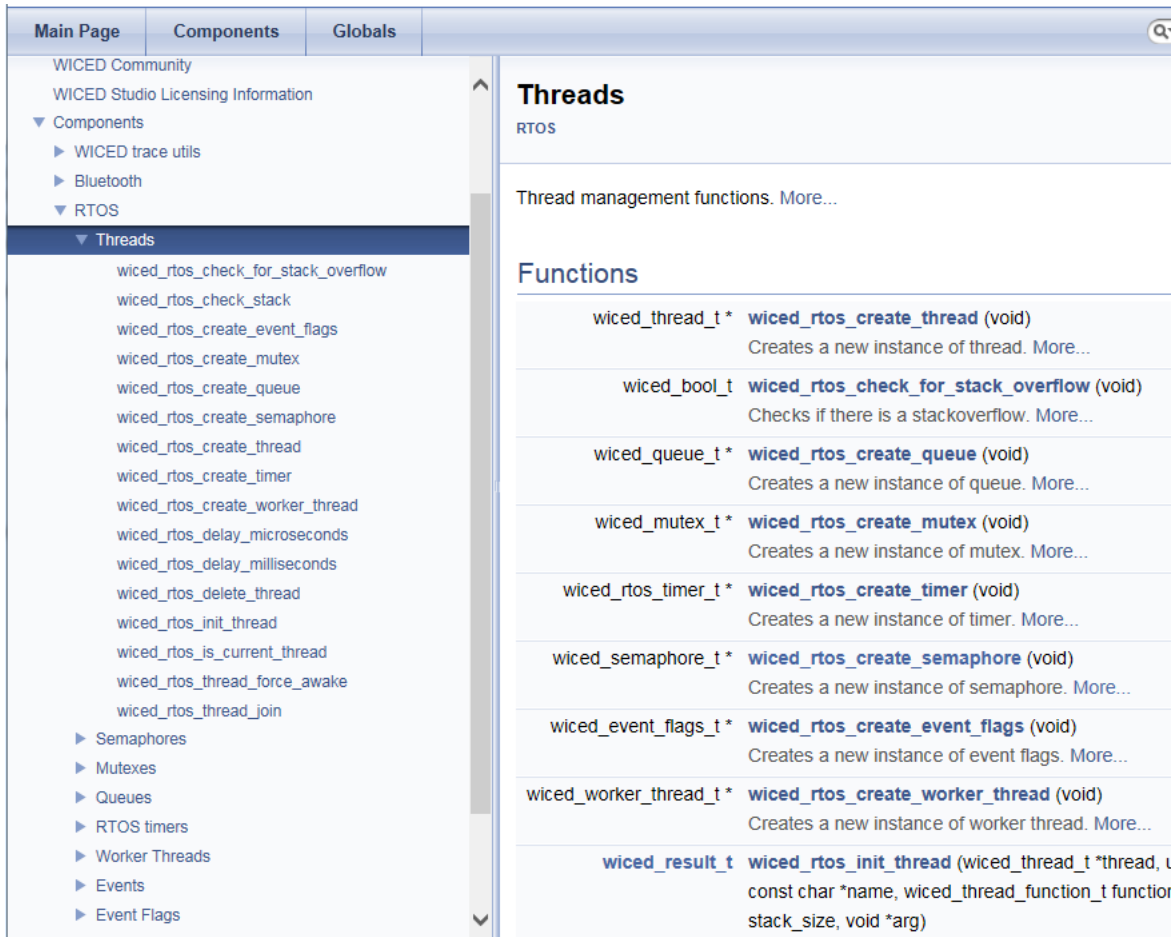
The body of a thread looks just like the "main" function of a typical C application. Often a thread will run forever so it will have an initialization section and a while(1) loop that repeats forever. For example:

```
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 > Threads" section of the WICED API reference.



The screenshot shows the WICED API reference interface. On the left, a navigation pane lists the hierarchy: Main Page, Components, Globals, and a search bar. Under 'Components', 'RTOS' is expanded, and 'Threads' is selected. The main content area is titled 'Threads' and 'RTOS'. It contains a section for 'Thread management functions' with a 'More...' link. Below this is a 'Functions' section listing various WICED RTOS thread functions, each with its return type, function name, parameters, and a brief description with a 'More...' link.

Return Type	Function Name	Parameters	Description
wiced_thread_t *	wiced_rtos_create_thread	(void)	Creates a new instance of thread. More...
wiced_bool_t	wiced_rtos_check_for_stack_overflow	(void)	Checks if there is a stackoverflow. More...
wiced_queue_t *	wiced_rtos_create_queue	(void)	Creates a new instance of queue. More...
wiced_mutex_t *	wiced_rtos_create_mutex	(void)	Creates a new instance of mutex. More...
wiced_rtos_timer_t *	wiced_rtos_create_timer	(void)	Creates a new instance of timer. More...
wiced_semaphore_t *	wiced_rtos_create_semaphore	(void)	Creates a new instance of semaphore. More...
wiced_event_flags_t *	wiced_rtos_create_event_flags	(void)	Creates a new instance of event flags. More...
wiced_worker_thread_t *	wiced_rtos_create_worker_thread	(void)	Creates a new instance of worker thread. More...
wiced_result_t	wiced_rtos_init_thread	(wiced_thread_t *thread, const char *name, wiced_thread_function_t function, stack_size, void *arg)	

2.5.2 Queue

There are two primary queue functions (besides the standard create and init) in the WICED RTOS abstraction API: *wiced_rtos_push_to_queue* and *wiced_rtos_pop_from_queue*.

The *wiced_rtos_push_to_queue* function 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 > Queues.



The screenshot shows the Cypress ModusToolbox documentation interface. The left sidebar contains a navigation menu with the following structure:

- Main Page
- Components
 - WICED Device Management
 - Application Thread Serialization
 - Bluetooth
 - RTOS
 - Macro
 - Global Variables
 - Data Structures
 - Enumerated Types
 - Functions
 - Thread
 - Semaphore
 - Mutex
 - Queue** (selected)
 - wiced_rtos_create_queue
 - wiced_rtos_get_queue_occupancy
 - wiced_rtos_init_queue
 - wiced_rtos_is_queue_empty
 - wiced_rtos_is_queue_full
 - wiced_rtos_pop_from_queue
 - wiced_rtos_push_to_queue
 - Worker Thread
 - Event Flag
 - Group_rtos_macros

The main content area is titled "Queue" and "Functions". It contains the following text:

Defines a group of APIs, which interface to RTOS queues functionality contained in ROM. [More...](#)

Functions

wiced_queue_t *	wiced_rtos_create_queue (void)	Allocates memory for a new queue instance and returns the pointer. More...
wiced_result_t	wiced_rtos_init_queue (wiced_queue_t *queue, const char *name, uint32_t max_count)	Initializes queue instance created by wiced_rtos_create_queue . More...
wiced_result_t	wiced_rtos_push_to_queue (wiced_queue_t *queue, void *message, uint32_t count)	Enqueues an element. More...
wiced_result_t	wiced_rtos_pop_from_queue (wiced_queue_t *queue, void *message, uint32_t count)	Dequeues element and memcpy's it to given pointer. More...
wiced_bool_t	wiced_rtos_is_queue_empty (wiced_queue_t *queue)	Returns boolean value indicating whether queue is empty. More...
wiced_bool_t	wiced_rtos_is_queue_full (wiced_queue_t *queue)	Returns boolean value indicating whether queue is full. More...
wiced_result_t	wiced_rtos_get_queue_occupancy (wiced_queue_t *queue, uint32_t *count)	Read number of elements enqueued. More...

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

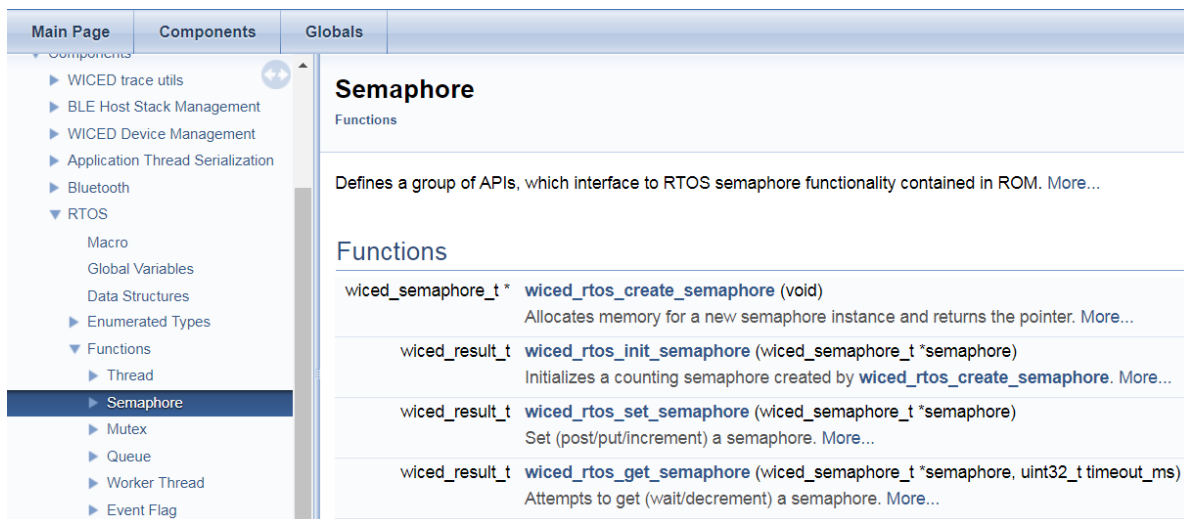
```
wiced_queue_t* myQueue; /* Typically defined as a global */
.
.
myQueue = wiced_rtos_create_queue(); /* Typically inside the
BTM_ENABLED_EVT */
wiced_rtos_init_queue( myQueue, "myQueue", sizeof(uint32_t), 5 );
```

2.5.3 Semaphore

Semaphores in the WICED RTOS abstraction have a set function called *wiced_rtos_set_semaphore* and a get function called *wiced_rtos_get_semaphore*.

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→Semaphores.



Semaphore

Functions

Defines a group of APIs, which interface to RTOS semaphore functionality contained in ROM. [More...](#)

Functions

wiced_semaphore_t *	wiced_rtos_create_semaphore (void)	Allocates memory for a new semaphore instance and returns the pointer. More...
wiced_result_t	wiced_rtos_init_semaphore (wiced_semaphore_t *semaphore)	Initializes a counting semaphore created by wiced_rtos_create_semaphore . More...
wiced_result_t	wiced_rtos_set_semaphore (wiced_semaphore_t *semaphore)	Set (post/put/increment) a semaphore. More...
wiced_result_t	wiced_rtos_get_semaphore (wiced_semaphore_t *semaphore, uint32_t timeout_ms)	Attempts to get (wait/decrement) a semaphore. More...

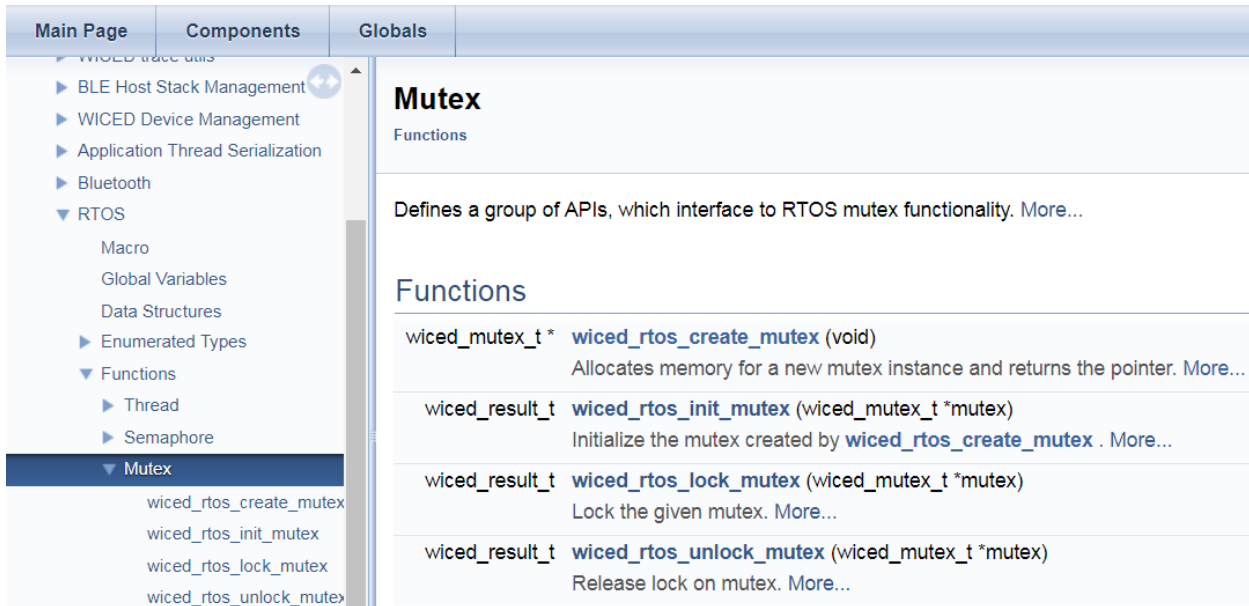
You should always create and initialize a semaphore before starting any threads that use it. Otherwise, you may see unpredictable behavior. For example:

```
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.

2.5.4 Mutex

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



Mutex

Functions

Defines a group of APIs, which interface to RTOS mutex functionality. [More...](#)

Functions

wiced_mutex_t *	wiced_rtos_create_mutex (void)	Allocates memory for a new mutex instance and returns the pointer. More...
wiced_result_t	wiced_rtos_init_mutex (wiced_mutex_t *mutex)	Initialize the mutex created by wiced_rtos_create_mutex . More...
wiced_result_t	wiced_rtos_lock_mutex (wiced_mutex_t *mutex)	Lock the given mutex. More...
wiced_result_t	wiced_rtos_unlock_mutex (wiced_mutex_t *mutex)	Release lock on mutex. More...

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.

2.5.5 Stack Usage

If you want to see how much stack a thread is using, you need to use the underlying functions for the RTOS you are using. For example, if you are using ThreadX (the default RTOS in WICED) you first need to include the header to get access to the native ThreadX functions:

```
#include "tx_api.h"
```

Next, you could add this function to find the max stack size used by a given thread since it was started:

```
/* This will return the max amount of stack a thread has used */
/* This is ThreadX specific code so it will only work for ThreadX */
uint32_t maxStackUsage(TX_THREAD *thread)
{
    uint8_t *end = thread->tx_thread_stack_end;
    uint8_t *start = thread->tx_thread_stack_start;
    while(start < end)
    {
        if(*start != 0xEF)
        {
            return end-start;
        }
        start++;
    }
    return 0;
}
```

Finally, you can call the function occasionally in your application (e.g. in a timer or whenever a button is pressed) and then print the value. For example:

```
uint32_t size;
size = maxStackUsage((TX_THREAD*) mySpecialThreadHandle);
WICED_BT_TRACE("Max Stack Size: %d\n\r", size);
```