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

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

Time: 2 Hours

Fundamentals

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.

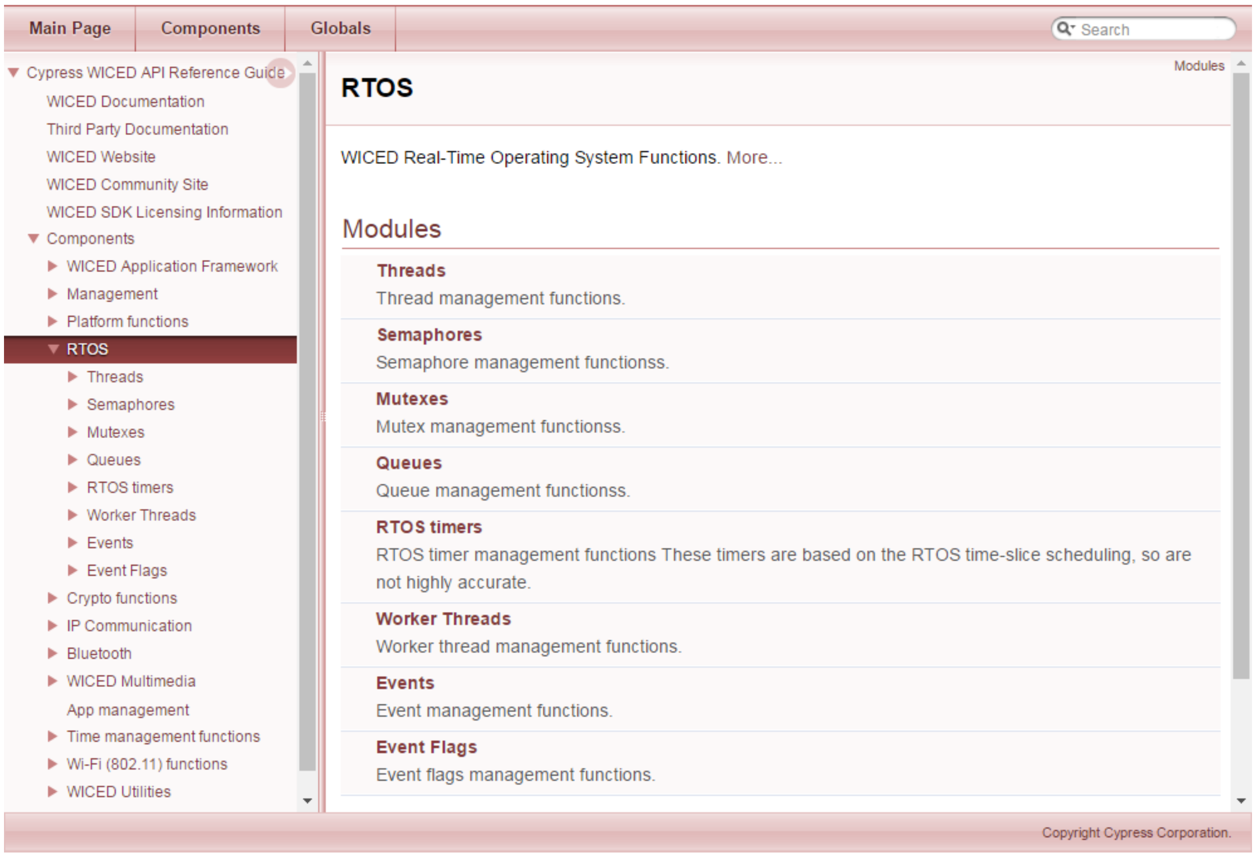
In preemptive multitasking the CPU completely controls which task is running and has the ability to stop and start them as required. In this scheme the scheduler uses CPU protected modes to wrest control from active tasks, halt them, and move onto the next task. Preemptive multitasking is the scheme 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 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.



RTOS

WICED Real-Time Operating System Functions. More...

Modules

Threads
Thread management functions.

Semaphores
Semaphore management functionss.

Mutexes
Mutex management functionss.

Queues
Queue management functionss.

RTOS timers
RTOS timer management functions These timers are based on the RTOS time-slice scheduling, so are not highly accurate.

Worker Threads
Worker thread management functions.

Events
Event management functions.

Event Flags
Event flags management functions.

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 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 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 global variables 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 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.
- *char *name* – A name for the thread. This name is really 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).
- *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      (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.

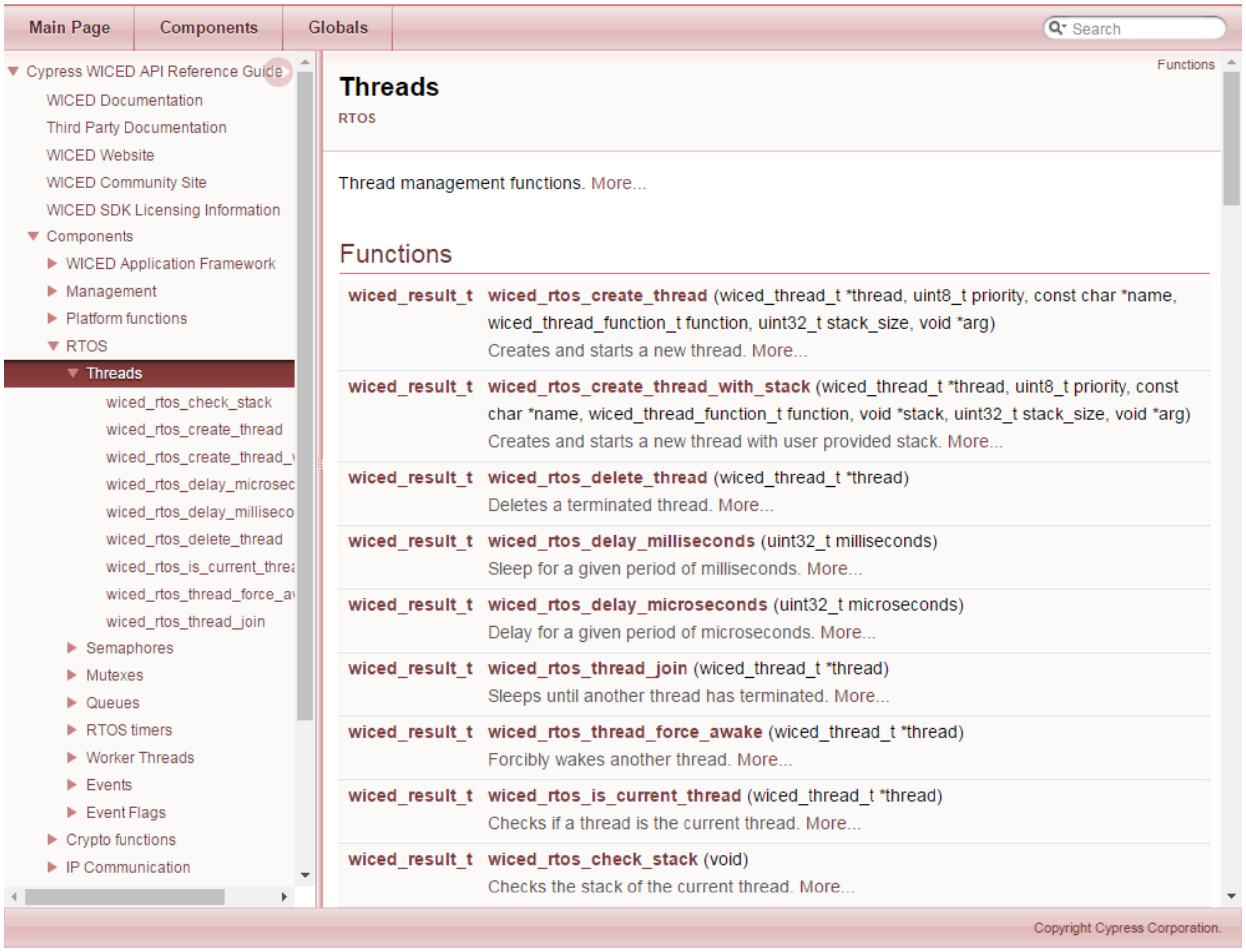
The body of a thread looks just like the "main" function of your application (in fact, the main function is really just a thread that gets initialized automatically). Typically a thread will run forever (just like 'main") so it will have an initialization section and a while(1) loop that repeats forever.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

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

Note: you should (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 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 which will 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 exist and will not take up any more CPU cycles.

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



Threads

RTOS

Thread management functions. More...

Functions

wiced_result_t **wiced_rtos_create_thread** (wiced_thread_t *thread, uint8_t priority, const char *name, wiced_thread_function_t function, uint32_t stack_size, void *arg)
Creates and starts a new thread. More...

wiced_result_t **wiced_rtos_create_thread_with_stack** (wiced_thread_t *thread, uint8_t priority, const char *name, wiced_thread_function_t function, void *stack, uint32_t stack_size, void *arg)
Creates and starts a new thread with user provided stack. More...

wiced_result_t **wiced_rtos_delete_thread** (wiced_thread_t *thread)
Deletes a terminated thread. More...

wiced_result_t **wiced_rtos_delay_milliseconds** (uint32_t milliseconds)
Sleep for a given period of milliseconds. More...

wiced_result_t **wiced_rtos_delay_microseconds** (uint32_t microseconds)
Delay for a given period of microseconds. More...

wiced_result_t **wiced_rtos_thread_join** (wiced_thread_t *thread)
Sleeps until another thread has terminated. More...

wiced_result_t **wiced_rtos_thread_force_awake** (wiced_thread_t *thread)
Forcibly wakes another thread. More...

wiced_result_t **wiced_rtos_is_current_thread** (wiced_thread_t *thread)
Checks if a thread is the current thread. More...

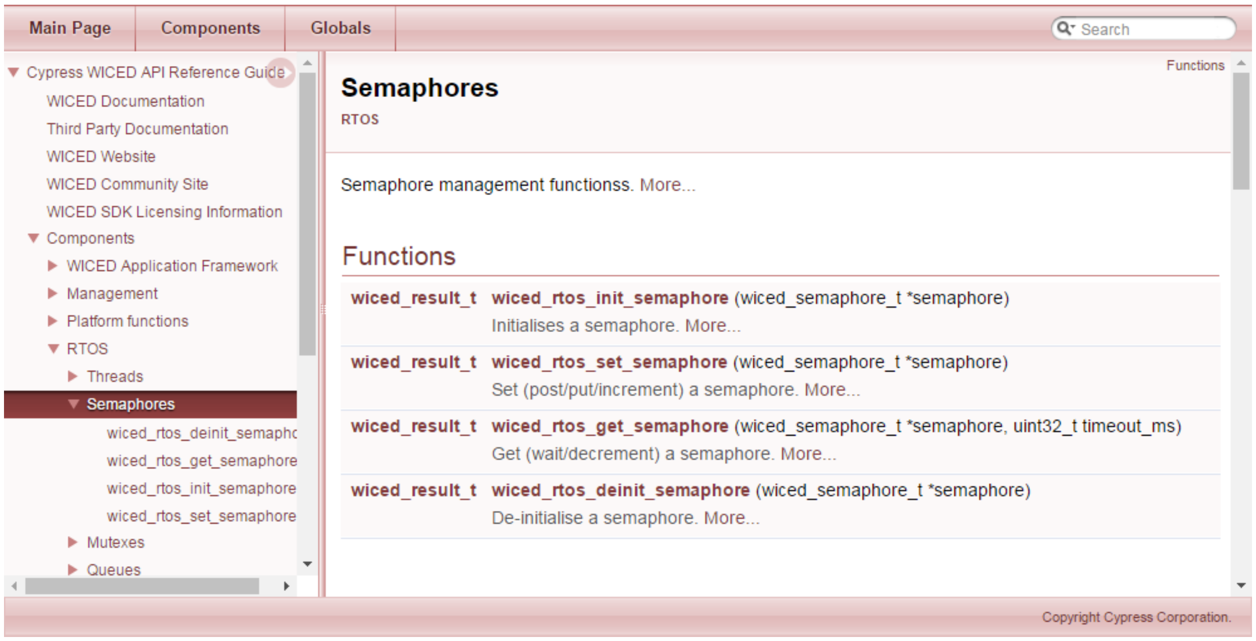
wiced_result_t **wiced_rtos_check_stack** (void)
Checks the stack of the current thread. More...

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 sets the time in milliseconds that the function waits before returning. If you want the thread to wait (almost) 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.



Semaphores

RTOS

Semaphore management functionss. More...

Functions

wiced_result_t	**wiced_rtos_init_semaphore** (wiced_semaphore_t *semaphore) Initialises a 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) Get (wait/decrement) a semaphore. More...
wiced_result_t	**wiced_rtos_deinit_semaphore** (wiced_semaphore_t *semaphore) De-initialise a semaphore. More...

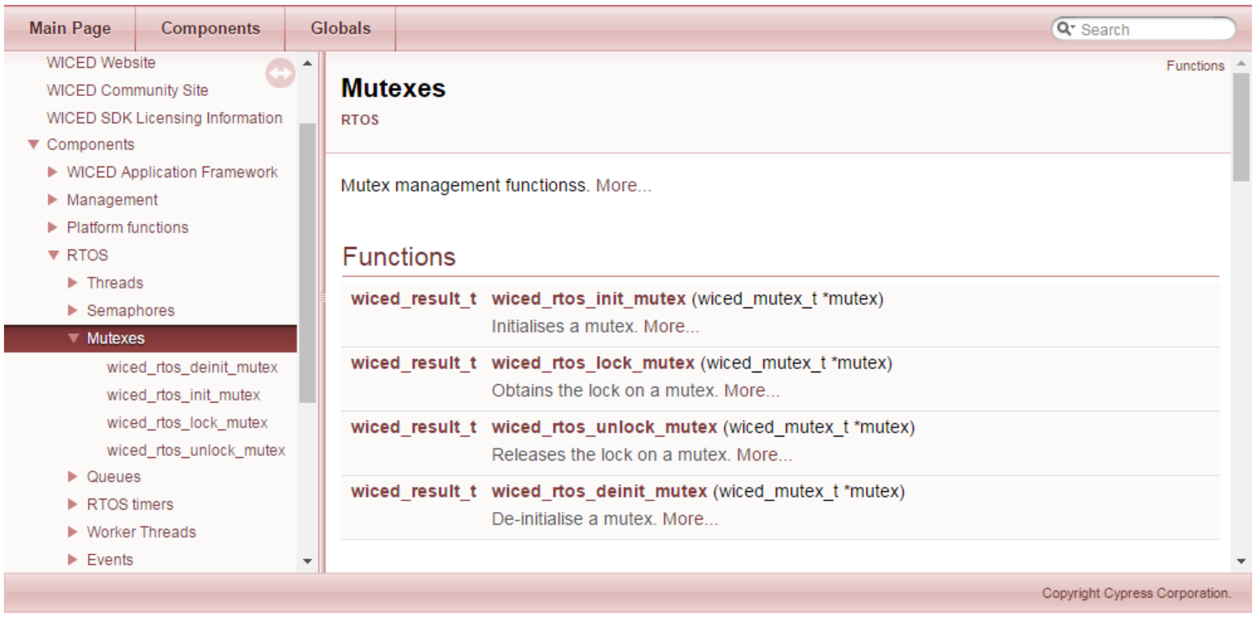
Copyright Cypress Corporation.

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

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 could happen. You can protect yourself by using a mutex.

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



Mutexes
RTOS

Mutex management functionss. More...

Functions

wiced_result_t	**wiced_rtos_init_mutex** (wiced_mutex_t *mutex) Initialises a mutex. More...
wiced_result_t	**wiced_rtos_lock_mutex** (wiced_mutex_t *mutex) Obtains the lock on a mutex. More...
wiced_result_t	**wiced_rtos_unlock_mutex** (wiced_mutex_t *mutex) Releases the lock on a mutex. More...
wiced_result_t	**wiced_rtos_deinit_mutex** (wiced_mutex_t *mutex) De-initialise a mutex. More...

Copyright Cypress Corporation.

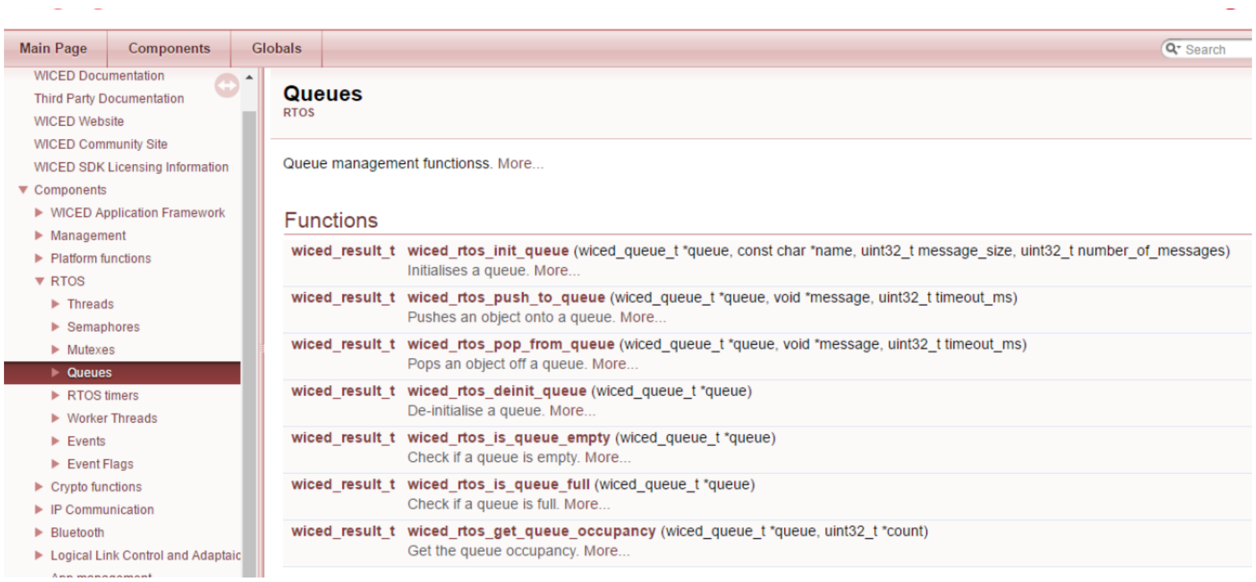
You should always initialize a mutex before starting any threads that use it. Otherwise, you may see unpredictable behavior.

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 sets the time in milliseconds that the function waits before returning if the queue is full. If you want the thread to wait (almost) 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 (almost) 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.



Queues
RTOS

Queue management functionss. More...

Functions

wiced_result_t	**wiced_rtos_init_queue** (wiced_queue_t *queue, const char *name, uint32_t message_size, uint32_t number_of_messages) Initialises a queue. More...
wiced_result_t	**wiced_rtos_push_to_queue** (wiced_queue_t *queue, void *message, uint32_t timeout_ms) Pushes an object onto a queue. More...
wiced_result_t	**wiced_rtos_pop_from_queue** (wiced_queue_t *queue, void *message, uint32_t timeout_ms) Pops an object off a queue. More...
wiced_result_t	**wiced_rtos_deinit_queue** (wiced_queue_t *queue) De-initialise a queue. More...
wiced_result_t	**wiced_rtos_is_queue_empty** (wiced_queue_t *queue) Check if a queue is empty. More...
wiced_result_t	**wiced_rtos_is_queue_full** (wiced_queue_t *queue) Check if a queue is full. More...
wiced_result_t	**wiced_rtos_get_queue_occupancy** (wiced_queue_t *queue, uint32_t *count) Get the queue occupancy. More...

You should always initialize a queue before 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).

On some WICED devices (including the 43907 used in the class), queue APIs do not work properly if they are called from inside an ISR or an RTOS timer function. Therefore, it is recommended to use the queue APIs outside of any ISRs or timer functions.

Timer

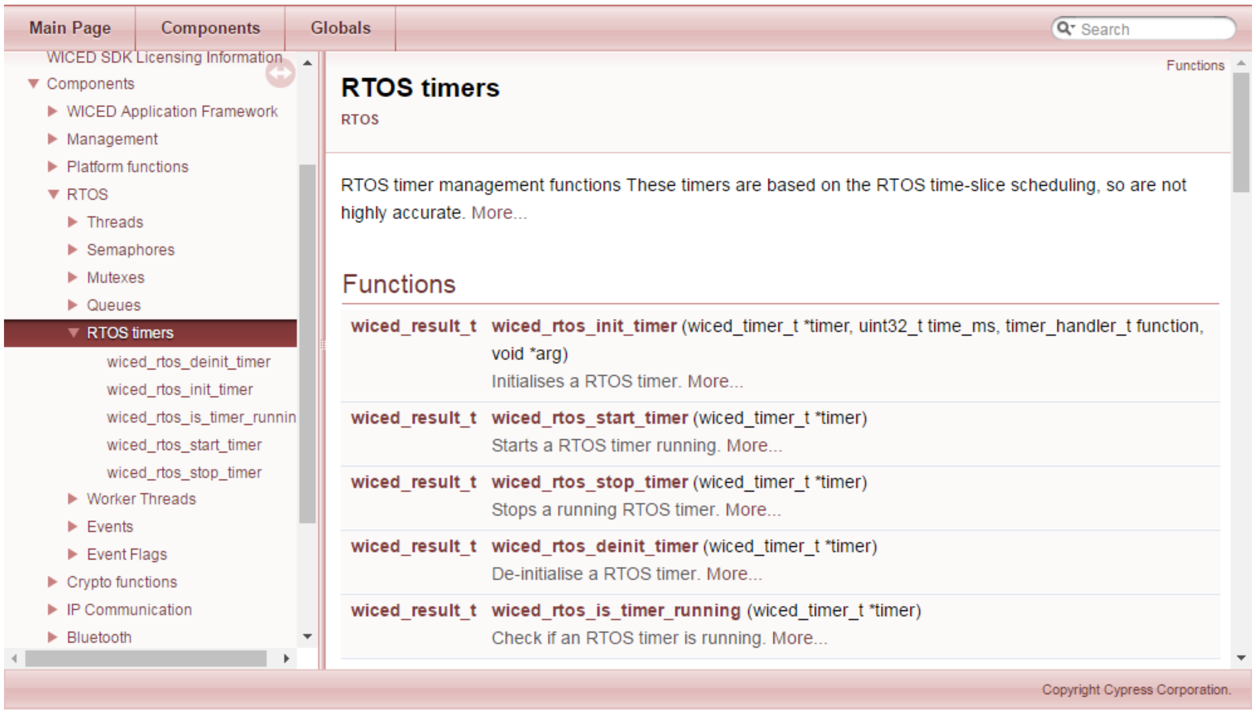
An RTOS timer allows you to schedule a function to run at a specified interval - e.g. send your data to the cloud every 10 seconds.

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.

<u>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 typically NOT have a while(1) loop – it should just run and exit each time the timer calls it</u>.

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.



RTOS timers

RTOS

RTOS timer management functions These timers are based on the RTOS time-slice scheduling, so are not highly accurate. More...

Functions

wiced_result_t	**wiced_rtos_init_timer** (wiced_timer_t *timer, uint32_t time_ms, timer_handler_t function, void *arg) Initialises a RTOS timer. More...
wiced_result_t	**wiced_rtos_start_timer** (wiced_timer_t *timer) Starts a RTOS timer running. More...
wiced_result_t	**wiced_rtos_stop_timer** (wiced_timer_t *timer) Stops a running RTOS timer. More...
wiced_result_t	**wiced_rtos_deinit_timer** (wiced_timer_t *timer) De-initialise a RTOS timer. More...
wiced_result_t	**wiced_rtos_is_timer_running** (wiced_timer_t *timer) Check if an RTOS timer is running. More...

Exercise(s)

01 (THREAD) 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 02/03_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 03_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.
3. Program your project to the board.

02 (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 thread 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.
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: If the thread has "blink" in its name you should rename it to be consistent with what it now does.

<u>Questions to answer:</u>

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 (Advanced) (MUTEX) The WPRINT_APP_INFO may 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 makefile and create a make target.
2. Add a second thread that blinks LED0 at a rate of 498ms.
    a. 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 (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 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 (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?

06 (Advanced) (DEBUGGING) 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:

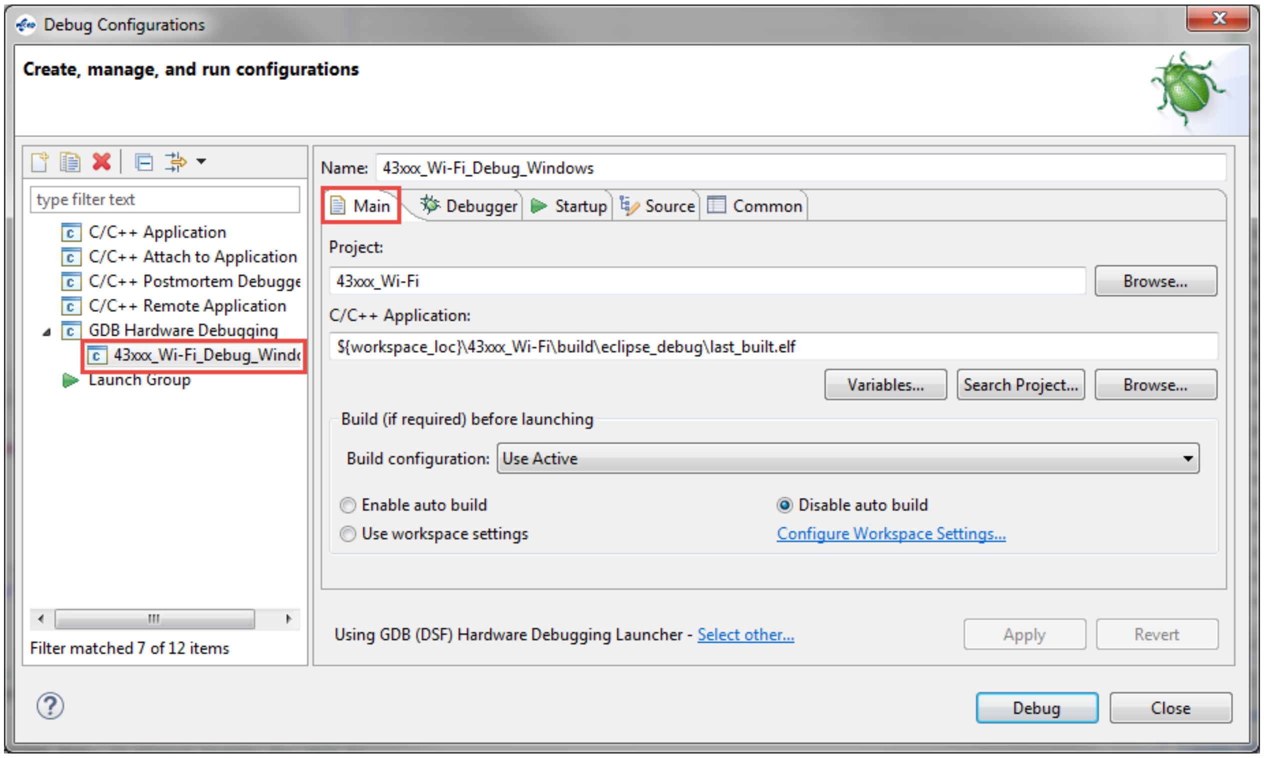
<folder1>.[<folder2>…].<project>-<platform>-debug download

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

ww101.02.02_blinkled-BCM943907_WW101-debug download

Setup

Before starting the debugger, we need to verify that it is setup correctly. From WICED Studio, click the down arrow next to the green bug icon and select “Debug Configurations…” Then select “GDB Hardware Debugging > 43xxx_Wi-Fi Debug_Windows” from the window on the left. Setup the various tabs as shown in the figures below. You should only have to make changes on the “Startup” and “Common” tabs but all are shown here for completeness.



Filter matched 7 of 12 items

Name: 43xxx_Wi-Fi_Debug_Windows

Project: 43xxx_Wi-Fi

C/C++ Application: ${workspace_loc}\43xxx_Wi-Fi\build\eclipse_debug\last_built.elf

Build (if required) before launching

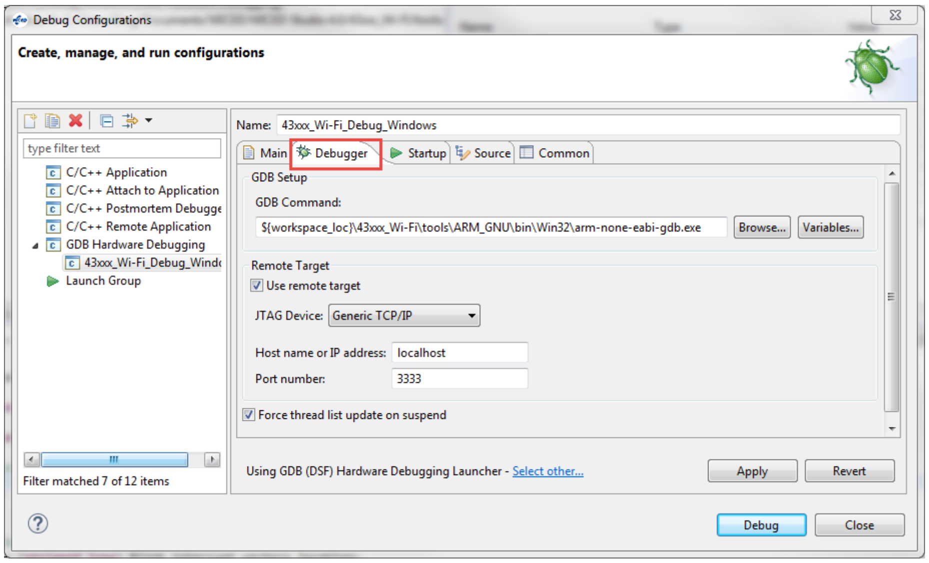
Build configuration: Use Active

Enable auto build [✓ Disable auto build]

Use workspace settings

Configure Workspace Settings...

Using GDB (DSF) Hardware Debugging Launcher - Select other...



Filter matched 7 of 12 items

Name: 43xxx_Wi-Fi_Debug_Windows

GDB Setup

GDB Command: ${workspace_loc}\43xxx_Wi-Fi\tools\ARM_GNU\bin\Win32\arm-none-eabi-gdb.exe

Remote Target

☑ Use remote target

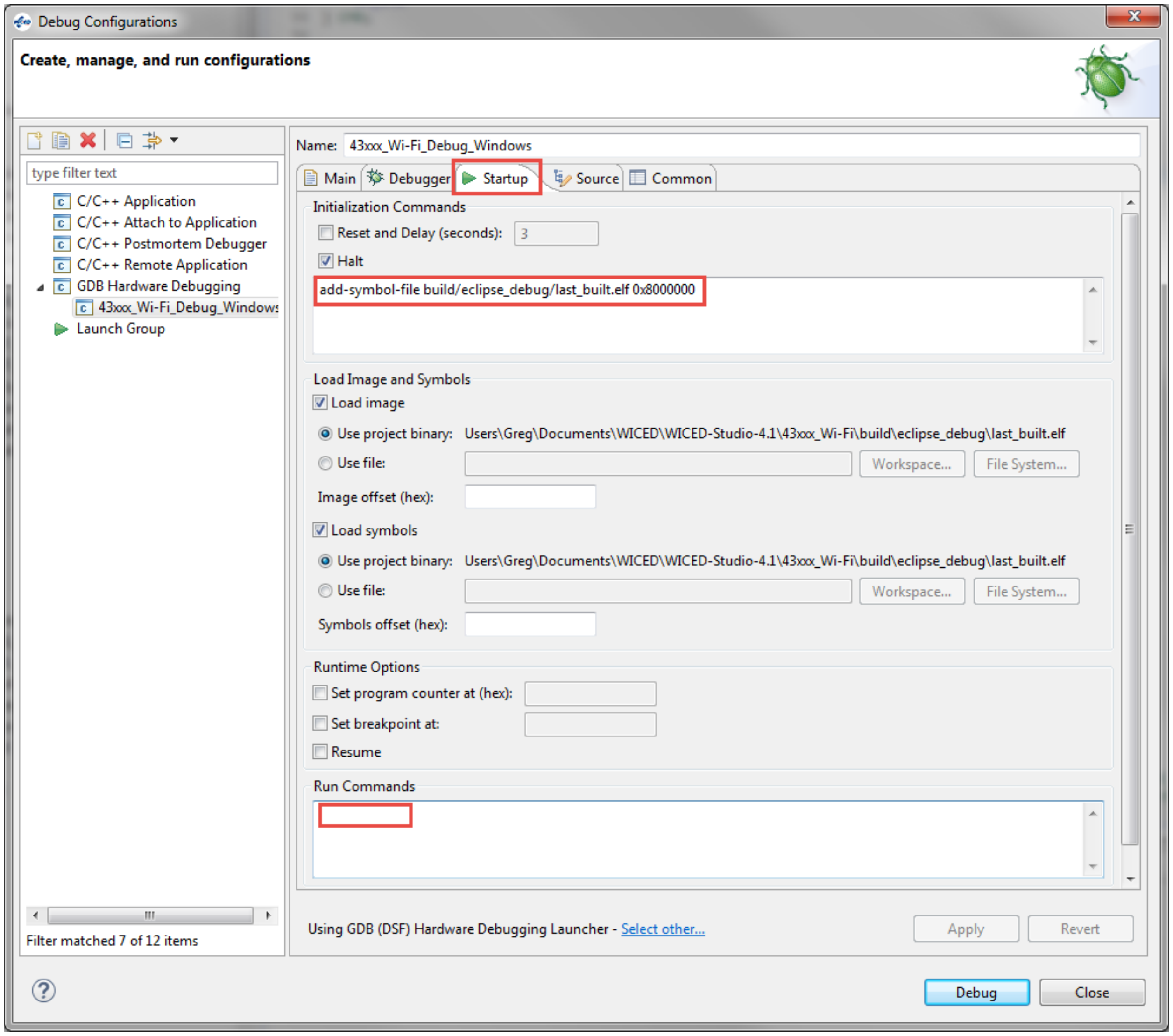
JTAG Device: Generic TCP/IP

Host name or IP address: localhost

Port number: 3333

☑ Force thread list update on suspend

Using GDB (DSF) Hardware Debugging Launcher - Select other...



Debug Configurations

Create, manage, and run configurations

Name: 43xxx_Wi-Fi_Debug_Windows

Main | Debugger | Startup | Source | Common

Initialization Commands

- [ ] Reset and Delay (seconds): 3
- [✓] Halt

```
add-symbol-file build/eclipse_debug/last_built.elf 0x8000000
```

Load Image and Symbols

- [✓] Load image
- (●) Use project binary: Users\Greg\Documents\WICED\WICED-Studio-4.1\43xxx_Wi-Fi\build\eclipse_debug\last_built.elf
- ( ) Use file:
- Image offset (hex):
- [✓] Load symbols
- (●) Use project binary: Users\Greg\Documents\WICED\WICED-Studio-4.1\43xxx_Wi-Fi\build\eclipse_debug\last_built.elf
- ( ) Use file:
- Symbols offset (hex):

Runtime Options

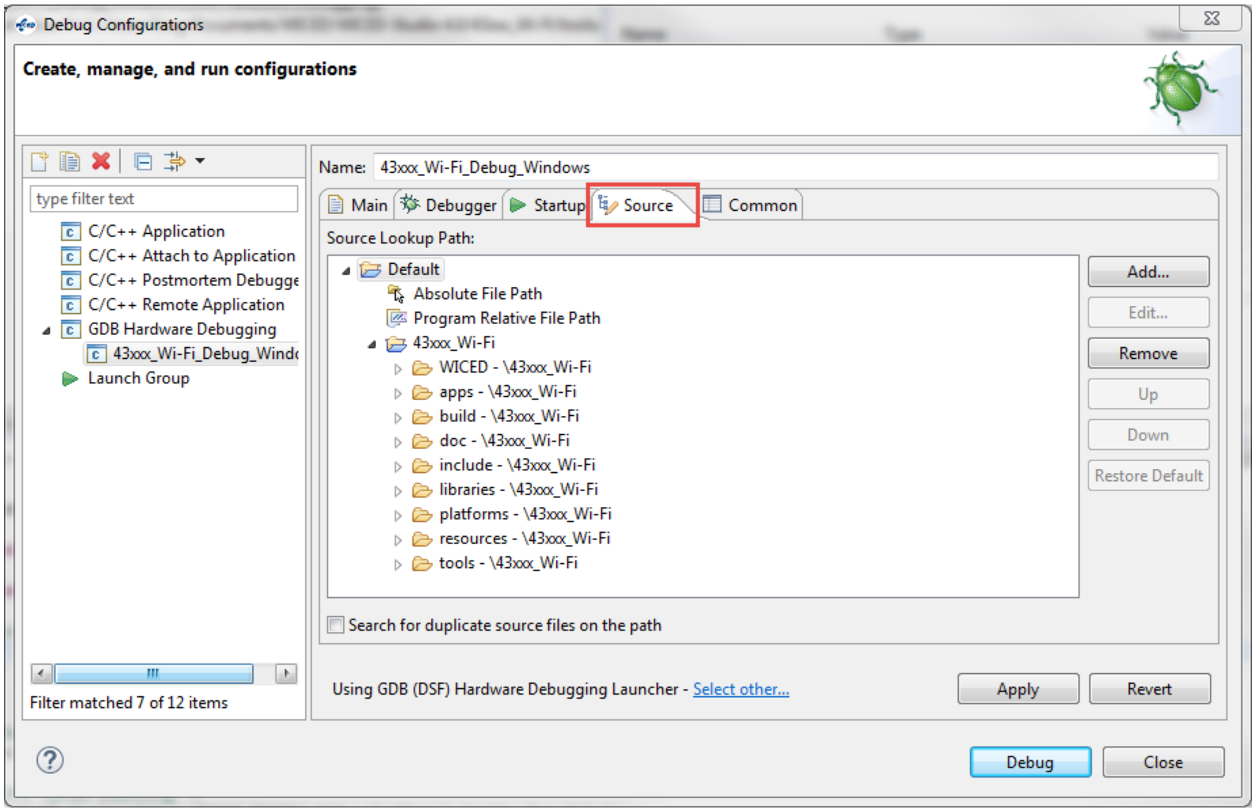
- [ ] Set program counter at (hex):
- [ ] Set breakpoint at:
- [ ] Resume

Run Commands

Using GDB (DSF) Hardware Debugging Launcher - Select other...

Filter matched 7 of 12 items

Note: the text in the box above is: ***add-symbol-file build/eclipse_debug/last_built.elf 0x8000000***


Debug Configurations

Create, manage, and run configurations

Name: 43xxx_Wi-Fi_Debug_Windows

Main | Debugger | Startup | Source | Common

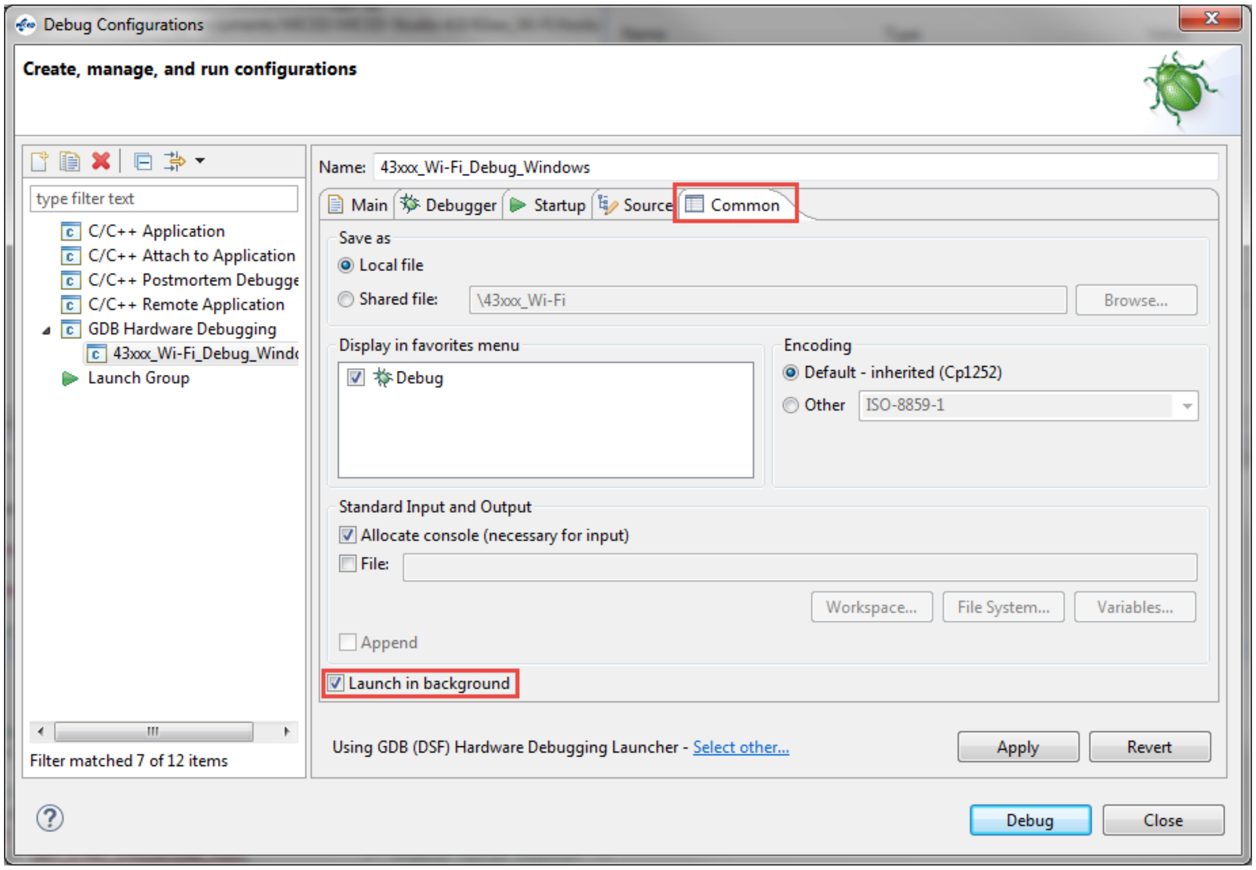
Source Lookup Path:

- Default
  - Absolute File Path
  - Program Relative File Path
  - 43xxx_Wi-Fi
    - WICED - \43xxx_Wi-Fi
    - apps - \43xxx_Wi-Fi
    - build - \43xxx_Wi-Fi
    - doc - \43xxx_Wi-Fi
    - include - \43xxx_Wi-Fi
    - libraries - \43xxx_Wi-Fi
    - platforms - \43xxx_Wi-Fi
    - resources - \43xxx_Wi-Fi
    - tools - \43xxx_Wi-Fi

Search for duplicate source files on the path

Using GDB (DSF) Hardware Debugging Launcher - Select other...

Filter matched 7 of 12 items


Debug Configurations

Create, manage, and run configurations

Name: 43xxx_Wi-Fi_Debug_Windows

Main | Debugger | Startup | Source | Common

Save as
- [✓ Local file]
- Shared file: \43xxx_Wi-Fi

Display in favorites menu
- [✓] Debug

Encoding
- [✓ Default - inherited (Cp1252)]
- Other ISO-8859-1

Standard Input and Output
- [✓] Allocate console (necessary for input)
- [ ] File:
- [ ] Append

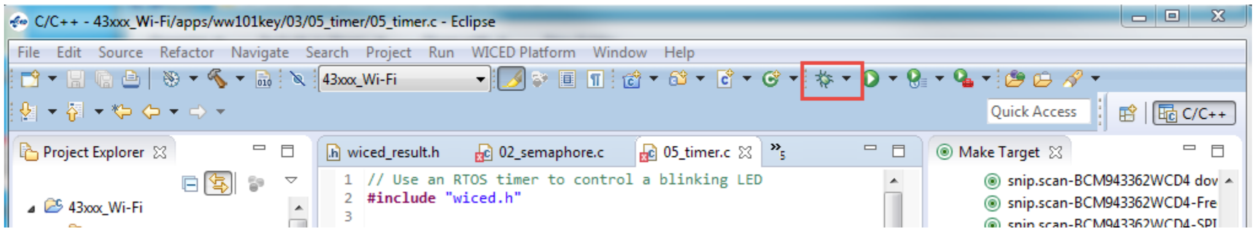
[✓] Launch in background

Using GDB (DSF) Hardware Debugging Launcher - Select other...

Filter matched 7 of 12 items

Running the Debugger

Once the setup is complete, execute the make target 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.


C/C++ - 43xxx_Wi-Fi/apps/ww101key/03/05_timer/05_timer.c - Eclipse
File Edit Source Refactor Navigate Search Project Run WICED Platform Window Help
43xxx_Wi-Fi
Quick Access
C/C++
Project Explorer
43xxx_Wi-Fi
wiced_result.h
02_semaphore.c
05_timer.c
Make Target
snip.scan-BCM943362WCD4 dov
snip.scan-BCM943362WCD4-Fre
snip.scan-BCM943362WCD4-SPI

```
// Use an RTOS timer to control a blinking LED
#include "wiced.h"
```

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 all “arm-none-eabi-gdb” processes.

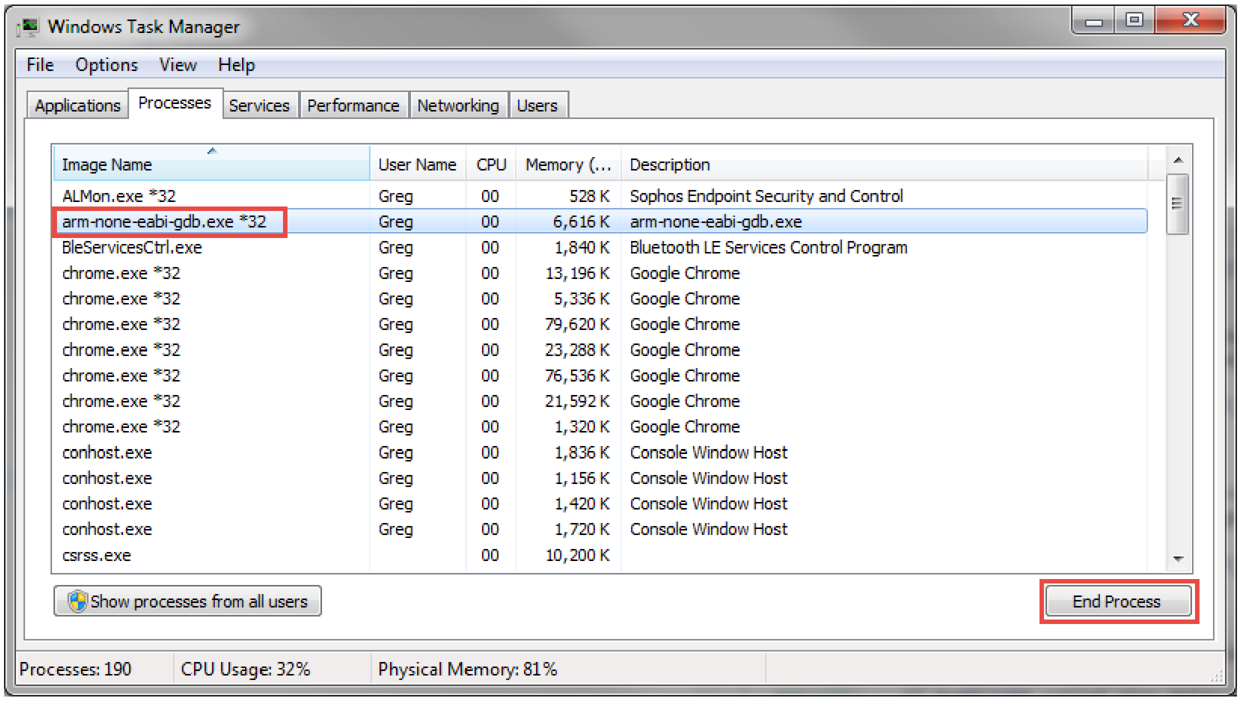
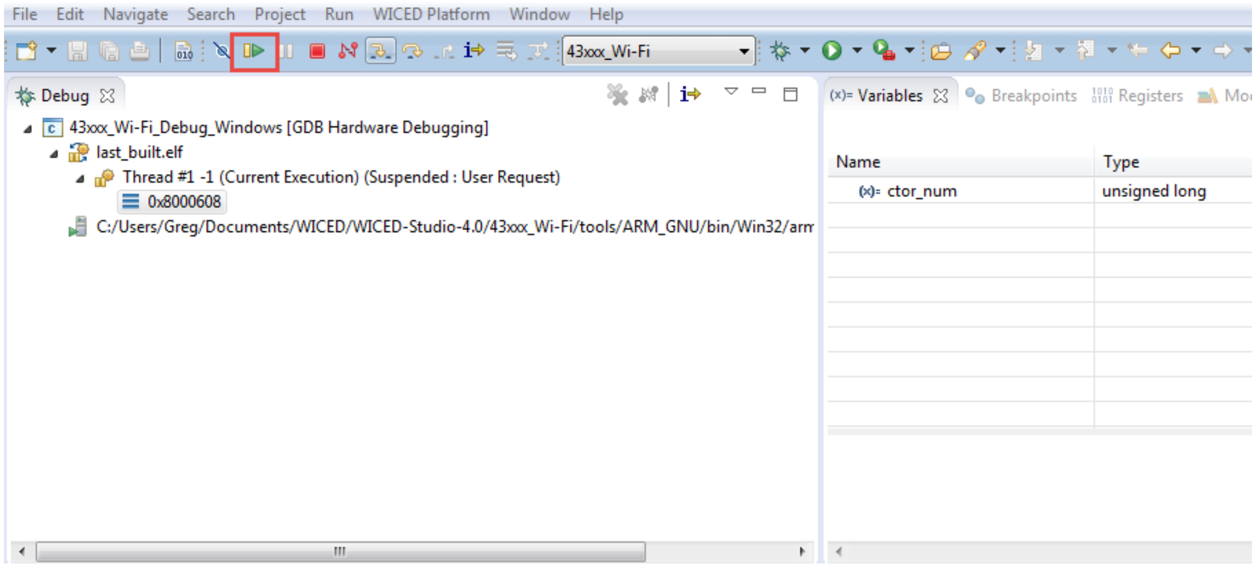


Image Name	User Name	CPU	Memory (...	Description
ALMon.exe *32	Greg	00	528 K	Sophos Endpoint Security and Control
arm-none-eabi-gdb.exe *32	Greg	00	6,616 K	arm-none-eabi-gdb.exe
BleServicesCtrl.exe	Greg	00	1,840 K	Bluetooth LE Services Control Program
chrome.exe *32	Greg	00	13,196 K	Google Chrome
chrome.exe *32	Greg	00	5,336 K	Google Chrome
chrome.exe *32	Greg	00	79,620 K	Google Chrome
chrome.exe *32	Greg	00	23,288 K	Google Chrome
chrome.exe *32	Greg	00	76,536 K	Google Chrome
chrome.exe *32	Greg	00	21,592 K	Google Chrome
chrome.exe *32	Greg	00	1,320 K	Google Chrome
conhost.exe	Greg	00	1,836 K	Console Window Host
conhost.exe	Greg	00	1,156 K	Console Window Host
conhost.exe	Greg	00	1,420 K	Console Window Host
conhost.exe	Greg	00	1,720 K	Console Window Host
csrss.exe		00	10,200 K	

Processes: 190 CPU Usage: 32% Physical Memory: 81%

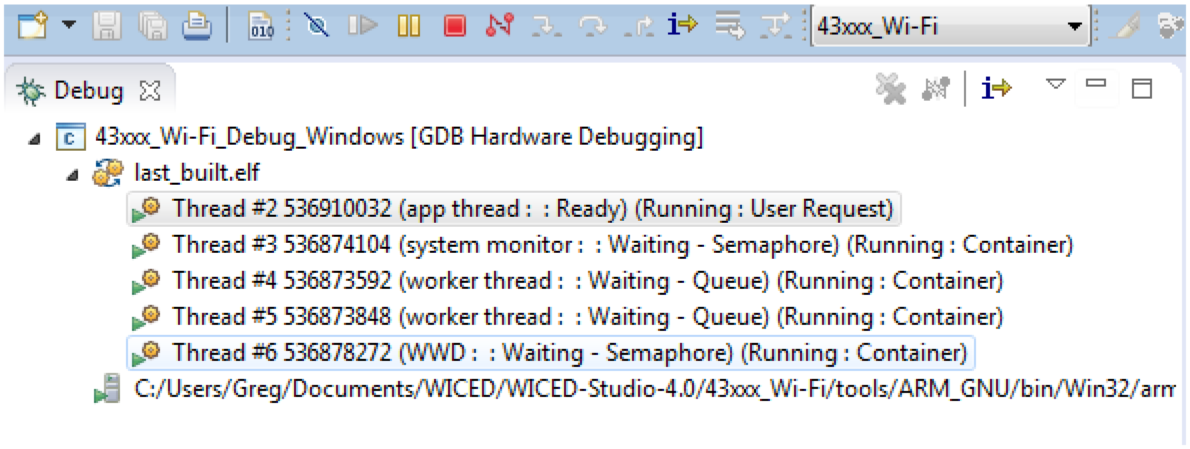
When the debugger starts the top banner will look like this:


File Edit Navigate Search Project Run WICED Platform Window Help
43xxx_Wi-Fi
Debug
Variables
Breakpoints
Registers
Name
Type

43xxx_Wi-Fi_Debug_Windows [GDB Hardware Debugging]
last_built.elf
Thread #1 -1 (Current Execution) (Suspended : User Request)
0x8000608
C:/Users/Greg/Documents/WICED/WICED-Studio-4.0/43xxx_Wi-Fi/tools/ARM_GNU/bin/Win32/arm

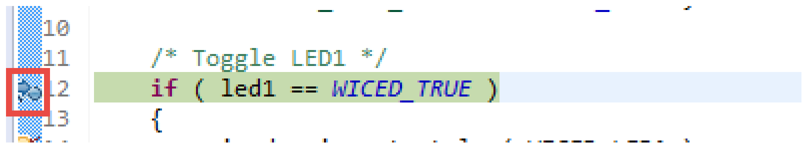
Name	Type
ctor_num	unsigned long

Click the “Resume” button a few times (shown in the figure above) until the program continues running and the resume button stays grey. Notice that additional threads along with information about them appears in the debug window.



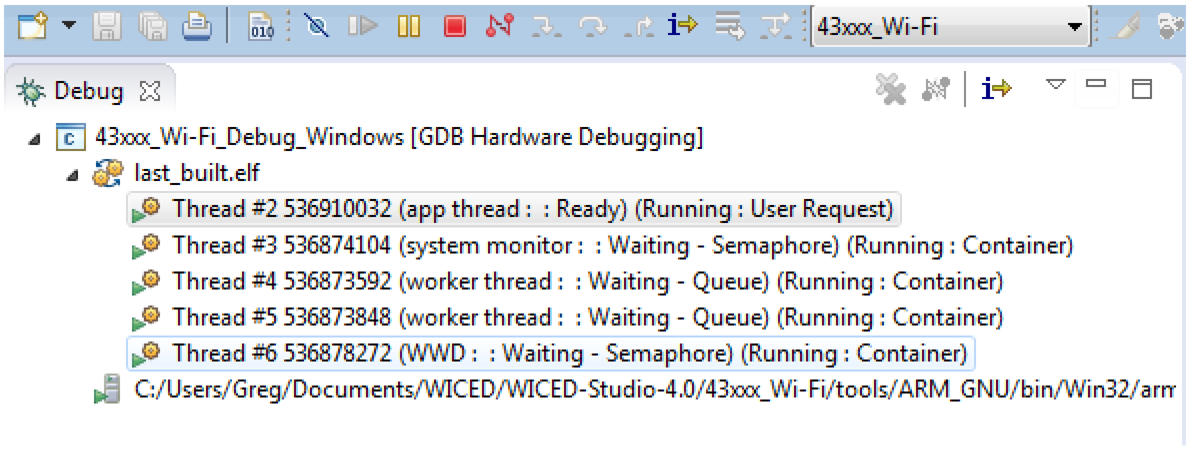
43xxx_Wi-Fi_Debug_Windows [GDB Hardware Debugging]
last_built.elf
Thread #2 536910032 (app thread : : Ready) (Running : User Request)
Thread #3 536874104 (system monitor : : Waiting - Semaphore) (Running : Container)
Thread #4 536873592 (worker thread : : Waiting - Queue) (Running : Container)
Thread #5 536873848 (worker thread : : Waiting - Queue) (Running : Container)
Thread #6 536878272 (WWD : : Waiting - Semaphore) (Running : Container)
C:/Users/Greg/Documents/WICED/WICED-Studio-4.0/43xxx_Wi-Fi/tools/ARM_GNU/bin/Win32/arm

You can toggle breakpoints by double clicking in the column to the left of the line numbers in the source code or you can right click and select "Toggle Breakpoint". The breakpoint symbol appears to the left of the line number as shown here.



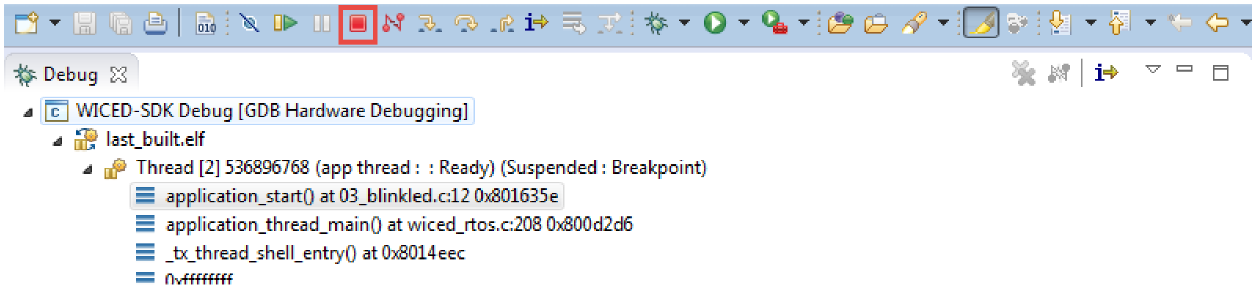
```
10
11      /* Toggle LED1 */
12      if ( led1 == WICED_TRUE )
13      {
```

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.



```
43xxx_Wi-Fi
Debug
43xxx_Wi-Fi_Debug_Windows [GDB Hardware Debugging]
  last_built.elf
    Thread #2 536910032 (app thread : : Ready) (Running : User Request)
    Thread #3 536874104 (system monitor : : Waiting - Semaphore) (Running : Container)
    Thread #4 536873592 (worker thread : : Waiting - Queue) (Running : Container)
    Thread #5 536873848 (worker thread : : Waiting - Queue) (Running : Container)
    Thread #6 536878272 (WWD : : Waiting - Semaphore) (Running : Container)
  C:/Users/Greg/Documents/WICED/WICED-Studio-4.0/43xxx_Wi-Fi/tools/ARM_GNU/bin/Win32/arm
```

Click the red "Terminate" button to stop debugging. Once you terminate the debugger, you may want to switch back to the C/C++ perspective by clicking on the button at the top right corner.



```
Debug
WICED-SDK Debug [GDB Hardware Debugging]
  last_built.elf
    Thread [2] 536896768 (app thread : : Ready) (Suspended : Breakpoint)
      application_start() at 03_blinkled.c:12 0x801635e
      application_thread_main() at wiced_rtos.c:208 0x800d2d6
      _tx_thread_shell_entry() at 0x8014eec
      0xffffffff
```

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?