Chapter 4: RTOS

As previously mentioned, the AIROC™ Bluetooth® SDK MCUs use an RTOS to manage the user's application along with Bluetooth® operation. After completing this chapter, you will understand general RTOS concepts and the BTSDK RTOS abstraction layer.

Table of contents

Document conventions

Convention	**Usage**	**Example**
`Courier New`	Displays code and text commands	`CY_ISR_PROTO(MyISR);` `make build`
Italics	Displays file names and paths	*sourcefile.hex*
[**bracketed, bold**]	Displays keyboard commands in procedures	[**Enter**] or [**Ctrl**] [**C**]
Menu > Selection	Represents menu paths	**File > New Project > Clone**
Bold	Displays GUI commands, menu paths and selections, and icon names in procedures	Click the **Debugger** icon, and then click **Next**.

4.1 Introduction to RTOS

4.1.1 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.

4.1.2 Tasks/threads

A task/thread is a specific execution context – both words are used interchangeably in this document. The RTOS maintains a list of tasks 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. 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.

4.1.3 Potential pitfalls

All of this sounds great, but there are three serious bugs that 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.

4.1.4 Queues

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".

4.1.5 Semaphores

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.

4.1.6 Mutexes

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 I^{2}C interface, a shared block of memory, etc.

4.1.7 Events flags

Event flags 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.

4.1.8 Interrupts

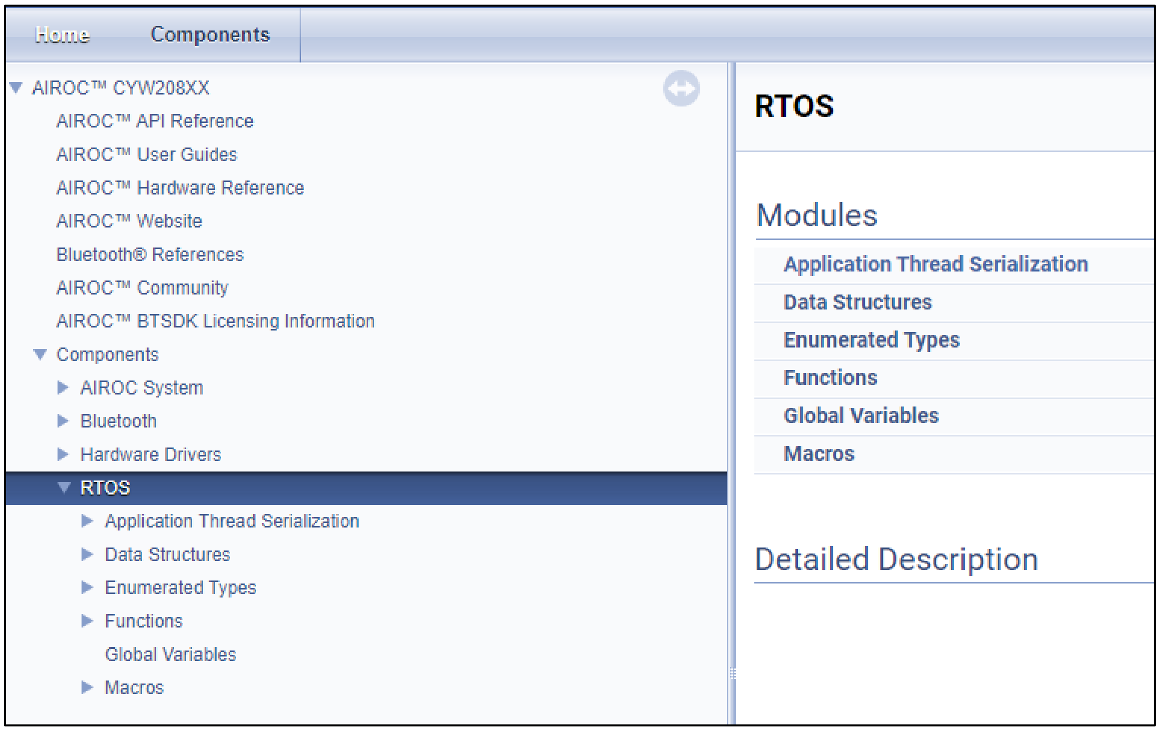
An interrupt is a signal to the processor that an event has occurred, and that immediate attention is required. While they are not an RTOS function, they do interact with the RTOS. An interrupt is handled with an interrupt service routine (ISR). An interrupt is a function, not a thread, so it should 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 threads 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.

4.2 RTOS Abstraction Layer

AIROC™ Bluetooth® SDK devices have the ThreadX RTOS built into the device ROM and the license is included for anyone using WICED chips.

The BTSDK has a built-in abstraction layer that provides a higher-level interface to the fundamental RTOS functions. You can find the documentation for the AIROC™ Bluetooth® SDK RTOS abstraction layer in the API guide under **Components > RTOS**. Typically, the application firmware will use the abstraction layer API to interact with the RTOS, but the underlying RTOS commands (ThreadX in this case) can also be used directly for functions that the abstraction layer does not support.



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.
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.

4.3 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.

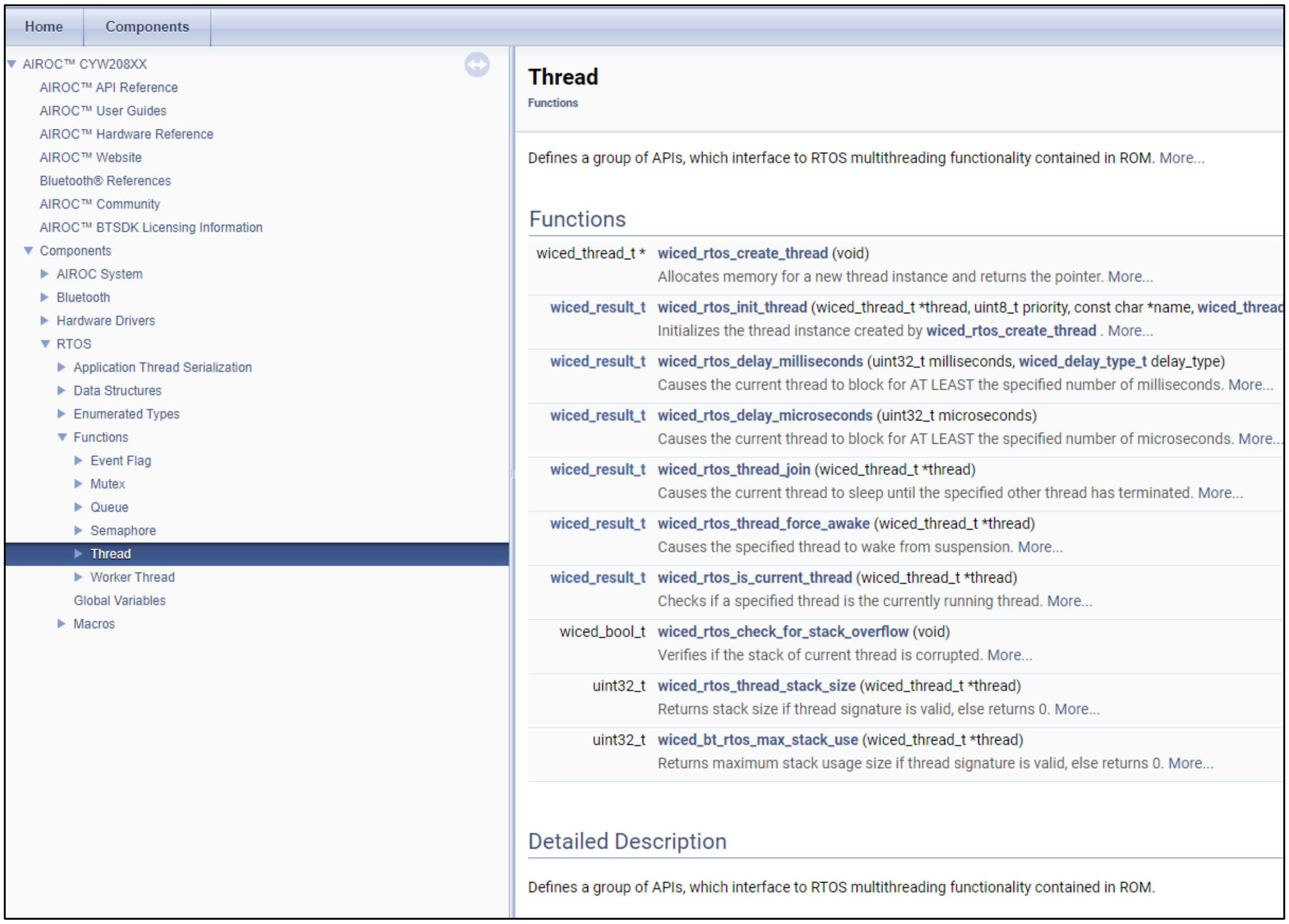
The body of a thread looks just like the `main` function of a typical C application. Often a thread will run forever so it will have an initialization section and a `while(1)` loop that repeats forever. For example:

```
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 BTSDK API reference.



Thread

Functions

Defines a group of APIs, which interface to RTOS multithreading functionality contained in ROM. More...

Functions

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_threa** 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...
uint32_t	**wiced_rtos_thread_stack_size** (wiced_thread_t *thread) Returns stack size if thread signature is valid, else returns 0. More...
uint32_t	**wiced_bt_rtos_max_stack_use** (wiced_thread_t *thread) Returns maximum stack usage size if thread signature is valid, else returns 0. More...

Detailed Description

Defines a group of APIs, which interface to RTOS multithreading functionality contained in ROM.

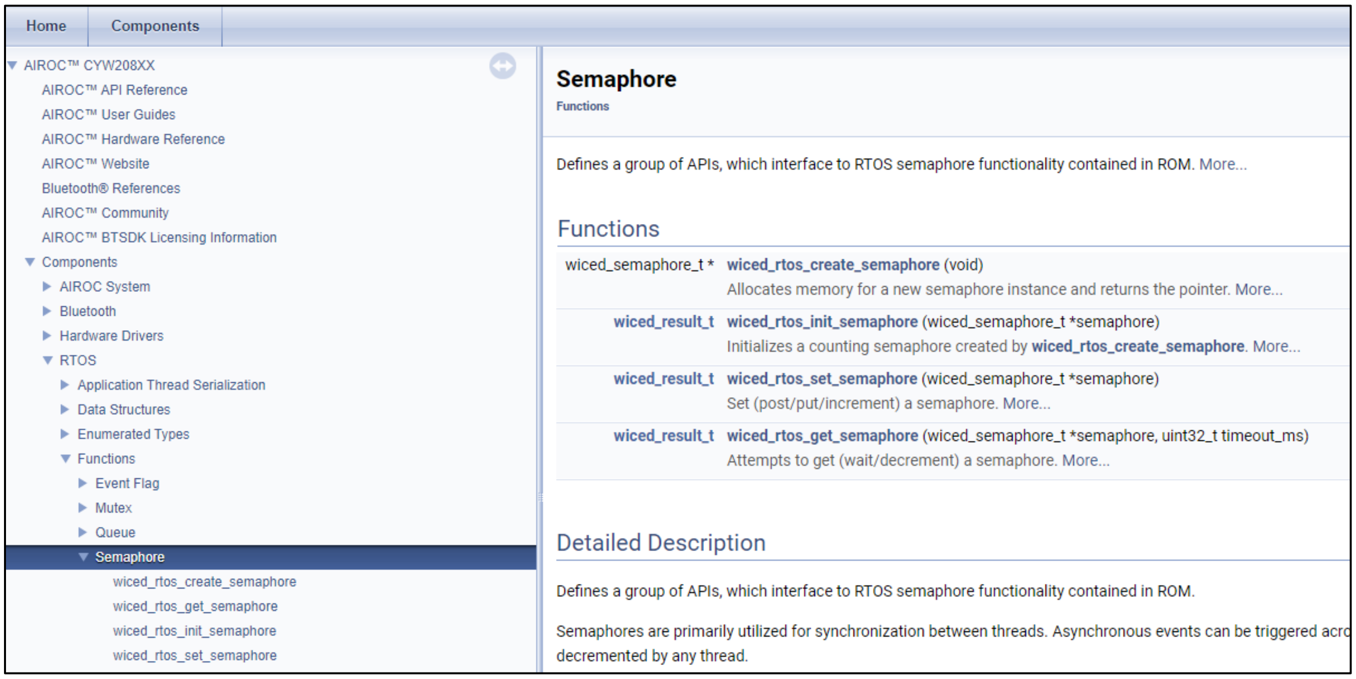
4.4 Semaphore

You will use this in Exercise 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**.



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...

Detailed Description

Defines a group of APIs, which interface to RTOS semaphore functionality contained in ROM.

Semaphores are primarily utilized for synchronization between threads. Asynchronous events can be triggered acro
decremented by any thread.

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 );
```

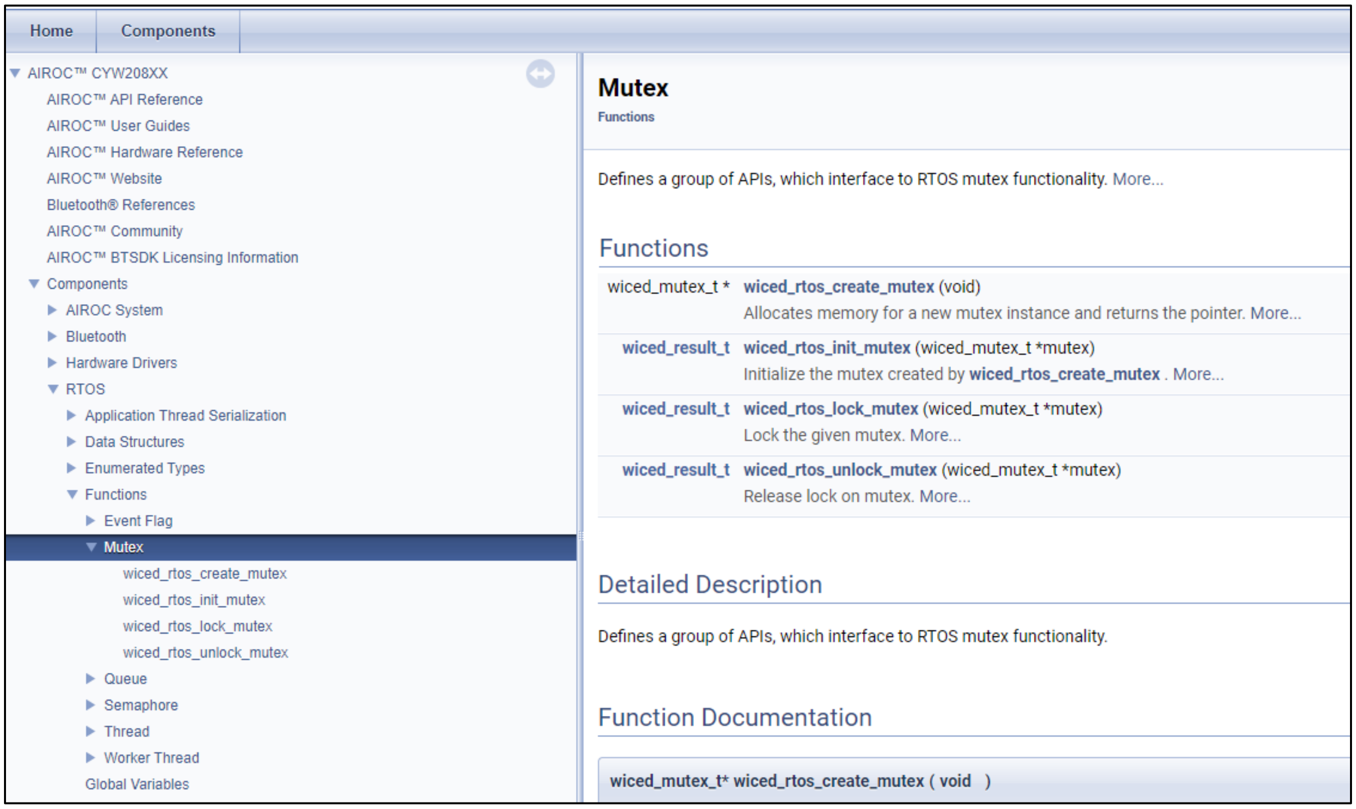
It is generally not a good idea to use a semaphore get inside an interrupt callback with a non-zero timeout since it may lock up your program waiting for a set that never occurs.

4.5 Mutex

You will use this in Exercise 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**.



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...

Detailed Description

Defines a group of APIs, which interface to RTOS mutex functionality.

Function Documentation

wiced_mutex_t* wiced_rtos_create_mutex (void)

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: *A mutex can only be unlocked by the same thread that locked it.*

4.6 Queue

You will use this in Exercise 3:

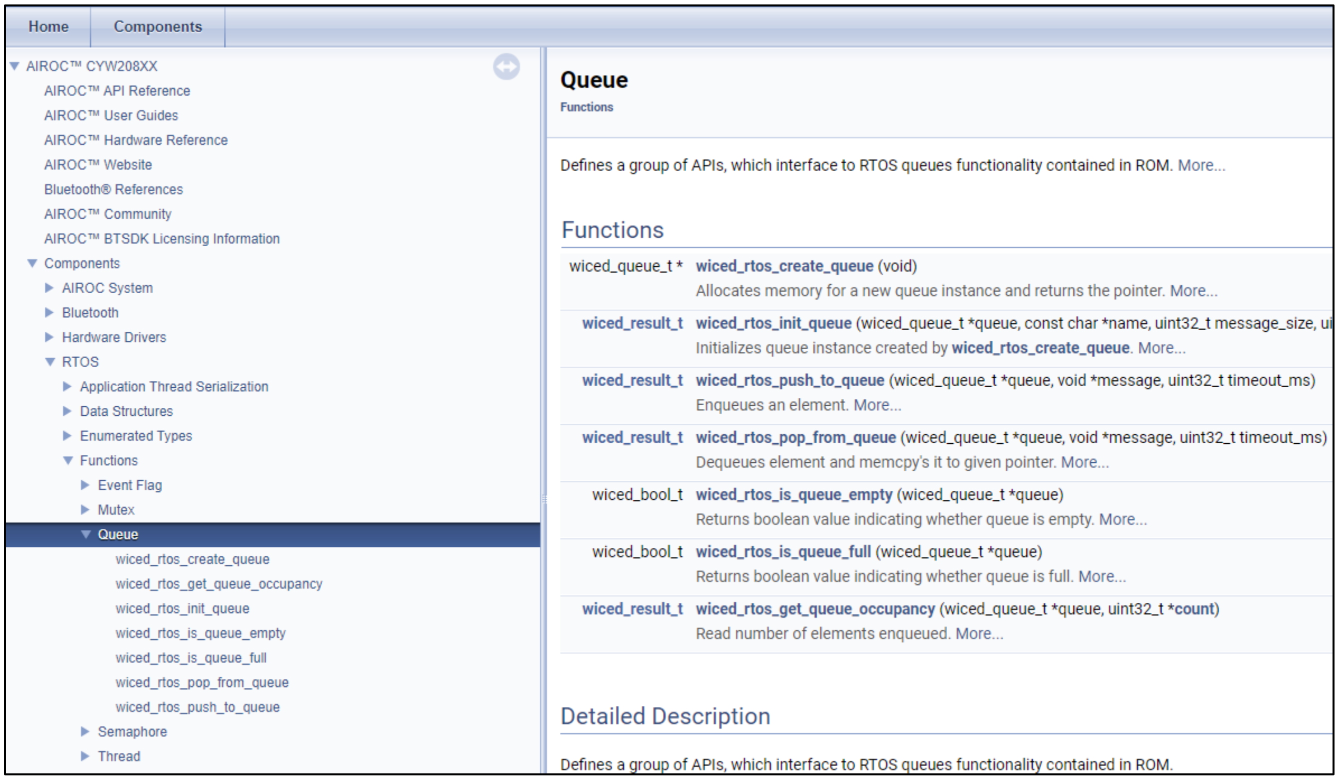
A queue is a thread-safe mechanism to send data to another thread. The queue is a FIFO - you read from the front and you write to the back. If you try to read a queue that is empty your thread will suspend until something is written into it. The payload in a queue (size of each entry) and the size of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



Queue
Functions

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 message_size, ui 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 timeout_ms) Enqueues an element. More...
wiced_result_t	**wiced_rtos_pop_from_queue** (wiced_queue_t *queue, void *message, uint32_t timeout_ms) 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...

Detailed Description

Defines a group of APIs, which interface to RTOS queues functionality contained in ROM.

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 );
```

The queue initialization function creates a separate private memory buffer pool so you must increase `max_number_of_buffer_pools` by the number of queues that are initialized (i.e. one per call to `wiced_rtos_init_queue`). This setting is in the `wiced_bt_cfg_settings_t` structure, which is usually defined in the file app_bt_cfg.c.

4.7 Exercises

Exercise 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 4.4.

☐ 1. Create a new application called **ch04_ex01_semaphore** using the ch04 template.

☐ 2. Create a new semaphore pointer as a global variable, then create and initialize the semaphore when the Bluetooth stack is enabled.

Note: You need both a create function call and an initialize function call.

Note: 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.

Note: 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.

Note: 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 2: 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 4.5.

☐ 1. Create a new application called **ch04_ex02_mutex** using the ch04_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: Queues

Use a queue to send a message to indicate the number of times to blink an LED. This material is covered in 4.6.

☐ 1. Create a new application called **ch04_ex03_queue** using the ch04 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.

Note: 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.

Note: 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).

4.8 Appendix

Answers to the questions asked in the exercises above are provided here.

4.8.1 Exercise 1 Answers

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

   No, because the `wiced_rtos_get_semaphore` will cause the thread to suspend each time through the infinite loop while it waits for another button press.

4.8.2 Exercise 2 Answers

1. Before you added the mutex, how did the LED behave when you pressed the button?

   The LED flashes in an irregular pattern.

2. What changed when you added the mutex?

   The LED flashes slowly (2Hz) when the button is not pressed and then flashes quickly (5Hz) when the button is pressed.

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

   The other thread will never execute because it will never be able to access the mutex. That is:

   - If you don't unlock the mutex in the fast LED thread, the slow LED thread will not execute. The LED will blink fast when the button is pressed but will not blink at all when the button is released.
   - If you don't unlock the mutex in the slow LED thread, the fast LED thread will never execute so the LED will blink slowly no matter what happens with the button.

Published by

Infineon Technologies AG

81726 Munich, Germany