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

Time 2 Hours

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. You will also understand how to configure and run the debugger.

3.1   An Introduction to RTOS

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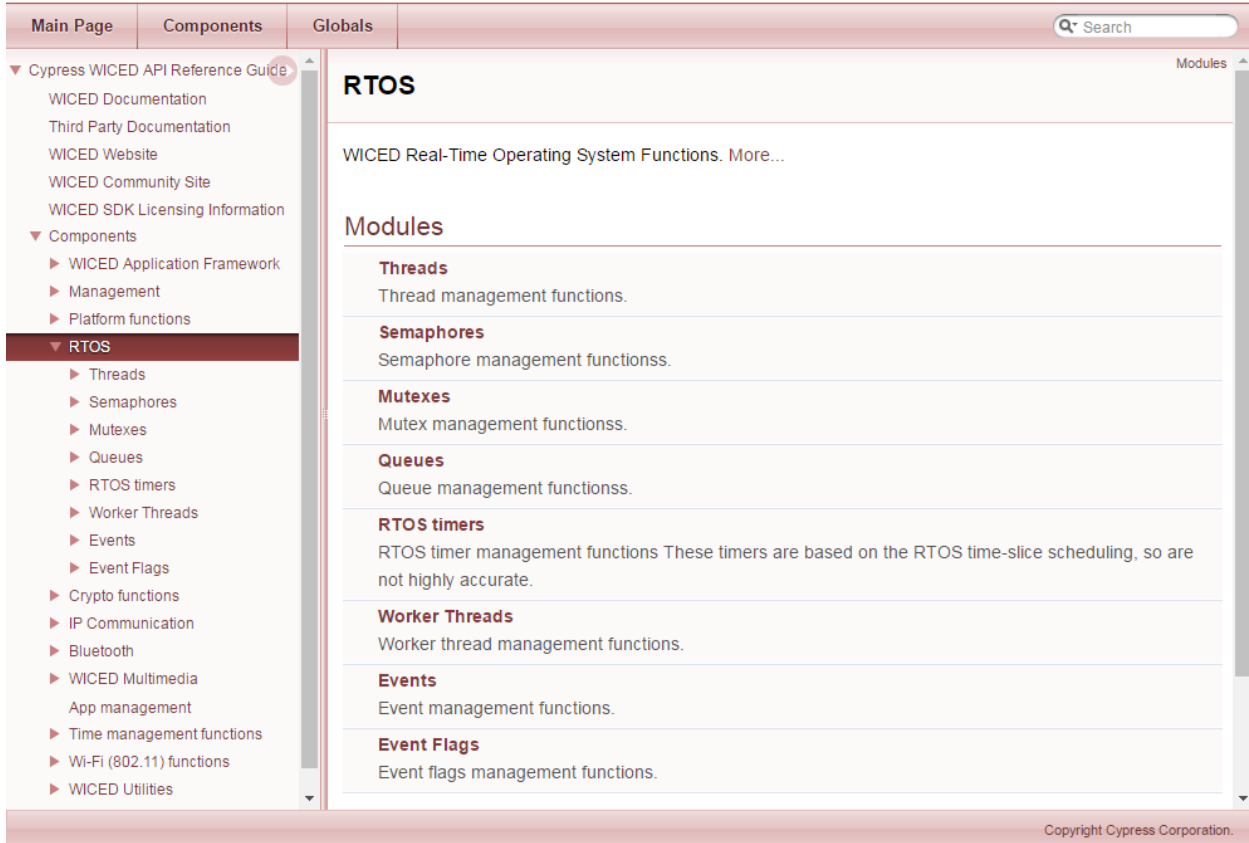
In co-operative multitasking each process must be a good citizen and yield control back to the RTOS. There are a number of mechanisms for yielding control such as rtos_delay_milliseconds, semaphores, mutexes, and queues (which we will discuss later in this document).

The WICED RTOSs are configred in a hybrid mode. It is preemptive for higher priority tasks but cooperative for equal priority tasks. 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.

3.2 WICED RTOS Abstraction Layer

Currently WICED Studio supports 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.

3.3   Problems with RTOSs

All of this sounds great, but everything is not peaches and cream (or whatever your favorite metaphor for a perfect situation might be). 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. The WICED RTOSs give you mechanisms to deal with these problems, specifically semaphores, mutexes, 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. If you don't need it anymore, free up the data structure with the appropriate de-init function (e.g. *wiced_rtos_deinit_mutex()*).

All these functions need to have access to the data structure, 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()* with the following arguments:

- *wiced_thread_t* thread* – A pointer to a thread handle data structure. This handle is used to identify the thread for other thread functions. You must first create the handle data structure before providing the pointer to the create thread function.
- *uint8_t priority* – This is the priority of the thread.
  - Priorities can be from 0 to 31 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 Wi-Fi Driver (WWD) runs at priority 3. You should usually use a lower priority than that.
  - The application_start thread runs at priority 7. Therefore, make sure you yield control in that function if you expect any threads with priority 7 or lower to run.
- *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 10000 is overkill but will work for any of the exercises we do in this class. Almost any thread needs more than 100).
  - A method to print the max stack size used by a thread will be shown later.
- *wiced_thread_arg_t *arg* – A generic argument which will be passed to the thread. It is a uint32.
  - 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      (10)
#define THREAD_STACK_SIZE    (10000)
.
.
wiced_thread_t mySpecialThreadHandle;
.
.
wiced_rtos_create_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 *wiced_thread_arg_t* and must have a *void* return.

The body of a thread looks just like the "main" application_start function of your application (in fact, that function is just a thread that gets initialized automatically). Often a thread will run forever (just like main) so it will have an initialization section and a while(1) loop that repeats forever. For example:

```
void mySpecialThread(wiced_thread_arg_t arg)
{
    const int delay=100;
    while(1)
    {
        processData();
        wiced_rtos_delay_milliseconds(delay);
    }
}
```

Note: you should usually put a *wiced_rtos_delay_milliseconds()* 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 since the main application is just another thread. 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.

Note that if the main application thread (application_start) only does initialization and starts other threads, then you can eliminate the while(1) loop completely from that function. In that case, after the other threads have started, the application_start function will just exit and will not take up any more CPU cycles. If you do that, make sure that any variables that are needed outside that thread (such as thread handles, semaphore handles, etc.) are declared as globals outside of application_start. Otherwise they will be undefined once application_start exits.
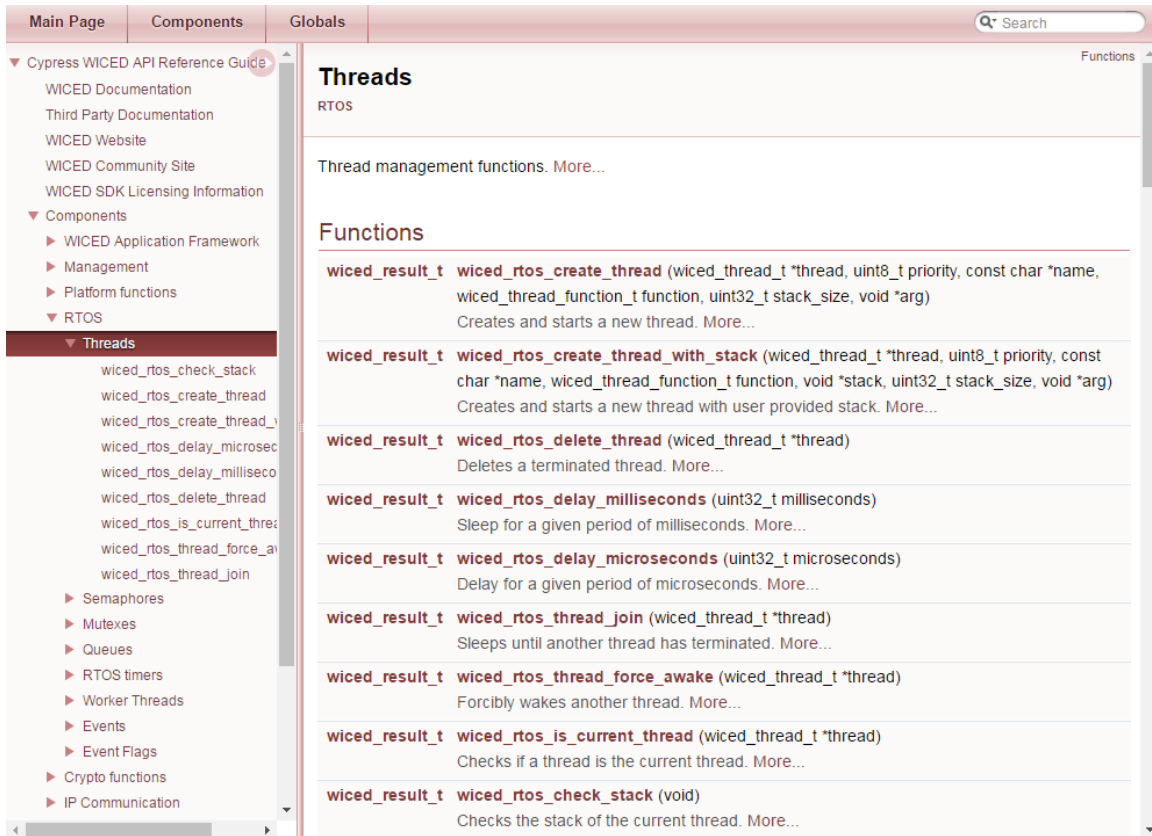
If you want to see how much stack a thread is using, you need to use the underlying RTOS functions for the RTOS you are using. For example, if you are using ThreadX (the default RTOS in WICED) you could add the following function:

```
/* This function will print out the max amount of stack a thread has used */
/* This is ThreadX specific code so it will only work for ThreadX RTOS */
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;
}
```

The function above will return the max number of bytes that the stack has used since it was started. You can call this function occasionally in your project (e.g. in a timer) and then print the value. For example:

```
WPRINT_APP_INFO(("Max Stack: %d\n",(int) maxStackUsage((TX_THREAD*) &mySpecialThreadHandle)));
```

The functions available to manipulate a thread are in the "Component→RTOS→Threads" section of the API guide.

3.5   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. 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 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.

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

If you are using a semaphore get function inside of an ISR or a timer function, you MUST use WICED_NO_WAIT as the timeout. Using a non-zero timeout is not supported in those cases. In that case, the thread will not wait even if the semaphore is not set.

3.6   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. In the exercises for this chapter you will create two different threads that blink the same LED. 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→Mutex.

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

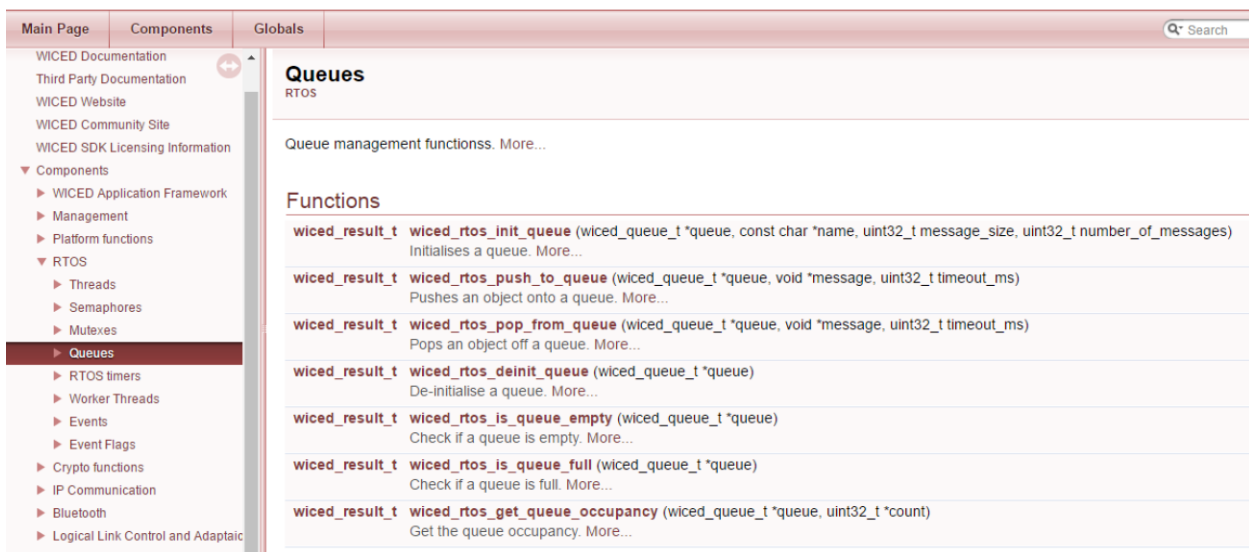Note that a mutex can only be unlocked by the same thread that locked it.

3.7 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_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 project 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.

You should always initialize a queue <u>before</u> starting any threads that use it. Otherwise, you may see unpredictable behavior.

The message size in a queue must be a multiple of 4 bytes. Specifying a message size that is not a multiple of 4 bytes will result in unpredictable behavior. It is good practice to use uint32_t as the minimum size variable (this is true for all variables since the ARM core processor is 32-bits).

If you are using a queue push or pop function inside of an ISR or a timer function, you MUST use WICED_NO_WAIT as the timeout. Using a non-zero timeout is not supported in those cases.

3.8   Timer
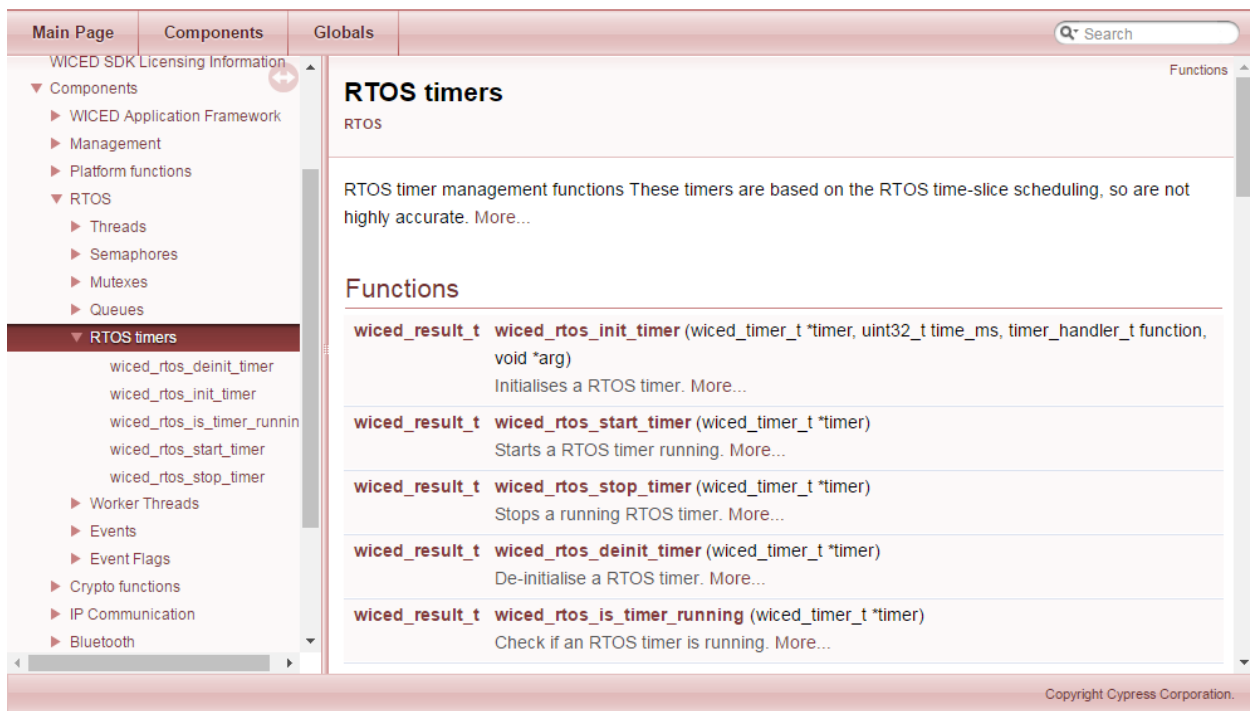
An RTOS timer allows you to schedule a function to run at a specified interval - e.g. send your data to the cloud every 10 seconds.

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, but the function itself must still have the *void* arg* argument in its definition.

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 a while(1) loop – it should just run and exit each time the timer calls it.

The timer is a function, not a thread. Therefore, make sure you don't exit the main application thread if your project has no other active threads.

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

3.9   Exercise(s)

Exercise - 3.1 Thread Blinking LED

Create a thread to blink an LED every 500ms

1. Make a new folder under the ww101 folder called 03 to hold the chapter 3 exercises. Copy the 02/02_blinkled project into the 03 folder. Rename the project to 01_thread. Update the makefile and create a make target.
2. Setup a new thread to blink the LED on/off every 500ms.
    a. Hint: Move the code from the 02_blinkled project's main application loop into the thread's loop.
    b. Hint: If there is nothing to be done in the main application loop, then you can just remove the while(1) loop entirely from application_start. If you leave the loop in, you need a delay such as *wiced_rtos_delay_milliseconds(1)* so that the LED thread gets a chance to run. If you remove the while(1) loop, make sure any variables that need to stick around (such as thread handles) are declared outside the application_start thread since they will become undefined once application_start exits..
3. Program your project to the board.

Exercise - 3.2 Semaphore

Create a program where the main 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 makefile and create a make target.
2. Create a new semaphore.
3. Look for a button press in the main application loop and set the semaphore when the button is pressed.
    a. Hint: You can use a pin interrupt to detect the button press and set the semaphore.
    b. Hint: Make sure you add a delay to the main thread so that the other thread gets a chance to run or remove the while(1) loop completely and let the main thread exit.
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.
    a. Hint: You can replace the delay function calls with the get semaphore function.
    b. Hint: If the thread has "blink" in its name you should rename it to be consistent with what it now does.
    c. Hint: Use WICED_WAIT_FOREVER so that the thread will wait until the button is pressed.

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?

Exercise - 3.3 (Advanced) Mutex

An LED may behave strangely if two threads try to blink it at the same time. Use a mutex to lock access.

1. Create a new project called 03_mutex.
2. Create two threads that do the following:
   a. Thread 1 will blink LED1 with an ON and OFF delay of 150ms while Button 1 is being pressed.
   b. Thread 2 will blink the same LED (LED1) with an ON and OFF delay of 100ms while Button 2 is being pressed.
   c. Make sure you yield control in the thread when the button is not being pressed.

      As an example, Thread 1 should look like this:

```
void led1Thread(wiced_thread_arg_t arg)
{
  while(1)
  {
    while(!wiced_gpio_input_get( WICED_BUTTON1 )) // Loop while button is pressed
    {
        wiced_gpio_output_low( WICED_LED1 );
        wiced_rtos_delay_milliseconds(150);
        wiced_gpio_output_high( WICED_LED1 );
        wiced_rtos_delay_milliseconds(150);
    }
    wiced_rtos_delay_milliseconds(1); // Yield control when button is not pressed
  }
}
```

3. In the main application, just setup the two threads and get them running.
   Create a Make Target and Program your project to the board.
   Press button 1 and button 2 separately to observe the blink rates. Then press both buttons simultaneously. Do you see issues with the blinking?
4. Add a mutex to the project so that when you press button 1 it will ignore button 2 and vice versa. That is, the LED blink rate will follow the first button that was pressed.

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

Exercise - 3.4 (Advanced) 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 makefile and create a make target.
2. Remove the semaphore(s) 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.
   a. Hint: remember to use WICED_NO_WAIT for the timeout parameter in the ISR. Otherwise the push function will not work.
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.

Exercise - 3.5 (Advanced) Timers

Make an LED blink using a timer.

1. Copy 01_thread to 05_timer. Update the makefile 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()*.
   a. 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:

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

What happens if the application_start doesn't have a while(1) loop? Why?

Does the while(1) loop in application_start need a delay? Why or why not?

Exercise - 3.6 (Advanced) Setup and Run the Debugger

Make Target

In order to use the debugger, create a new make target for an existing project so that *–debug* is added after the platform name (with no space) and remove run from the end of the target. That is, the target should look like:

<folder1>.[<folder2>…].<project>-<platform>-debug download

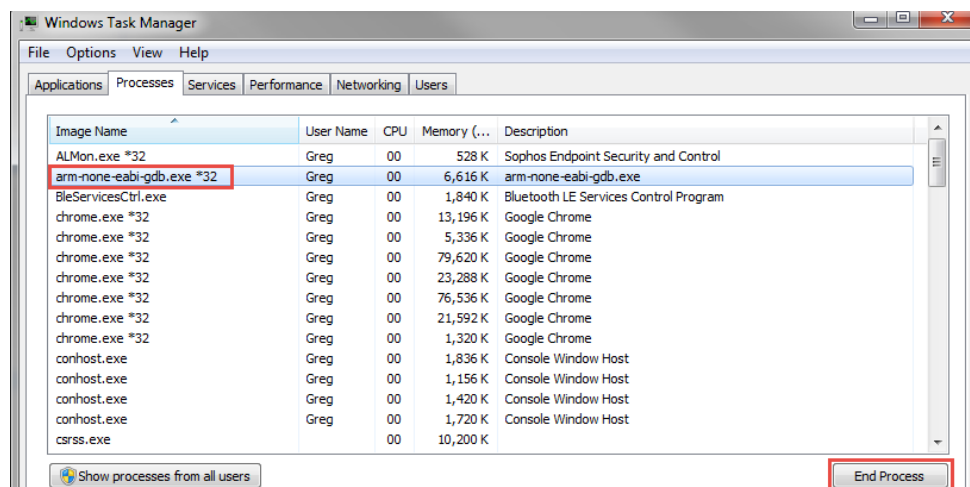For example, the make target for the 02_blinkled project from the previous chapter would be:

ww101.02.02_blinkled-WW101_2_CYW943907AEVAL1F-debug download

Running the Debugger

First, execute the make target created above to download the program to the board. Once the project is downloaded, click the down arrow next to the green bug icon and select "43xxx_Wi-Fi_Debug_Windows". If you get a message asking if you want to open the debug perspective, click "Yes". You can click the check box to tell the tool to switch automatically in the future.
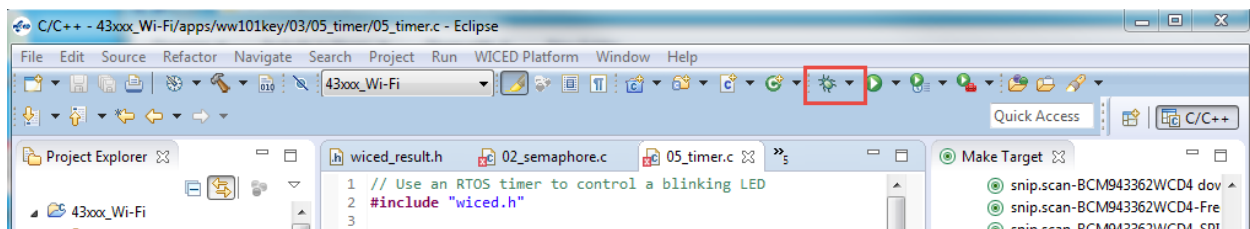
Note: If you get an error when trying to launch the debugger you may need to terminate an existing debug process. Open the Windows Task Manager, select the Process tab, click on "Image Name" to sort by the process name and terminate processes called "arm-none-eabi-gdb" and "openocd-all-brcm-libftdi".

You may also need to set some of the debug options. These can be found in the community article at:

https://community.cypress.com/community/wiced-wifi/wiced-wifi-forums/blog/2014/05/09/creating-andor-editing-debug-configurations

If you still get an error, execute the "clean" make target and then re-execute the make target for the project that you want to debug.

When the debugger starts, you will be in the "Debug Perspective". The session will halt in the start_GCC.s file. The top of the window will look something like this:
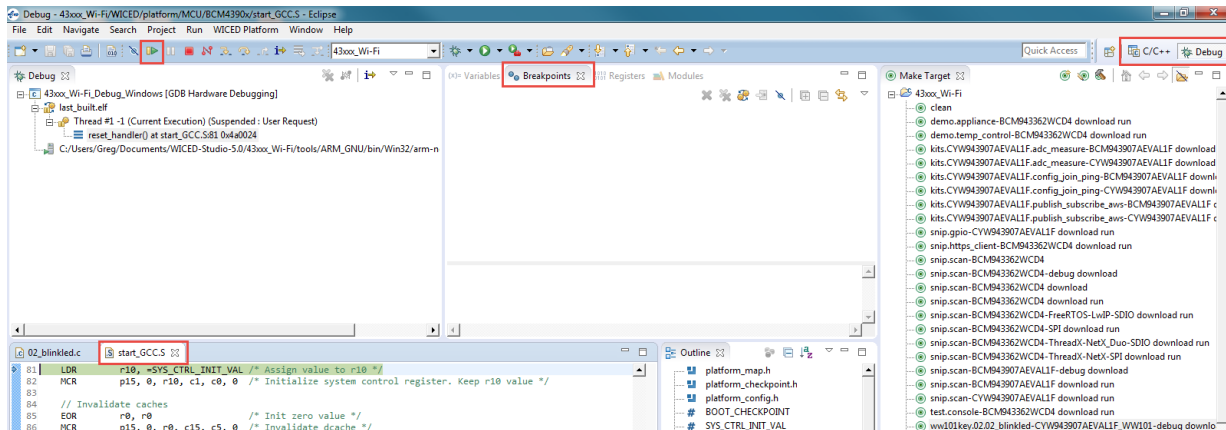
In order to add a breakpoint, open the source file (such as 02_blinkled.c), click on the line where you want a breakpoint and press Ctrl-Shift-B or from the menu select "Run > Toggle Breakpoint". If you need to see the project explorer window in order to open the source file, click on "C/C++" in the upper right corner in order to switch to the C/C++ Perspective. Once you have opened the file, switch back to the Debug Perspective.

Click the "Resume" button (shown in the figure above) to resume execution. The program will halt once it reaches the breakpoint.
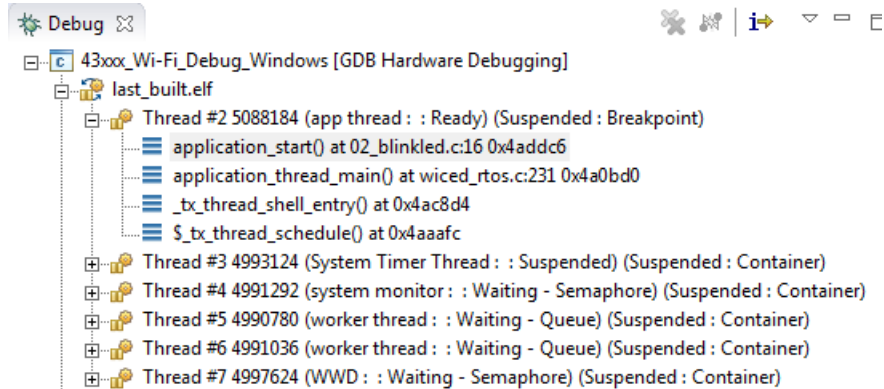

Once a thread suspends due to a breakpoint you will see that line of code highlighted in green as shown above and you will see that the thread is suspended due to the breakpoint in the debug window as shown below.

You can enable or disable breakpoints by double clicking on the green circle next to the line in the source code or from the "Breakpoints" window. If you don't see the Breakpoints tab, use the menu item "Window > Show View > Breakpoints".

If breakpoints are created prior to starting the current debug session, they will not be associated with the current thread and will be indicated with a blue circle without a check mark. To enable the breakpoints in the current thread, right-click the desired breakpoint and select "Breakpoint Properties…" Click on "Filter" and then select the "last_built.elf" check box as shown below.
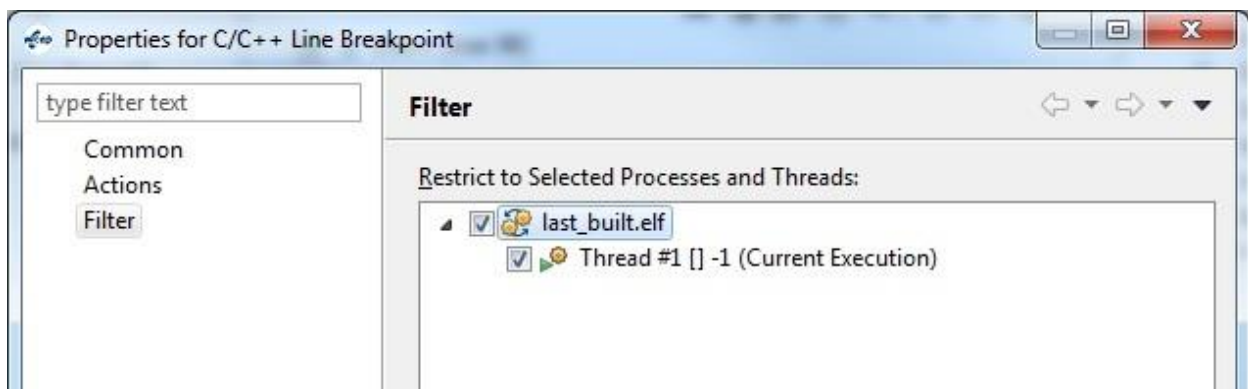
Note: If you do not see any breakpoints in the Breakpoints window, click the "Show Breakpoints Supported by Selected Target" button as shown below.
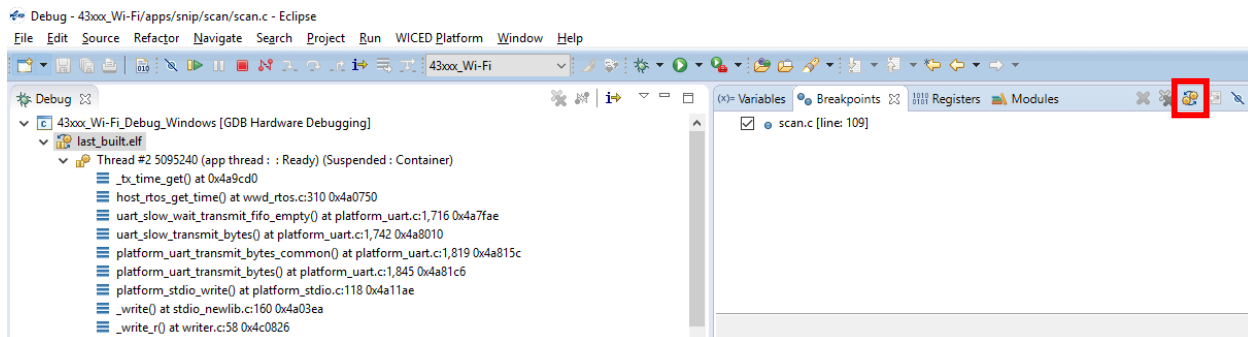
Click the red "Terminate" button to stop debugging. Once you terminate the debugger, you will want to switch back to the C/C++ Perspective by clicking on the button at the top right corner.
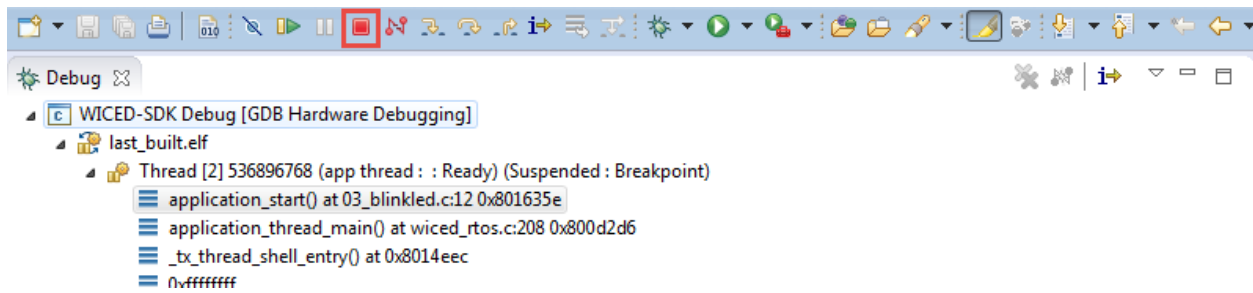
3.10 Related Example "Apps"

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.