# 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, mutex, 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 to a 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.

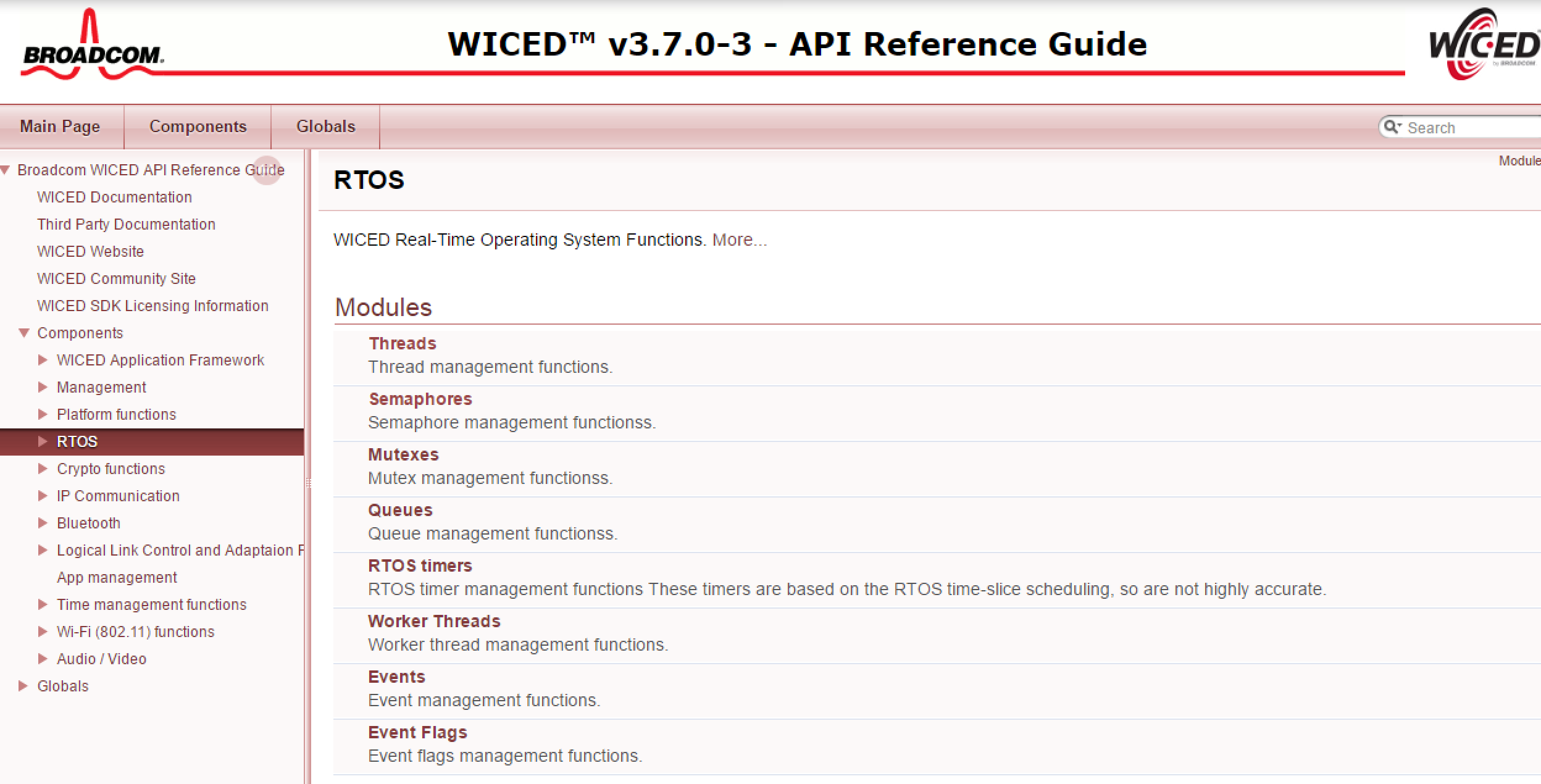
In co-operative multitasking each process has to be a good citizen and yield control back to the RTOS. There are a number of mechanisms for yielding control (which we will discuss later in this document). The WICED RTOSs are all co-operative- so you need to play nice.

### WICED RTOS Abstraction Layer

Currently WICED support 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 interprocess communication.

But all hope is not lost. The WICED RTOSs give you mechanisms to deal with these problems, specifically mutex, semaphore, queue and timers. All of these functions generally work the same way. You start by creating a data structure of the right type (e.g. wiced\_mutex\_t). Then you call the rtos initialize function (e.g. wiced\_rtos\_init\_mutex). Then you can access the datastructure using one of the access functions (e.g. wiced\_rtos\_lock\_mutex). Then you can kill your data structure with the appropriate de-init function (e.g. wiced\_rtos\_deinit\_mutex). All of these function need to have access to the datastructure, 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 – a blank thread datastructure
* uint8\_t – priority (if the scheduler knows that two threads are eligble to run, it will run the thread with the higher priority)
* char \*name – a name for the thread
* wiced\_thread\_function – a function pointer to the function that is the thread
* uint32\_t stack size – how many bytes should be in the threads 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

To make a thread function you need to declare a function that matches the wiced\_thread\_function\_t

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) {

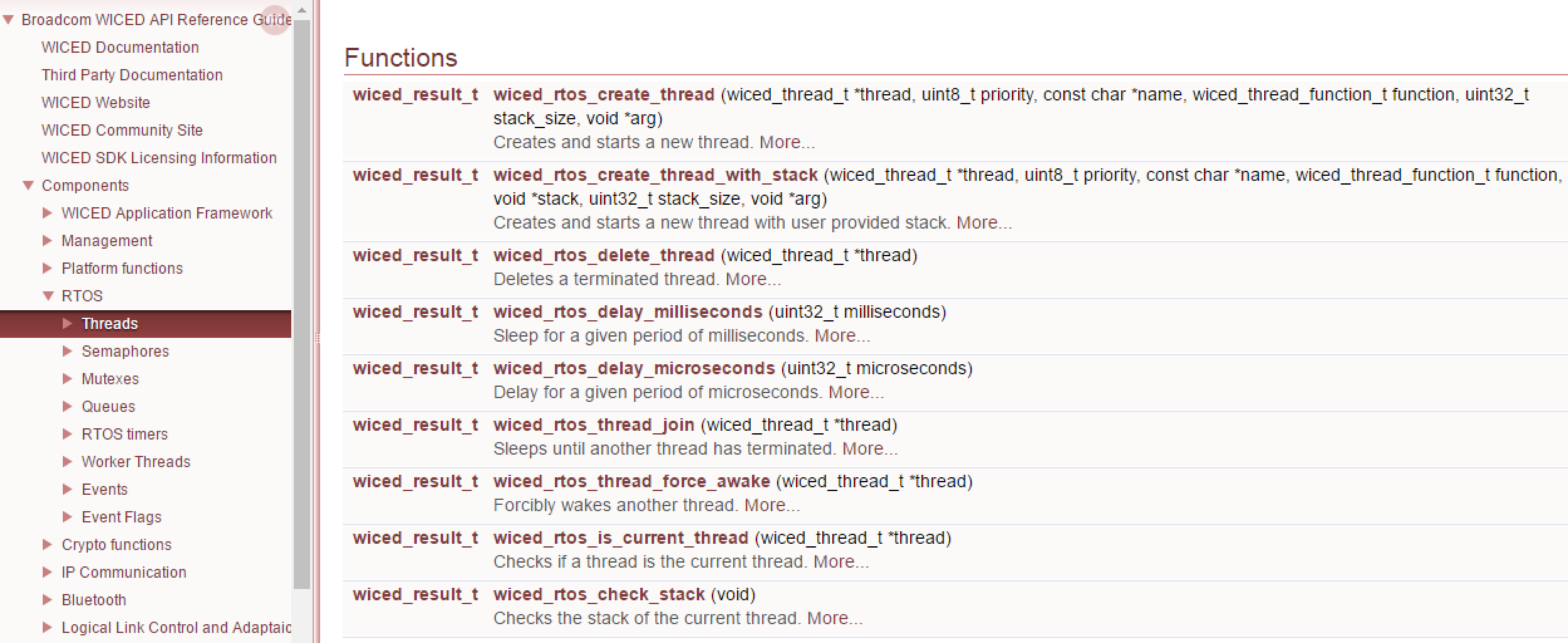
processData();

wiced\_rtos\_delay(delay);

}

}

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



### Mutex

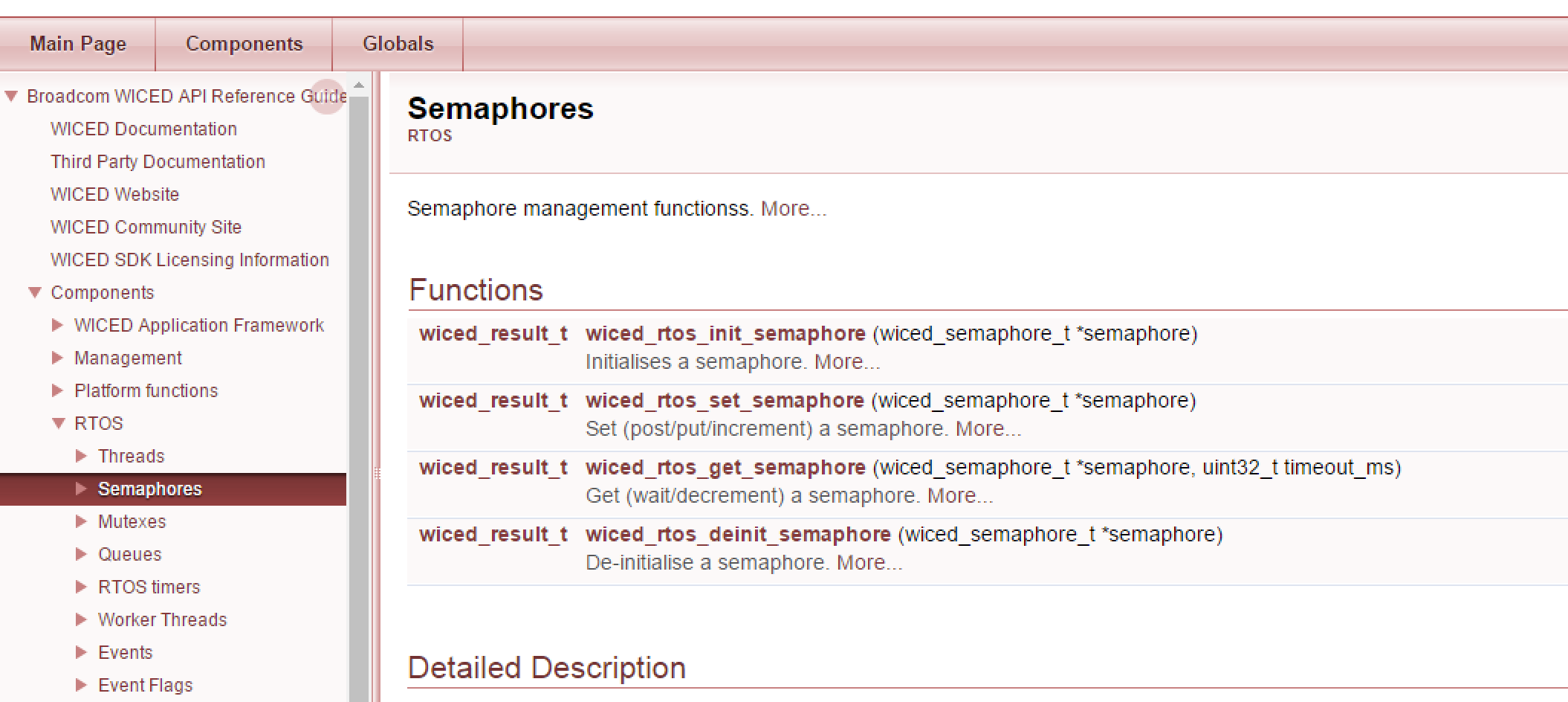
Mutex is an abbreviation for “Mutual Exclusion”. A Mutex is a lock on a specific resource, if you request a Mutex on a resource that is already locked by another thread, then your thread will go to sleep until the lock is released. In the exercises for this chapter you will create a mutex for the WPRINT\_APP\_INFO 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.



### 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 can “set” the semaphore when it has new data available that needs to be sent to the cloud.

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



### Queue

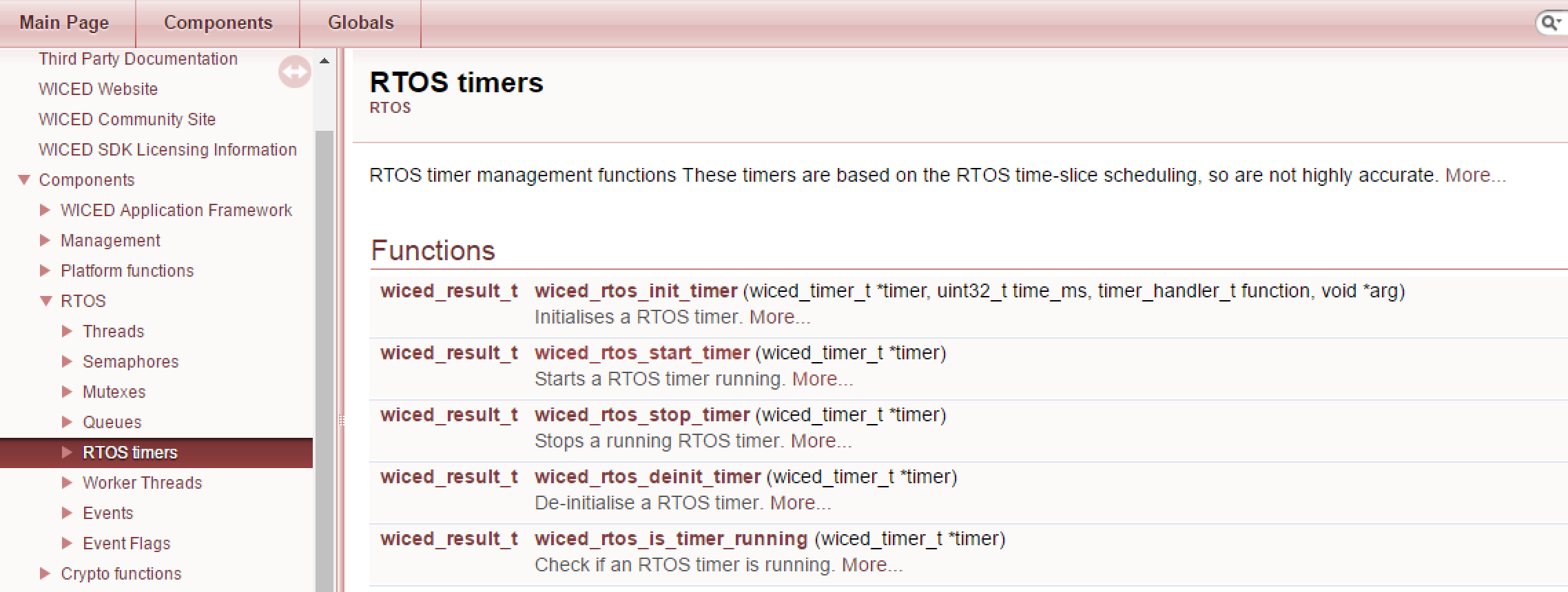
A queue is a thread safe mechanism to send data to another thread. The queue is FIFO that is you read from the front and you write to the back. If you try to read a queue that is empty your thread will suspend until something is written into it. The payload in a queue and the size of the queue is user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

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



### Timer

A 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 specific the function you want run and how often you want it run.



## Exercise(s)

### (thread) Create a blinking led thread to blink every 500ms

### (semaphore) Create a program where the main thread looks for a button press the uses a semaphore to lock/unlock a toggle led thread

### (mutex) The WPRINT\_APP\_INFO will go haywire if two thread write to it at the same time. Create a new function that uses a mutex to lock printing

### (queues) Use a queue to send a message to indicate the number of times to blink and LED

### (timers) Make a blinking led based on timer