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

## Objective

### An introduction to RTOS

For complex systems with many parallel things going on, an RTOS is critical. WICED devices are ALWAYS running an RTOS.

In an IoT device you might have:

1. One or more tasks that reads data from the pins (task = thread)
2. Tasks that manage the WiFi connection
   1. **ARP**
   2. **Read/Write Activity (data going to from socket)**
   3. **HTTP server**
   4. **DNS server**
   5. **DHCP server**
   6. **WICED manages that complexity for you – makes you a good citizen to the network**
3. One or more tasks that sends data to the cloud

An RTOS helps to simplify multiple parallel independent tasks like those. It manages things so that:

1. everyone gets a turn
2. things happen at the right time
3. shared resources are used properly

Two major schemes:

1. preemptive multitasking (CPU completely controls when a task runs and can stop/start)
2. cooperative multitasking – each thread needs to yield control

The WICED RTOS’s are cooperative so your tasks need to play nice

### WICED RTOS Abstraction Layer

WICED Studio supports ThreadX and FreeRTOS.

ThreadX:

1. built into the device ROM
2. license is included if you are using WICED chips

Built in abstraction layer means they both look the same to the user

* Can also use low level RTOS functions for more complex things (not covered in this class)

Show RTOS documentation (Components > RTOS)

To use FreeRTOS, add “-FreeRTOS-LwIP” to make target after platform with no space (show example)

### Problems with RTOSs

Potential problems with RTOS:

1. Deadlocks (A waiting for B, B waiting for C, C waiting for A)
2. resource conflicts with shared resources (I2C, UART, LED)
3. inter-process communication

To solve these issues, we use:

1. Semaphores
2. Queues
3. Mutexes
4. Timers

Note that the first three are ways for a thread to yield control

The process to use each is basically:

1. Create a (global) data structure
2. Initialize – always init before using. That means init before starting threads that use it.
3. Access the Data Structure (set, get, lock, unlock, push, pop)
4. Deinit (maybe)

### Threads

Creating a thread

Thread function looks just like the main application\_start function - typically has an infinite loop – it will keep running its task over and over. Note that application\_start is really a thread.

Point out that the delay causes thread to yield. Should include unless some other mechanism is used.

* Delay of 10ms means at least 10ms. Might be more.

Usually have while(1) loop but don’t have to have it in a thread (or even application\_start)

### Semaphore

like a ship’s signal flag

one (or more) threads can set the flag, one (or more) threads can get the flag

counting semaphore:

set will increment flag

get will decrement flag

if count = 0 the get will suspend thread until someone sets it or until an optional timeout

use WICED\_WAIT\_FOREVER if you don’t want a timeout

**Can’t call any function from an ISR that doesn’t immediately return –** use WICED\_NO\_WAIT for get

### Mutex

Mutual exclusion

Use when 2 resources want to access the same shared resource (e.g. memory, I2C, UART, LEDs, etc.)

Lock > Use Resource > Unlock

If someone else has already locked the mutex, the thread will suspend until it is unlocked

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

### Queue

Like a Semaphore, but with a message passed

Use Push to Queue and Pop from Queue

The message in the queue must be a multiple of 4 bytes

The queue can be as many messages deep as you want

During a pop, if nothing is in the queue, the thread will suspend until there is (or until a timeout)

Same comment about ISR – must use WICED\_NO\_WAIT to use pop inside an ISR

### Timer

A timer is just a function that executes at a specified interval

Unlike a thread, the timer function should NOT have a while(1) loop – it will be called again each time the timer expires.

## Exercise(s)

Exercises – 90 minutes

Debugger is exercise 06