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

## Objective

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

## Time: 2 Hours

## Fundamentals

### An introduction to RTOS

The [purpose of an RTOS](http://rtos.com/PDFs/What_Is_An_RTOS_and_Why_Use_One_Embedded.com_.pdf) 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, 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**) and develop them in a somewhat independent fashion.

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 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.

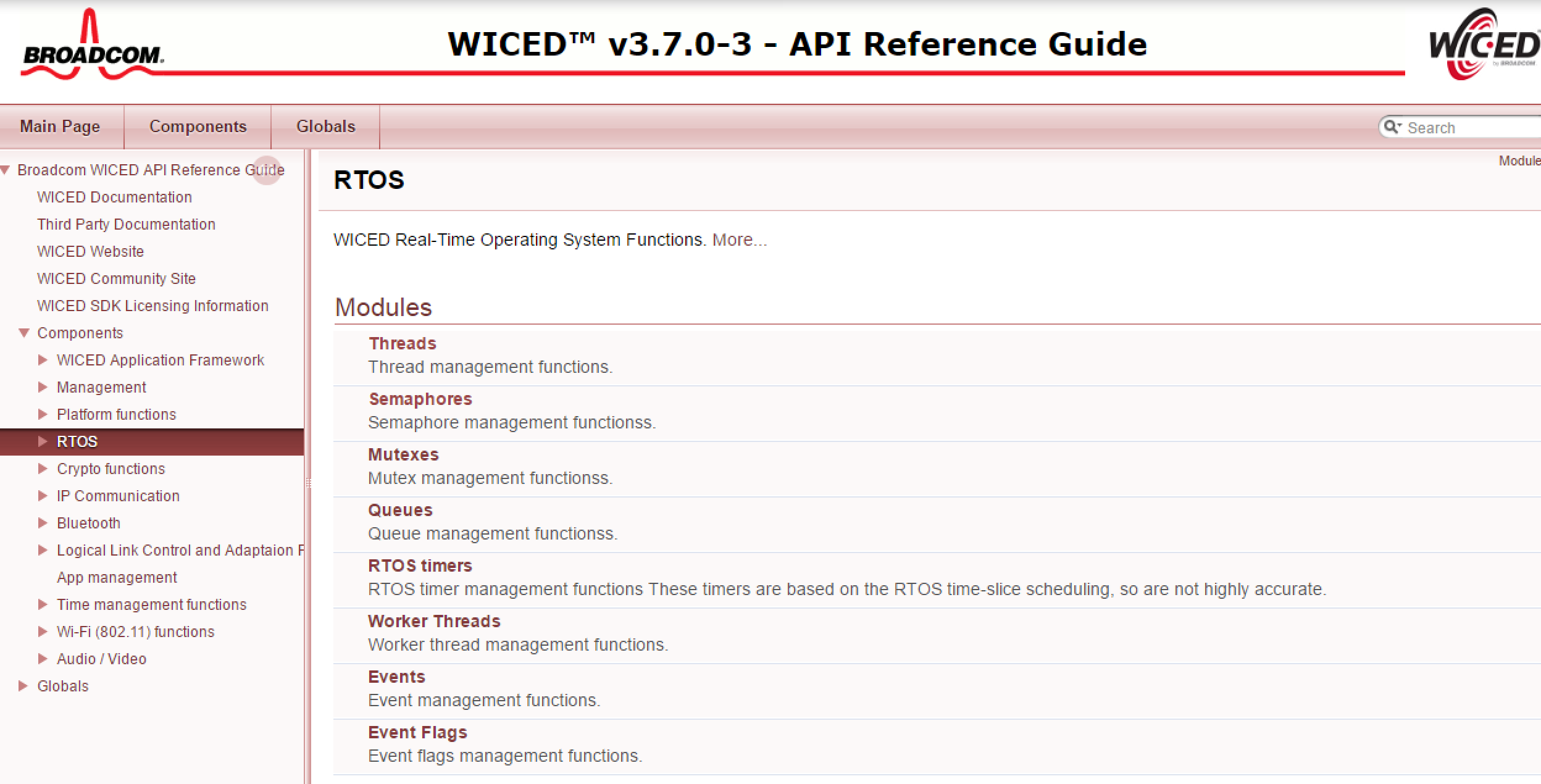
In co-operative multitasking each process has to be a good citizen and yield control back to the RTOS. There are a number of mechanisms for yielding control such as rtos\_delay, semaphores, mutexes, and queues (which we will discuss later in this document). The WICED RTOSs are all co-operative - so you need to play nice.

### WICED RTOS Abstraction Layer

Currently WICED supports multiple RTOSs including:

* [ThreadX](http://rtos.com/products/threadx/) by [Express Logic](http://rtos.com/)
* [FreeRtos](http://www.freertos.org/RTOS.html) by [FreeRTOS](http://www.freertos.org/)

However, much of ThreadX is built into the ROMs on the WICED chips so it is generally the best answer. 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.



### Problems with RTOSs

All of this sounds great, but everything is not peaches and cream (or whatever your favorite metaphor for a perfect place might be). There are three serious bugs which can easily be created in these types of systems that can be very hard to figure out. 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 with sharing memory and sharing peripherals which can cause erratic non-deterministic behavior
* Difficulties in executing inter-process communication.

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

1. Start by creating a data structure of the right type (e.g. *wiced\_mutex\_t*).
2. Call the RTOS initialize function (e.g. *wiced\_rtos\_init\_mutex*). Provide it with a pointer to the structure that was created in the first step. This is a “handle” that is used by the other functions.
3. Access the data structure using one of the access functions (e.g. *wiced\_rtos\_lock\_mutex*).
4. Kill your data structure with the appropriate de-init function (e.g. *wiced\_rtos\_deinit\_mutex*).

All of these functions need to have access to the data structure, so I generally declare these “shared” resources as static globals within the file that they are used.

### Threads

As we discussed earlier, threads are at the heart of an RTOS. It is easy to create a new thread, all you need to do is call the function *wiced\_rtos\_create\_thread* with the right arguments:

* wiced\_thread\_t\* thread – A pointer to a thread handle. 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 255 where 0 is the highest priority.
  + If the scheduler knows that two threads are eligible to run, it will run the thread with the higher priority.
  + The WICED WiFi Driver (WWD) runs at priority 3.
* char \*name – A name for the thread.
* 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).
* 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.

The thread function must match type *wiced\_thread\_function\_t*. It must take a single argument of type wiced\_thread\_arg\_t and must have a void return.

The body of a thread looks just like the infinite loop of “main”. For example:

void mySpecialThread(wiced\_thread\_arg\_t arg)

{

const int delay=100; //

while(1)

{

processData();

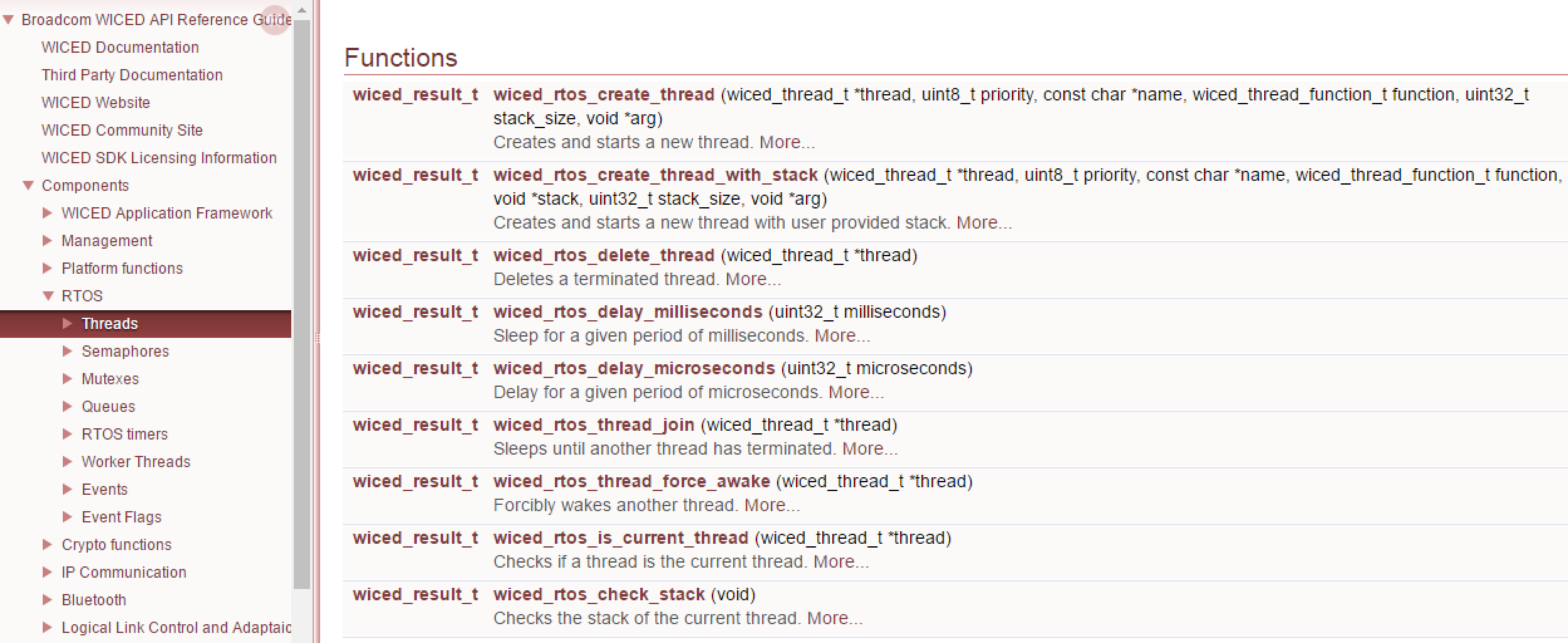
wiced\_rtos\_delay\_milliseconds(delay);

}

}

Note: you should (almost) always put a *wiced\_rtos\_delay\_milliseconds* or *wiced\_rtos\_delay\_microseconds* of some amount in every thread so that other threads get a chance to run. This applies to the main application while(1) loop as well – the main application is just another thread.

The functions available to manipulate a thread are in the “Component🡪RTOS🡪Threads” section of the API guide.

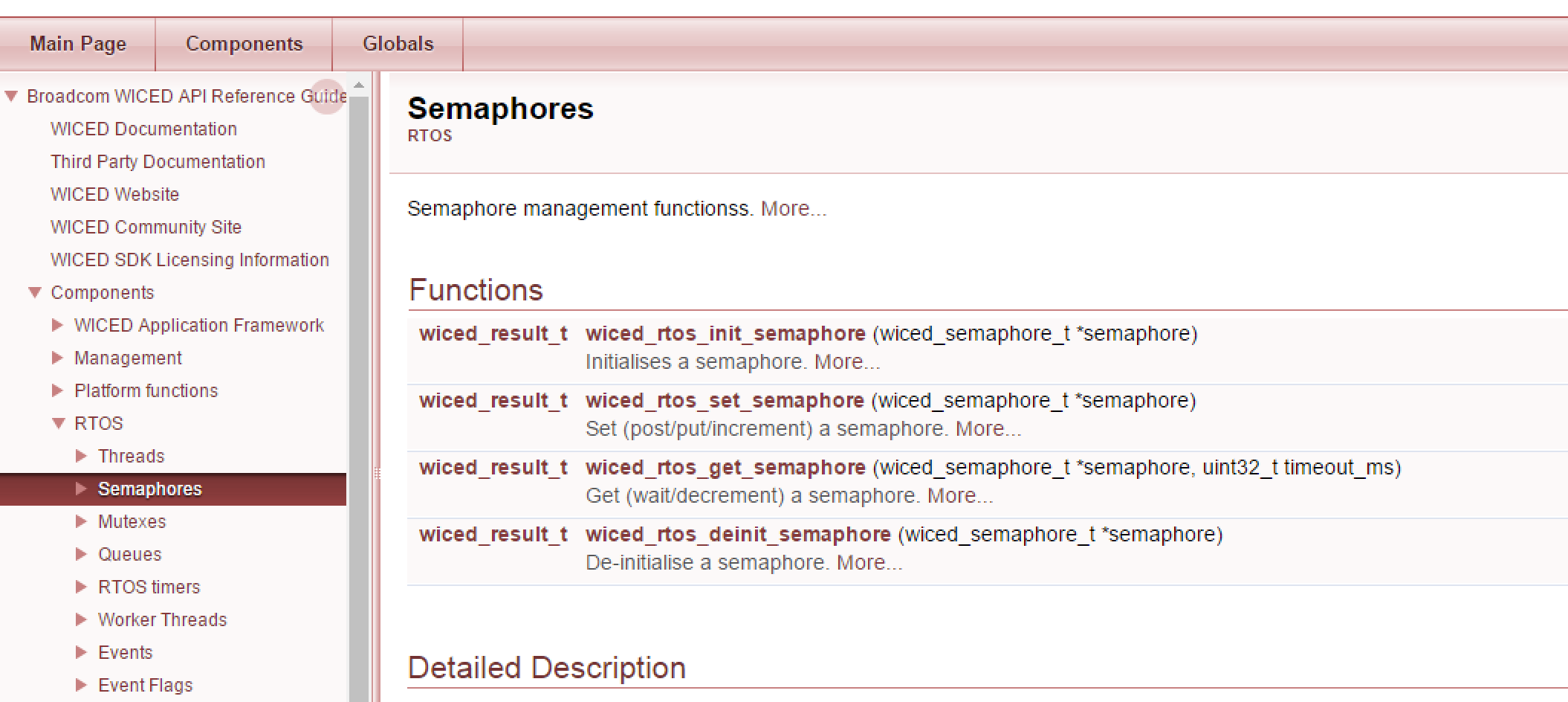


### Semaphore

A [semaphore](https://en.wikipedia.org/wiki/Semaphore_(programming)) 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 WICED 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 “sendToCloud” thread and a “collectDataThread”. The sendToCloud 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 to the cloud.

The get function requires a timeout parameter. This sets the time in milliseconds that the function waits before returning. If you want the thread to wait indefinitely for the semaphore to be set rather than continuing execution after a specific delay then use WICED\_WAIT\_FOREVER.

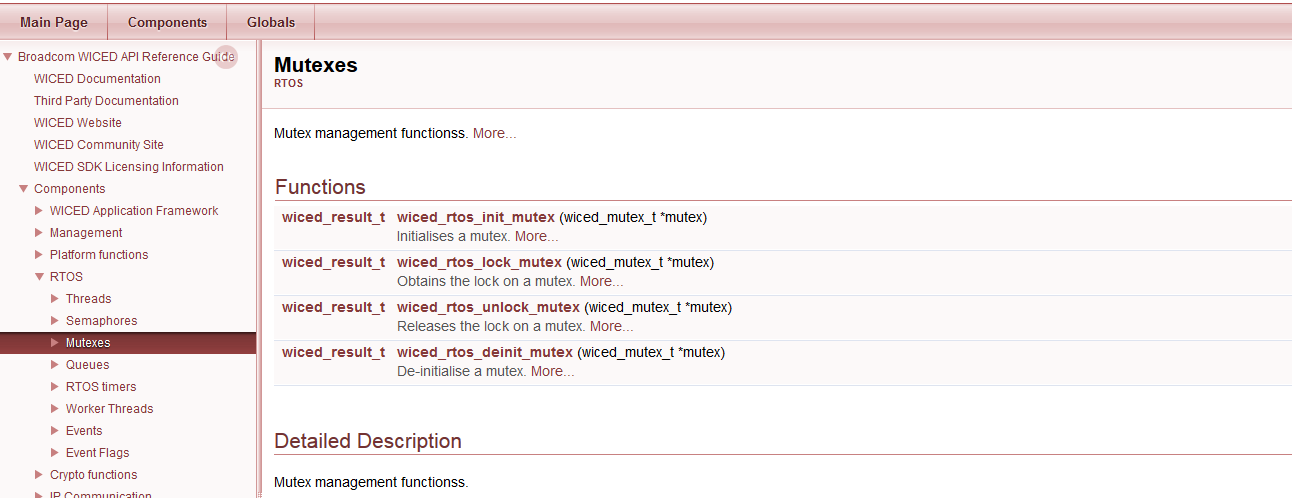
The semaphore functions are available in the documentation under Components🡪RTOS🡪Semaphores.



### 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. In the exercises for this chapter you will create a mutex for the WPRINT\_APP\_INFO function. This function takes a variable amount of time to stream the bytes out through the UART. If more than one thread uses this function to write to the UART at the same time, bad things will happen. You can protect yourself by using a mutex.

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



### 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.

The *wiced\_rtos\_queue\_push\_to\_queue* requires a timeout parameter. This sets the time in milliseconds that the function waits before returning if the queue is full. If you want the thread to wait indefinitely for space in the queue rather than continuing execution after a specific delay then use WICED\_WAIT\_FOREVER. 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.

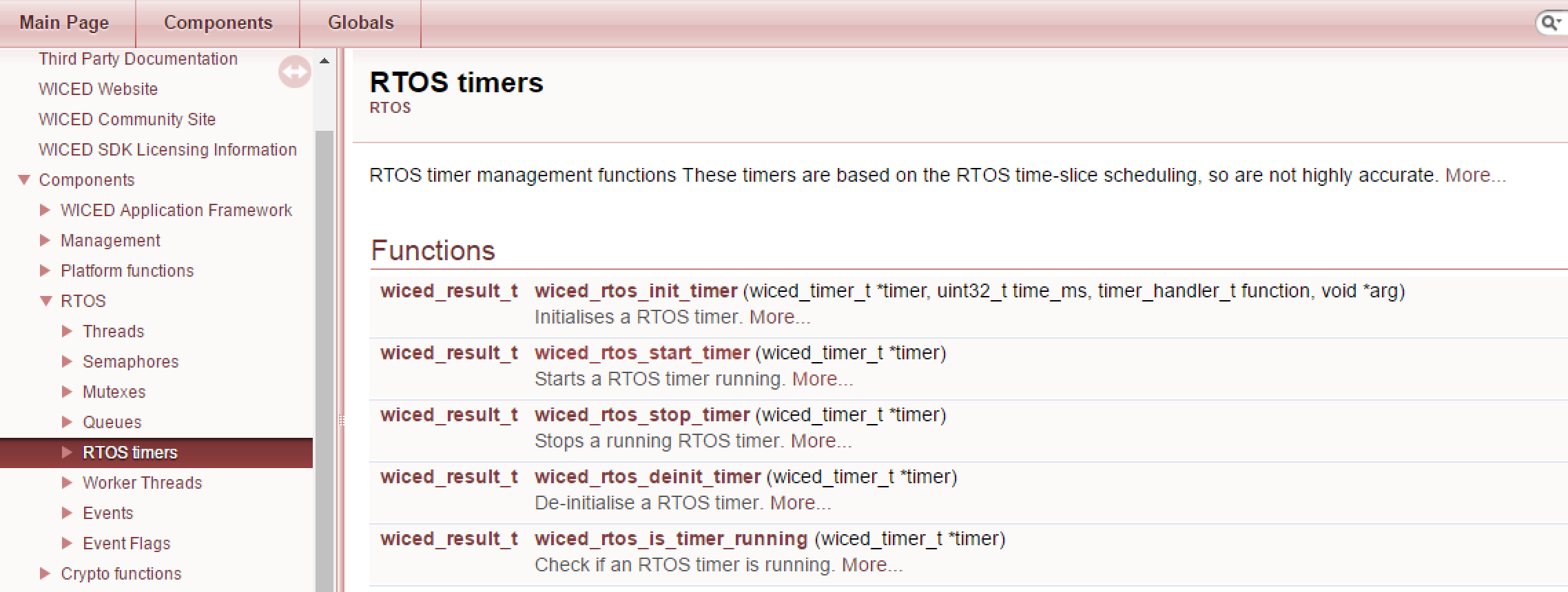
The queue functions are available in the documentation under Components🡪RTOS🡪Queues.



### Timer

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 the timer you specify the function you want run and how often you want it run. The function that the timer calls takes a single argument of *void\* arg*. If the function doesn’t require any arguments you can specify NULL in the timer initialization function.

The timer functions are available in the documentation under Components🡪RTOS🡪RTOS Timers.



### Debugging

TBD

## Exercise(s)

### 01 (THREAD) Create a blinking LED thread to blink every 500ms

1. Make a new folder under the wa101 folder called 03. Copy the 02/03\_blinkled project into the 03 folder. Rename the project to 01\_thread. Update the make file and create a make target.
2. Setup a new thread to blink the LED on/off every 500ms.
   1. Hint: Move the code from the 03\_blinkled project’s main application loop into the thread’s loop.
   2. Hint: The main application loop needs a delay such as *wiced\_rtos\_delay\_milliseconds(1)* – if you don’t include this, the LED thread won’t get a chance to run because the main application thread will never suspend.
3. Program your project to the board.

### 02 (SEMAPHORE) Create a program where one thread looks for a button press then uses a semaphore to communicate to the toggle LED thread

1. Copy 01\_thread to 02\_semaphore. Update the make file and create a make target.
2. Create a new semaphore.
3. Look for a button press in the main application thread and set the semaphore when the button is pressed.
   1. Hint: You can use a pin interrupt to detect the button press and set the semaphore.
   2. Hint: Make sure you add a delay to the main thread so that the other thread gets a chance to run.
4. Use *wiced\_rtos\_get\_semaphore* inside the LED thread so that it waits for the semaphore forever and then toggles the LED rather than blinking constantly.
   1. Hint: If the thread has “blink” in its name you should rename it to be consistent with what it now does.

Questions to answer:

Do you need *wiced\_rtos\_delay\_millisecon*ds in the LED thread? Why or why not?

What happens if you use a value of 100 for the semaphore timeout? Why?

### 03 (MUTEX) The WPRINT\_APP\_INFO will go haywire if two threads write to it at the same time. Use a mutex to lock printing.

1. Copy 01\_thread to 03\_mutex. Update the make file and create a make target.
2. Add a second thread that blinks LED2 at a rate of 498ms.
   1. Hint: Use a delay of 250ms in one thread and a delay of 249ms in the other thread.
3. Add a WPRINT\_APP\_INFO to each of the threads with different messages.
4. Program your project to the board.
5. Open a terminal and look at how the messages are printed. Do you see any issues?
6. Add a mutex to the project so that each thread can print properly.

Questions to answer:

What happens if you forget to unlock the mutex in one of the threads? Why?

Do the LEDs still blink? Why?

### 04 (QUEUES) Use a queue to send a message to indicate the number of times to blink an LED.

1. Copy 02\_semaphore to 04\_queue. Update the make file and create a make target.
2. Remove the semaphore from the project and instead create a queue.
3. Add a static variable to the ISR that increments each time the button is pressed. Push the value onto the queue to give the LED thread access to it.
4. In the LED thread, pop the value from the queue to determine how many times to blink the LED
5. Program your project to the board. Press the button a few times to see how the number of blinks is increased with each press.

### 05 (TIMERS) Make a blinking LED based on timer.

1. Copy 01\_thread to 05\_timer. Update the make file and create a make target.
2. Update the LED thread function so that it is just a simple function to toggle the LED with no while(1) loop and no *wiced\_rtos\_delay\_milliseconds*.
   1. Hint: the variable to remember the state of the LED must be static since the function will exit each time it completes rather than running infinitely like the thread.
3. Remove the thread creation function call and instead setup an RTOS timer that will call the LED function every 250ms.
4. Program your project to the board.

Questions to answer:

Can the main application loop be empty in this case? Why?

What happens if you don’t remove the while(1) loop from the function that blinks the LED? Why?

## Related Example “Apps”

|  |  |
| --- | --- |
| **App Name** | **Function** |
| snip.thraed\_monitor | Demonstrates using the system monitor API to monitor operation of an application thread. |
| snip.stack\_overflow | Demonstrates a stack overflow condition. |

## Known Errata + Enhancements + Comments

How do you know what size stack is required for a given thread?