INTERNET OF THINGS | COEN 243
DEPARTMENT OF COMPUTER ENGINEERING, SANTA CLARA UNIVERSITY, CALIFORNIA
BEHNAM DEZFOULI, PHD

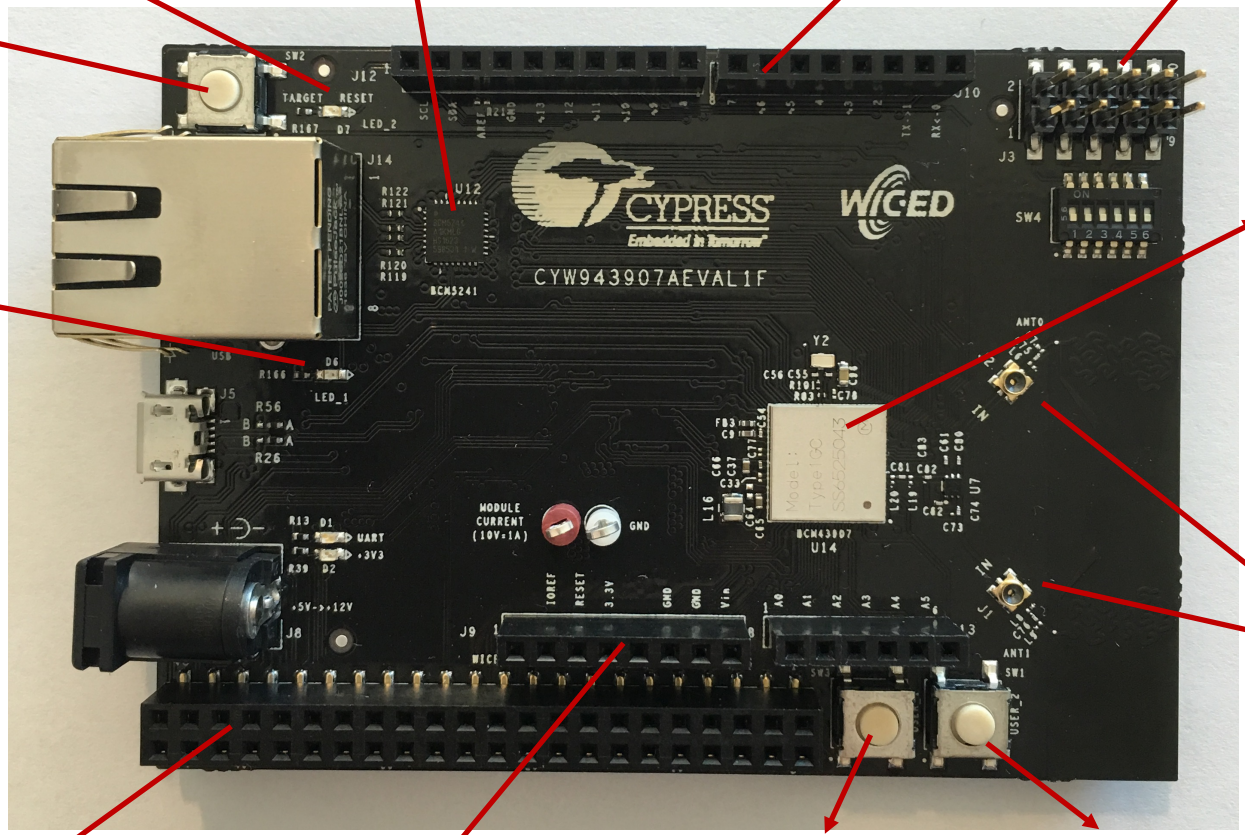
Internet of Things

Hands-On: Hardware and Operating System

Hardware Platform

Hardware Platform

Cypress CYW943907



LED 2

Reset

LED 1

BCM5241
Ethernet transceiver

Arduino headers

External JTAG Header

Cypress
CYW43907

WiFi antenna

WICED header

Arduino headers

User Switch 1 SW3

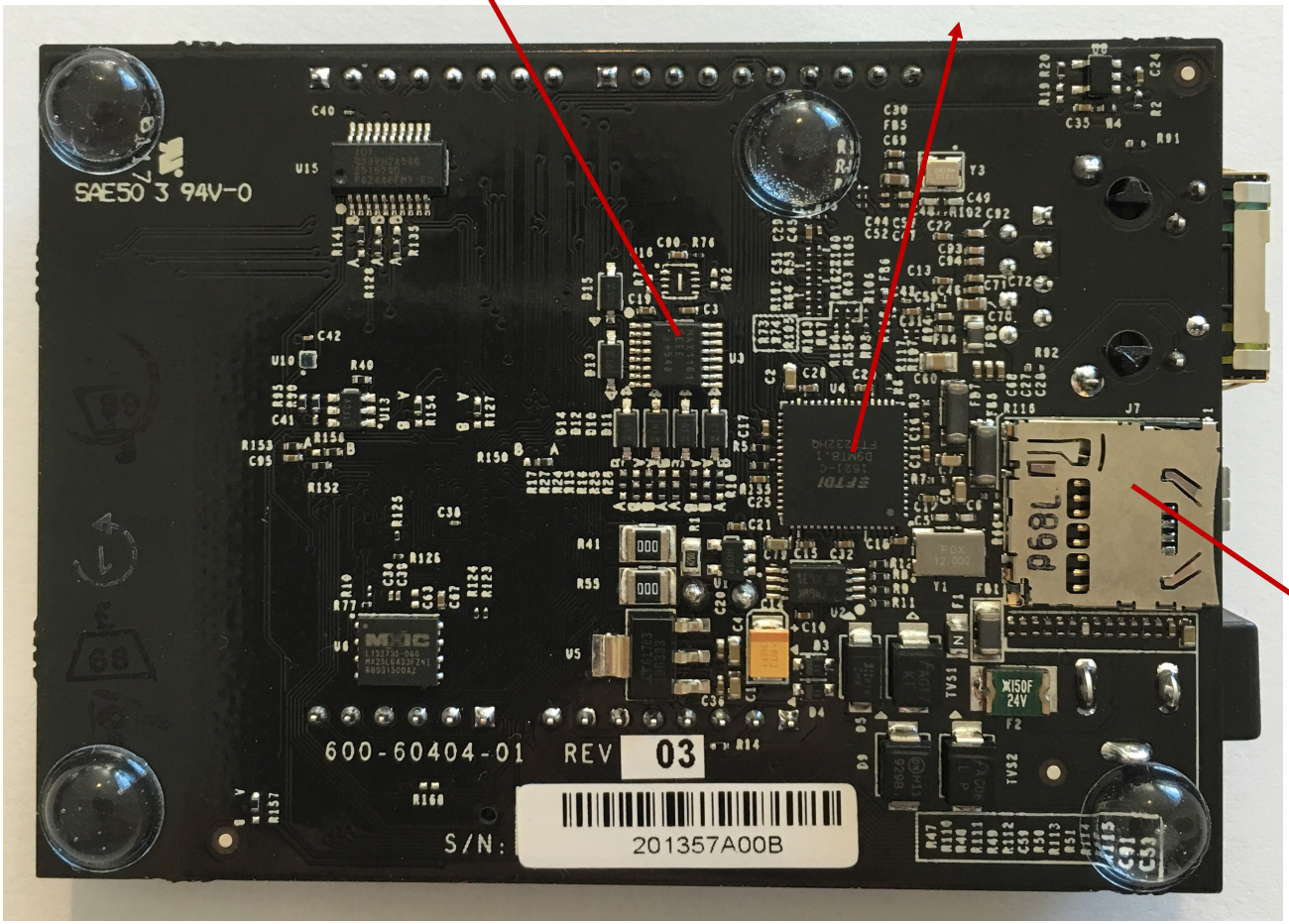
User Switch 2 SW1

Hardware Platform

Cypress CYW943907

MAX11615 ADC
Helps to achieve Analog functionality on the Arduino headers

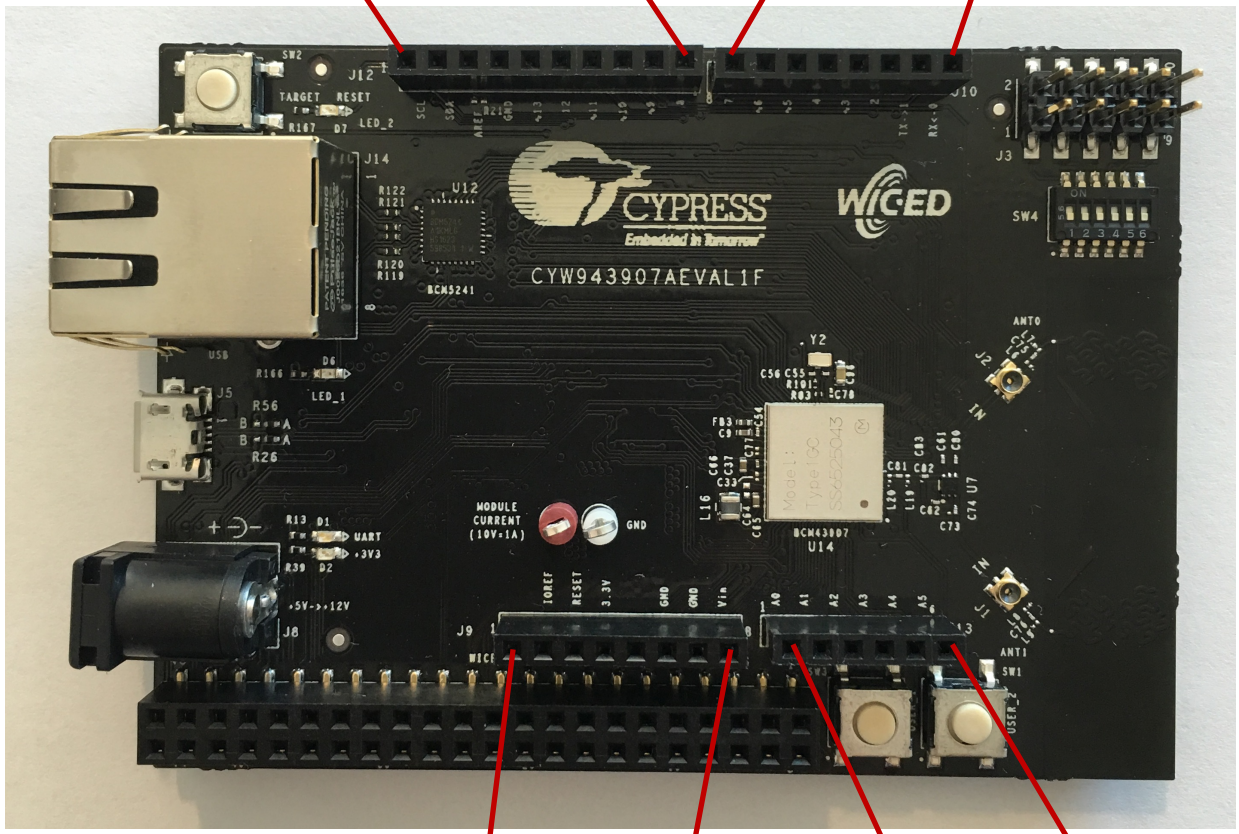
- **FT2232H is FTDI's 5th generation of USB devices**
- The FT2232H is a USB 2.0 Hi-Speed (480Mb/s) to UART/FIFO IC



SD Card

Hardware Platform

Cypress CYW943907



Cypress CYW943907

Eval Board Header	CYW43907 Pin Name/ Kit Signal Name	ARDUINO Header Name	WICED Enumeration	Alternate Enumeration
J10.1	GPIO_0	D0	WICED_GPIO_1	N/A
J10.2	GPIO_1	D1	WICED_GPIO_2	N/A
J10.3	GPIO_13	D2	WICED_GPIO_9	N/A
J10.4	GPIO_7	D3	WICED_GPIO_3	WICED_PWM_6
J10.5	GPIO_14	D4	WICED_GPIO_10	N/A
J10.6	GPIO_16	D5	WICED_GPIO_12	WICED_PWM_3
J10.7	GPIO_15	D6	WICED_GPIO_11	WICED_PWM_4
J10.8	I2S0_SD_IN	D7	WICED_GPIO_31	WICED_I2S_1
J12.1	I2S1_SD_IN	D8	WICED_GPIO_36	WICED_I2S_3
J12.2	PWM_4	D9	WICED_GPIO_17	WICED_PWM_5
J12.3	GPIO_11	D10	WICED_GPIO_7	WICED_PWM_2
J12.4	GPIO_10	D11	WICED_GPIO_6	WICED_PWM_1
J12.5	GPIO_12	D12	WICED_GPIO_8	N/A
J12.6	GPIO_9	D13	WICED_GPIO_5	WICED_LED2
J12.7	GND	GND	N/A	N/A
J12.8	ARD_AREF	AREF	N/A	N/A

Hardware Platform

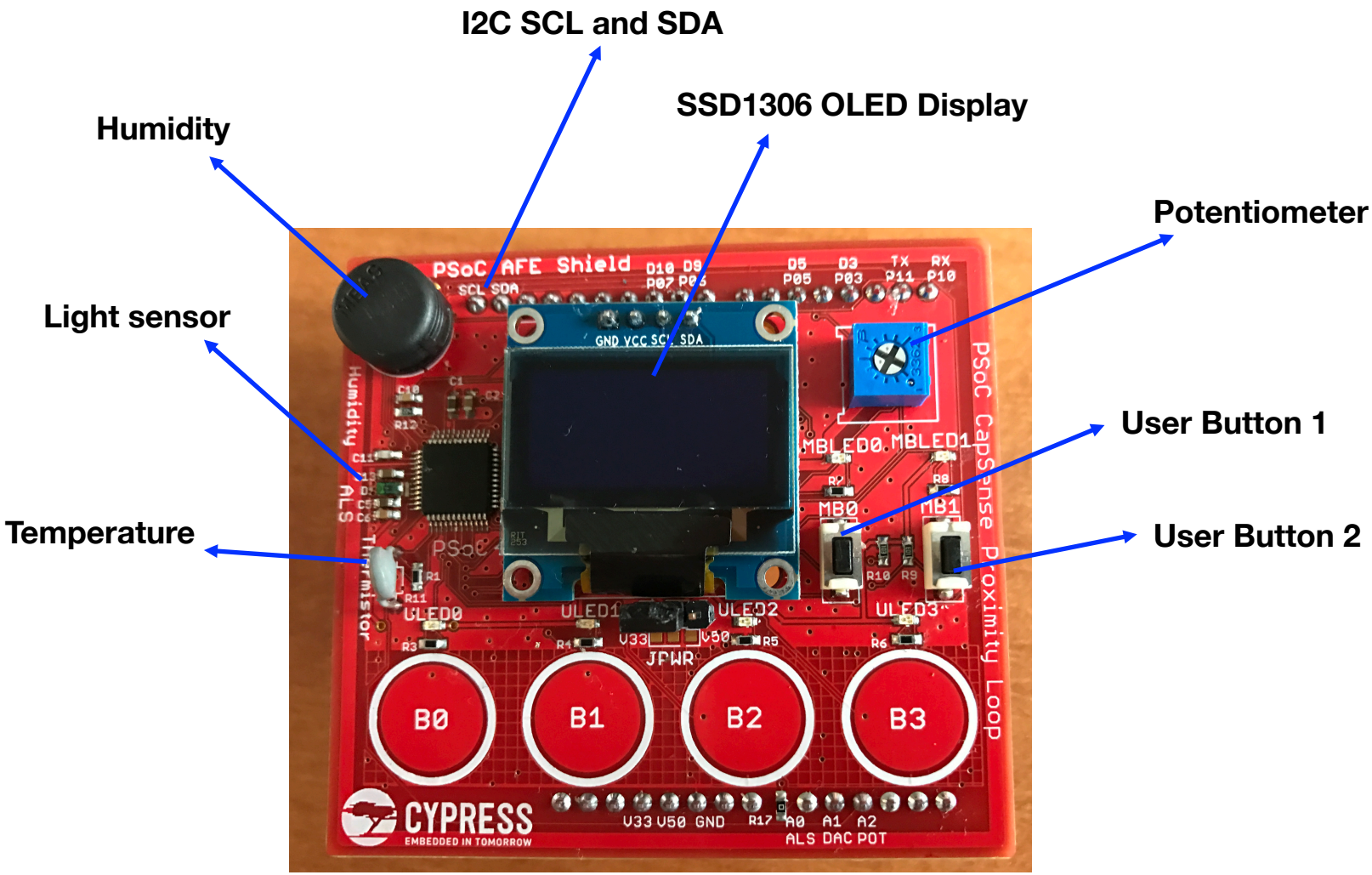
Cypress CYW943907

Eval Board Header	CYW43907 Pin Name/ Kit Signal Name	ARDUINO Header Name	WICED Enumeration	Alternate Enumeration
J12.9	I2C_1_SDA	SDA	WICED_GPIO_50	WICED_I2C_2
J12.10	I2C_1_SCL	SCL	WICED_GPIO_51	WICED_I2C_2
J13.1	ARD_AD0	A0	N/A	N/A
J13.2	ARD_AD1	A1	N/A	N/A
J13.3	ARD_AD2	A2	N/A	N/A
J13.4	ARD_AD3	A3	N/A	N/A
J13.5	ARD_AD4_SDA	A4	N/A	N/A
J13.6	ARD_AD5_SCL	A5	N/A	N/A
J9.1	NC	NC	N/A	N/A
J9.2	ARD_IOREF	IOREF	N/A	N/A
J9.3	ARD_RESET	RESET	N/A	N/A
J9.4	3V3	3.3V	N/A	N/A
J9.5	NC	5V	N/A	N/A
J9.6	GND	GND	N/A	N/A
J9.7	GND	GND	N/A	N/A
J9.8	VIN_EXT	VIN	N/A	N/A

Hardware Platform

Raspberry Pi

Cypress CYW943907



Software Development using WICED® Studio

Installation

- The WICED software tool is called "**WICED Studio**" and it is based on Eclipse
- Download the SDK

  http://www.cypress.com/products/wiced-software

- The first time you open WICED Studio, you will be asked for which platform you want to use
  - We will use **43xxx_Wi-Fi** for this class
  - You can change this setting later

- For MAC OS, install the FTDI driver: https://cdn.sparkfun.com/assets/learn_tutorials/7/4/FTDIUSBSerialDriver_v2_3.dmg

SDK Structure

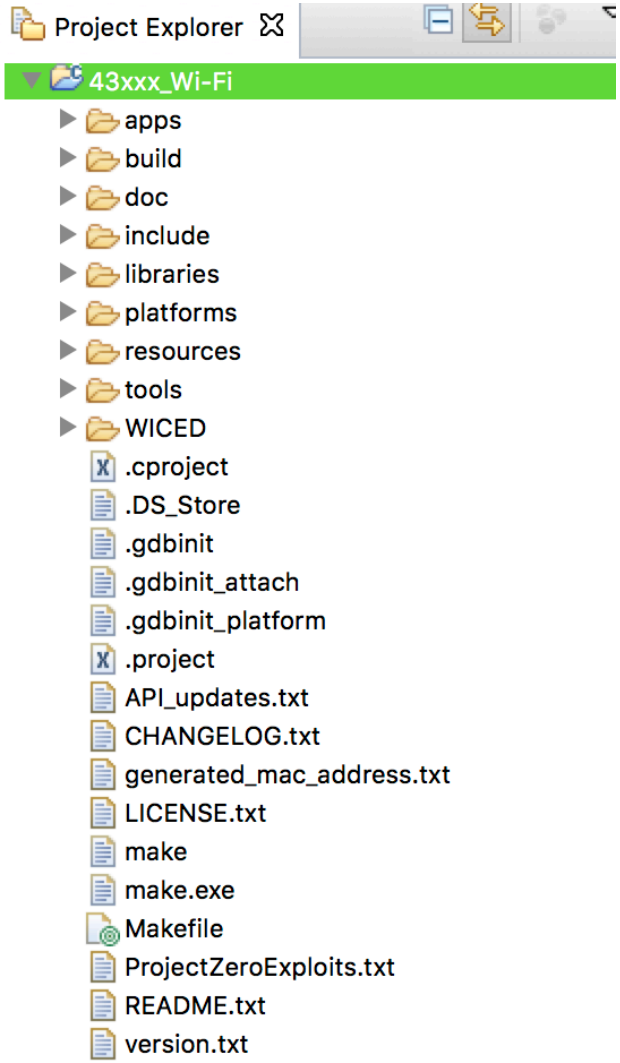
- WICED Studio supports multiple RTOSs
- ThreadX (by Express Logic) is built into the device ROM and the license is included for anyone using WICED chips so that is by far the best choice
- It also supports FreeRTOS
- In order to simplify using multiple RTOSs, **the WICED SDK has a built in abstraction layer that provides a unified interface to the fundamental RTOS function**

- Find the documentation for the WICED RTOS APIs under:

  Cypress WICED API Reference -> Components -> RTOS

- The WICED RTOSs provide important OS mechanisms such as mutexes, semaphores, queues and timers

SDK Structure



SDK Structure

`apps`: where all of the **example projects** reside, as well as where you will put your own projects

- **`snip`**: These are short examples that typically demonstrate one feature
  - For example:
    - **`snip/gpio`** demonstrates GPIO use by reading buttons and blinking LEDs
    - **`snip/scan`** scans for Wi-Fi access points every 5 seconds and displays the results to a terminal window
- **`demo`**: These are more complex and complete demonstrations
  - For example:
    - **`demo/temp_control`** demonstrates an application for controlling and reporting temperatures
    - **`demo/bt_smartbridge`** demonstrates a Bluetooth to Wi-Fi bridge
- **`test`**: These are test and utility programs such as a console that allows you to scan for and connect to Wi-Fi access points
  - For example:
    - **`test/console`** provides a console application on a terminal window Type “help” in the console for a list of all supported commands

SDK Structure

`doc`: The doc folder contains the documentation for the SDK Workspace

- Of particular interest is the API html file which documents all of the WICED API functions
- Also contains other useful documents such as the QSG (Quick Start Guide), how to use DCT (Device Configuration Tables), FLAC (Free Lossless Audio Compression), and OTA (Over the Air) updates

  https://community.cypress.com/welcome

`platform`: contains information on different kits (i.e. hardware platforms)

- These files are necessary in order to program a given project into specific hardware
- In our case, the kit we are using is called **CYW943907AEVAL1F**
- This kit has a platform folder, but since we are also using a shield attached to it, we will use a custom set of platform files (**CYW943907AEVAL1F_WW101**) that also includes the peripherals on the shield
- You will have to copy over the custom platform files before using the shield and kit
- You can even create platform files for your own custom hardware that you design

SDK Structure

`libraries`: contains various sets of library function files

- **`audio`:** Contains support for Apollo (a streaming audio standard), and codecs including Free Lossless Audio compression.
- **`crypto`:** Elliptic curve cryptography (ECC)
- **`daemons`:** Contains some typical "Unix" daemons to provide networking support including an HTTP Server, TFTP, DHCP, DNS etc.
- **`drivers`:** Contains hardware support files for SPI flash, USB etc.
- **`filesystems`:** FAT, FILE and other file systems that could be written to an SPI flash
- **`graphics`:** Support for the U8G OLED displays
- **`inputs`:** Drivers for buttons and GPIOs
- **`protocols`**: Support for application layer protocols including HTTP, COAP, MQTT etc.
- **`test`**: Tools to test network performance, iPerf, malloc, TraceX, audio
- **`utilities`**: Support for JSON, console, printf, buffers, etc.

`resources`: where you store files that are required by your application

- For example, if your application contains a web server, the html files for the server would be in the resources folder under `apps/https_server`

SDK Structure

- We are using a baseboard kit along with an analog front end shield which contains a PSoC analog co-processor chip
- In order to make it easier to interface with the shield, a set of platform files has been created
- Since this is not installed by default in the SDK we need to copy the platform folder into the SDK Workspace
- Copy the entire folder "`CYW943907AEVAL1F_WW101`" from the class materials into the "platforms" directory in the SDK Workspace

- Two key files here are `platform.c` and `platform.h`
- `platform.h` contains `#define` and type definitions used to set up and access the various kit and shield peripherals

SDK Structure

- For example, the **bases board** contains two LEDs and two mechanical buttons

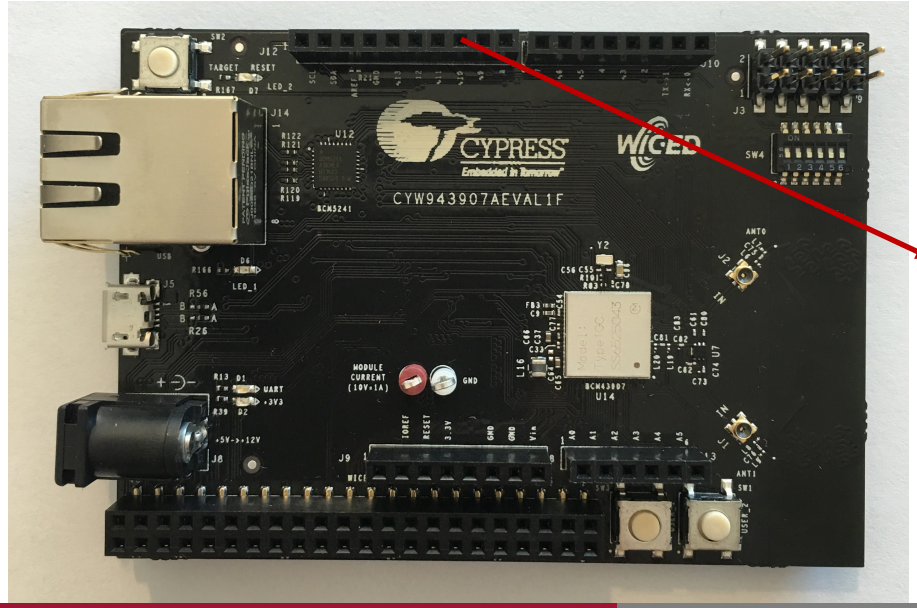
```
/* LEDs and buttons on the base board */
#define WICED_LED1          ( WICED_GPIO_16 )
#define WICED_LED2          ( WICED_GPIO_5 )
#define WICED_BUTTON1       ( WICED_GPIO_18 )
#define WICED_BUTTON2       ( WICED_GPIO_4 )
```

Switch	CYW43907 Pin Name	WICED_ENUM_ID	Alternate Enumeration in WICED
LED_1	PWM_3	WICED_GPIO_16	WICED_LED1
LED_2	GPIO_9	WICED_GPIO_5	WICED_LED2

SDK Structure

- The **shield board** contains two LEDs and two mechanical buttons

```
/* LEDs and buttons on the shield */
#define WICED_SH_LED0                   ( WICED_GPIO_7 )
#define WICED_SH_LED1                   ( WICED_GPIO_17 )
#define WICED_SH_LED0_ON_STATE          ( WICED_ACTIVE_HIGH )
#define WICED_SH_LED1_ON_STATE          ( WICED_ACTIVE_HIGH )
#define WICED_SH_MB0                    ( WICED_GPIO_12 )
#define WICED_SH_MB1                    ( WICED_GPIO_3 )
```




```
J12.3 = WICED_SH_LED0 = WICED_GPIO_7 =
D10 (Arduino) = GPIO_11 (CYW43907)
```

SDK Structure

- The `platform.c` file contains several constant arrays and structures that are used to configure the peripherals
- This files maps the chip pins to WICED pins
- This file also contains the functions used to initialize and control the peripherals

❑For example, the LED pins are initialized as outputs and the button pins are initialized as inputs with a resistive pullup

SDK Structure

- A WICED Studio project can be located anywhere within the `apps` folder of the SDK Workspace
- **The key parts of a project are:**
  - A folder with the name of the project
  - A C source file called `<project>.c` inside the project folder
  - A makefile called `<project>.mk` inside the project folder
- Note: The `<project>` name must be the same for the folder name, C file name, and makefile

- **The makefile contains:**
  - The application name (any unique string)
  - The list of all source files (including `<project>.c`)
  - A list of valid and/or invalid platforms for the given project
  - Makefile macros to provide access to libraries
  - Other resources such as images, web pages, etc.

SDK Structure

- In order to download the project to your board, you will need to create a new make target of the form:

  `<folder1>.[<folder2>...].<project>-<platform> download run`

  - `<folder1>` is the name of the folder below the apps folder
  - `<folder2>, <folder3>`, etc., are the rest of the path down to the project name; use a period to separate the folder names
  - `<project>` is the name of the project
    - The folder, main C file, and makefile must all have the same name
  - `<platform>` is the name of the hardware platform
    - Example: CYW943907AEVAL1F_WW101
    - There must be an entry in the platforms directory that matches the name provided here

SDK Structure

- For example, if we create a folder called "`scu_iot_projects`" for our class projects and a subfolder called "`os`", and call the first project "`blinkled`", the build target for our board would be:

  ```
  scu_iot_projects.os.blinkled-CYW943907AEVAL1F download run
  ```

- The make targets that are defined can be seen in the "Make Target" window along the right side of WIKED Studio
- Expand "43xxx_Wi-Fi" to see the existing make targets

- To create a new make target:
  - Right click on an existing make target that is similar to what you want to create and select *New...*
  - Delete "*Copy of* " (don't forget to remove the space!) and change the name as necessary for your new make target

SDK Structure

- You must `#include "wiced.h"` at the top of the main C file
- You must also call the `wiced_init()` function in the initialization section of the main C file
  - This function **does all of the initialization** required to get the other WICED APIs to work properly and calls the functions that initialize the peripherals for the kit

GPIO

- As explained previously, GPIOs must be initialized before they are used
- The IOs on the kit that are connected to specific peripherals such as LEDs and buttons **are often automatically initialized for you as part of the platform files** (`platform.h` and `platform.c`)
- Once initialized:
  - outputs can be driven using `wiced_gpio_output_high()` *and* `wiced_gpio_output_low()`
  - input pins can be read using `wiced_gpio_input_get()`

- The parameter for these functions is the WICED pin name such as `WICED_GPIO_1` or a peripheral name for your platform such as `WICED_LED1`

GPIO

- In the next example, we use **GPIO** and **`wiced_rtos_delay_milliseconds`**

```
wiced_result_t  wiced_rtos_delay_milliseconds (uint32_t  milliseconds)
```

- **Sleep for a given period of milliseconds**
- Causes the **current thread** to sleep for **at least** the specified number of milliseconds (**why "at least"?**)
  - param[in] `milliseconds`: The time to sleep in milliseconds
  - return `WICED_SUCCESS`: on success
  - return `WICED_ERROR`: if an error occurred

❑ **Example**: The next example shows how we blink LED1 on the base board and shield with a frequency of 2 Hz

Example: Blink LED (`blinkled`)
Part: 1/1

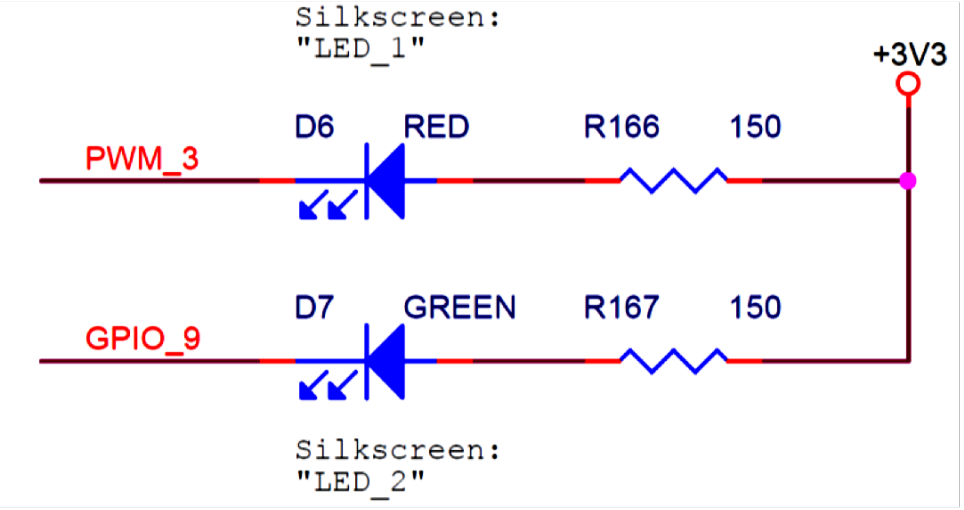
```
#include "wiced.h"
void application_start( )
{
    wiced_init();       /* Initialize the WICED device */

    /* The LED is initialized in platform.c.
     * If it was not, you would need the following:
     * wiced_gpio_init(WICED_SH_LED1, OUTPUT_PUSH_PULL); */

    while ( 1 )
    {
        /* LED on */
        wiced_gpio_output_low( WICED_LED1 );
        wiced_gpio_output_low( WICED_LED2 );
        wiced_gpio_output_low( WICED_SH_LED1 );
        wiced_rtos_delay_milliseconds( 250 );
        /* LED off */
        wiced_gpio_output_high( WICED_LED1 );
        wiced_gpio_output_high( WICED_LED2 );
        wiced_gpio_output_high( WICED_SH_LED1 );
        wiced_rtos_delay_milliseconds( 250 );
    }
}
```

Toggle mainboard and shield LEDs

GPIO

- Note that the **onboard LEDs** are connected as follows:

Switch	CYW43907 Pin Name	WICED_ENUM_ID	Alternate Enumeration in WICED
LED_1	PWM_3	WICED_GPIO_16	WICED_LED1
LED_2	GPIO_9	WICED_GPIO_5	WICED_LED2



Debug Printing

- The SDK includes **built-in debug print functions** which can be used to display messages via the USB-UART Bridge built into the kit
- The file "`wwd_debug.h`" defines all of the different message types
- We will use one called "`WPRINT_APP_INFO`" which is meant for printing application information
- This is a macro that uses standard `printf()` formatting
- It is enabled by default in the SDK ("`wiced_defaults.h`")

❑ For example, to print a variable called "test"

```
WPRINT_APP_INFO( ("The value of test is: %d\n", test) );
```

- Note: The extra set of parenthesis is required due to the way the macro is defined

- Use a serial port tool (e.g., SerialTools on MAC OS) with baudrate 115200 bps

GPIO Interrupts

- GPIO interrupts are controlled using:
  - `wiced_gpio_input_irq_enable()`
  - `wiced_gpio_input_irq_disable()`

```
wiced_result_t wiced_gpio_input_irq_enable
        ( wiced_gpio_t gpio, wiced_gpio_irq_trigger_t trigger,
                wiced_gpio_irq_handler_t handler, void* arg )
```

Enables an interrupt trigger for an input GPIO pin
Using this function on an *uninitialized gpio pin* or a *gpio pin which is set to output* mode is undefined.

- param[in] **`gpio`**: The gpio pin which will provide the interrupt trigger
- param[in] **`trigger`**: The type of trigger (rising/falling edge, high/low level)
- param[in] **`handler`**: A function pointer to the interrupt handler
- param[in] **`arg`**: An argument that will be passed to the interrupt handler
- return **`WICED_SUCCESS`**: on success
- return **`WICED_ERROR`**: if an error occurred with any step

Software Development using WICED Studio

Example: Interrupt-LED (`interrupt_led`)
Part: 1/1

```
#include "wiced.h"
/* Interrupt service routine for the button */
void button_isr(void* arg)
{
        static wiced_bool_t led1 = WICED_FALSE;
        if ( led1 == WICED_TRUE ) {
                wiced_gpio_output_low( WICED_LED1 );
                led1 = WICED_FALSE;
        }
        else {
                wiced_gpio_output_high( WICED_LED1 );
                led1 = WICED_TRUE;
        }
}

void application_start( ) // Main application
{
    wiced_init();/* Initialize the WICED device */

    wiced_gpio_input_irq_enable(WICED_BUTTON2, IRQ_TRIGGER_FALLING_EDGE,
                                button_isr, NULL); /* Setup interrupt */

    while ( 1 ) {
        /* No main loop code required - just wait for interrupt */
    }
}
```

set up a **falling edge interrupt** for the GPIO connected to the button

GPIO Interrupts

- The onboard user buttons are as follows

Switch	CYW43907 Pin Name	WICED_ENUM_ID	Alternate Enumeration in WICED
USER_1 (SW3)	PWM_5	WICED_GPIO_18	WICED_BUTTON1
USER_2 (SW1)	GPIO_8	WICED_GPIO_4	WICED_BUTTON2

- The chip itesels provides pull-up resistor
- So the board does not require to use any pull-up

UART

❑ **Example:** We are interested to implement the following application:

- **The number of times the button has been pressed is sent out over the UART interface whenever the button is pressed**
- For simplicity, just count from 0 to 9 and then wrap back to 0 so that you only have to send a single character each time
- We want to setup a UART configuration structure for a baud rate of 115200, data width 8, no parity, 1 stop bit, and no flow control

- **In order to modify UART configuration, we need to avoid automatic UART configuration by WICED**
- To this end, we need to add the following like to the make file

```
GLOBAL_DEFINES := WICED_DISABLE_STDIO
```

UART

```
/** Initializes a UART interface
 * Prepares an UART hardware interface for communications
 * @param[in]  uart  : The interface which should be initialized
 * @param[in]  config: UART configuration structure defined in
 *                     WICED/platform/include/platform_peripheral.h
 * @param[in] optional_rx_buffer : Pointer to an optional RX ring buffer
 * @return    WICED_SUCCESS : on success.
 * @return    WICED_ERROR   : if an error occurred with any step

wiced_result_t wiced_uart_init( wiced_uart_t uart,
                  const wiced_uart_config_t* config,
                            wiced_ring_buffer_t* optional_rx_buffer );
```

```
/** Transmit data on a UART interface
 * @param[in] uart : The UART interface
 * @param[in]  data     : Pointer to the start of data
 * @param[in]  size     : Number of bytes to transmit
 * @return    WICED_SUCCESS : on success.
 * @return    WICED_ERROR   : if an error occurred with any step

wiced_result_t wiced_uart_transmit_bytes( wiced_uart_t uart, const void* data,
                                                        uint32_t size );
```

Software Development using WICED Studio

Example: UART-Send (`uartsend`)
Part: 1/2

```
#include "wiced.h"
volatile wiced_bool_t newPress = WICED_FALSE;

void button_isr(void* arg) // Interrupt service routine for the button
{
        static wiced_bool_t led1 = WICED_FALSE;

        /* Toggle LED1 */
        if ( led1 == WICED_TRUE ) {
                wiced_gpio_output_low( WICED_LED1 );
                led1 = WICED_FALSE;
        }
        else {
                wiced_gpio_output_high( WICED_LED1 );
                led1 = WICED_TRUE;
        }
        newPress = WICED_TRUE; /* Need to send new button count value */
}

/* Main application */
void application_start( )
{
    uint8_t pressCount = 0;
    char    printChar;

    wiced_init();/* Initialize the WICED device */
```

Software Development using WICED Studio

```
 wiced_gpio_input_irq_enable(WICED_SH_MB1, IRQ_TRIGGER_FALLING_EDGE,
                                    button_isr, NULL); /* Setup interrupt */

    /* Configure and start the UART */
    wiced_uart_config_t uart_config =
    {
        .baud_rate    = 115200,
        .data_width   = DATA_WIDTH_8BIT,
        .parity       = NO_PARITY,
        .stop_bits    = STOP_BITS_1,
        .flow_control = FLOW_CONTROL_DISABLED,
    };

    wiced_uart_init(WICED_UART_1, &uart_config, NULL); /* Setup UART */

    while ( 1 ) {
        if(newPress) {
                pressCount ++; /* Increment counter */
                if(pressCount > 9) pressCount = 0;

                printChar = pressCount + '0';

                wiced_uart_transmit_bytes(WICED_UART_1, &printChar , 1);

                newPress = WICED_FALSE;   /* Reset for next press */
        }
    }
}
```

For the configuration to work, we need to use `GLOBAL_DEFINES := WICED_DISABLE_STDIO` in the makefile

We use NULL for the receive buffer since we will only be transmitting values

Alternatively, we could write `pressCount + 0x30` for integer to ASCII conversion

Example: UART-Send (`uartsend`)
Part: 2/2

UART

❑ **Example:** We are interested to implement the following application:

- **Receive characters from the UART interface**
- **0 turns the LED off, 1 turns the LED on**

Initialize a ring buffer

```
 * @param[out] ring_buffer: Pointer to the ring buffer structure to be
 * initialized.
 * @param[in]  buffer      : Pointer to the buffer to use for the ring buffer.
 * @param[in]  buffer_size : Size of the buffer
 * (maximum buffer_size - 1 bytes will be stored).
wiced_result_t ring_buffer_init ( wiced_ring_buffer_t* ring_buffer,
                                  uint8_t* buffer, uint32_t buffer_size );
```

Receive data on a UART interface

```
 * @param[in] uart: The UART interface
 * @param[out]  data : Pointer to the buffer which will store incoming data
 * @param[in,out] size : Number of bytes to receive, function return in same
 * parameter number of actually received bytes
 * @param[in] timeout : Timeout in milliseconds WICED_WAIT_FOREVER and
 * WICED_NO_WAIT can be specified for infinite and no wait.
 * @return    WICED_SUCCESS : on success.
 * @return    WICED_ERROR   : if an error occurred with any step
wiced_result_t wiced_uart_receive_bytes( wiced_uart_t uart, void* data,
                                         uint32_t* size, uint32_t timeout );
```

UART

Example: UART-Receive (`uartreceive`)
Part: 1/2

```
#include "wiced.h"
/* Main application */
void application_start( )
{
    char    receiveChar;
    uint32_t  expected_data_size = 1;

    wiced_init(); /* Initialize the WICED device */

    /* Configure and start the UART. */
    #define  RX_BUFFER_SIZE (5)
    wiced_ring_buffer_t  rx_buffer;
    uint8_t  rx_data[RX_BUFFER_SIZE]; // A 5 byte circular array

    // Initialize ring buffer to hold receive data
    ring_buffer_init(&rx_buffer, rx_data, RX_BUFFER_SIZE );

    wiced_uart_config_t uart_config =
    {
        .baud_rate    = 115200,
        .data_width   = DATA_WIDTH_8BIT,
        .parity       = NO_PARITY,
        .stop_bits    = STOP_BITS_1,
        .flow_control = FLOW_CONTROL_DISABLED,
    };
    wiced_uart_init( WICED_UART_1, &uart_config, &rx_buffer); /* Setup UART */
```

For the configuration to work, we need to use `GLOBAL_DEFINES := WICED_DISABLE_STDIO` in the makefile

UART

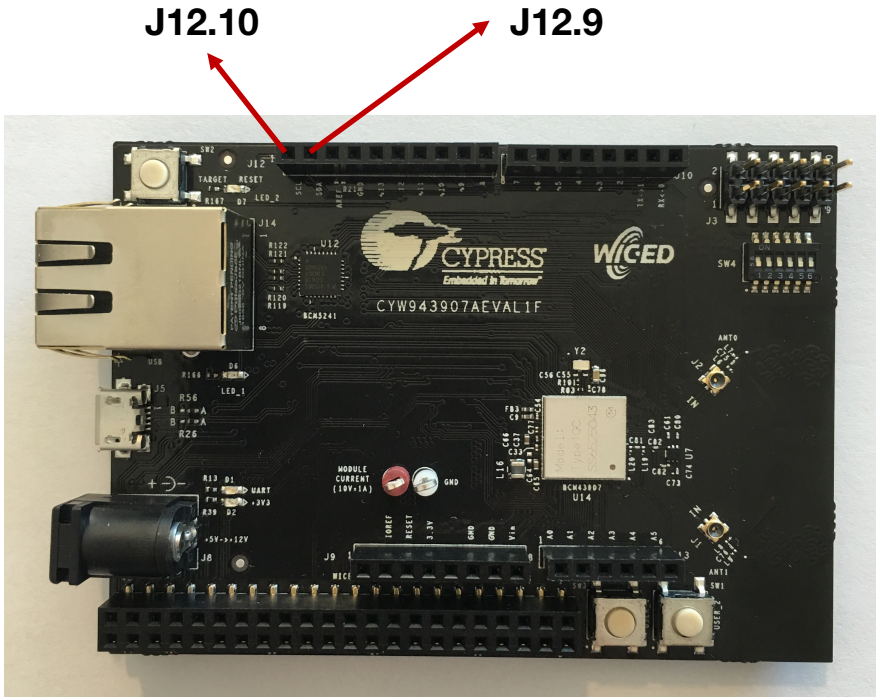
Example: UART-Receive (`uartreceive`)
Part: 2/2

```
while ( 1 )
    {
        if ( wiced_uart_receive_bytes(WICED_UART_1, &receiveChar,
                          &expected_data_size, ICED_NEVER_TIMEOUT )
                                                    == WICED_SUCCESS )
        {
            /* If we get here then a character has been received */
         if(receiveChar == '0') /* Turn LED off */
         {
                  wiced_gpio_output_low( WICED_LED1 );
         }
         if(receiveChar == '1') /* Turn LED on */
         {
                  wiced_gpio_output_high( WICED_LED1 );
         }
        }

    }
}
```

I2C

- The analog co-processor shield contains an I2C slave
- Connected to to the following pins

Eval Board Header	CYW43907 Pin Name/ Kit Signal Name	ARDUINO Header Name	WICED Enumeration	Alternate Enumeration
J12.9	I2C_1_SDA	SDA	WICED_GPIO_50	WICED_I2C_2
J12.10	I2C_1_SCL	SCL	WICED_GPIO_51	WICED_I2C_2



I2C

- **I2C slave address = 0x42**
- Standard Speed (100kHz)
- EZI2C register access
  - The first byte written is the register offset
  - All reads start at the previous write offset
  - The register map is as follows:

Offset	Description	Detail
0x00–0x03	DAC value	This value is used to set the DAC output voltage
0x04	LED Values	4 least significant bits control CSLED3-CSLED0
0x05	LED Control	Set bit 1 in this register to allow the LED Values register to control the LEDs instead of the CapSense (CS) buttons
0x06	Button Status	
0x07–0x0A	Temperature	Floating point temperature measurement from the thermistor
0x0B–0x0E	Humidity	Floating point humidity measurement
0x0F–0x12	Ambient Light	Floating point ambient light measurement
0x13–0x16	Potentiometer	Floating point potentiometer voltage measurement

I2C

❑ **Example: We are interested to implement the following application:**

- When the button on the base board is pressed, send a character over the I2C bus to the shield board
- This is used by the processor on the shield to toggle through the four LEDs on the shield

- We need to use the following facilities:

```
wiced_result_t wiced_i2c_init( const wiced_i2c_device_t* device );
```

Initializes an I2C interface: Prepares an I2C hardware interface for communication as a **master**

param [in] `device`: The device for which the i2c port should be initialized
return `WICED_SUCCESS`: on success
return `WICED_ERROR`: if an error occurred during initialization

I2C

```
wiced_result_t wiced_i2c_init_tx_message
        ( wiced_i2c_message_t* message, const void* tx_buffer,
                uint16_t tx_buffer_length,
                        uint16_t retries, wiced_bool_t disable_dma );
```

Initialize the `wiced_i2c_message_t` structure for i2c tx transaction

param[out] **message:** Pointer to a message structure, this should be a valid pointer
param[in] **tx_buffer:** Pointer to a tx buffer that is already allocated
param[in] **tx_buffer_length:** Number of bytes to transmit
param[in] **retries:** The number of times to attempt send a message in case it can't not be sent
param[in] **disable_dma:** If true, disables the dma for current tx transaction. You may find it useful to switch off dma for short tx messages.
return **WICED_SUCCESS:** message structure was initialized properly
return **WICED_BADARG:** one of the arguments is given incorrectly

I2C

```
wiced_result_t wiced_i2c_transfer
        ( const wiced_i2c_device_t* device,
                wiced_i2c_message_t* message,
                        uint16_t number_of_messages );
```

Transmits and/or receives data over an I2C interface

param[in] **device:** The i2c device to communicate with
param[in] **message:** A pointer to a message (or an array of messages) to be transmitted/received
param[in] **number_of_messages :** The number of messages to transfer. [1 .. N] messages
return **WICED_SUCCESS:** on success.
return **WICED_ERROR:** if an error occurred during message transfer

Software Development using WICED Studio

Example: I2C-Write (`i2cwrite`)
Part: 1/3

```
#include "wiced.h"

#define I2C_ADDRESS  (0x42) //I2C slave address
#define RETRIES      (1)
#define DISABLE_DMA  (WICED_TRUE)
#define NUM_MESSAGES (1)

/* I2C register locations */
#define CONTROL_REG (0x05) //offset: LED control
#define LED_REG     (0x04) //offset: LED values

volatile wiced_bool_t buttonPress = WICED_FALSE;

/* Interrupt service routine for the button */
void button_isr(void* arg)
{
    buttonPress = WICED_TRUE;
}

/* Main application */
void application_start( )
{
    wiced_init();/* Initialize the WICED device */

    wiced_gpio_input_irq_enable(WICED_SH_MB1, IRQ_TRIGGER_FALLING_EDGE,
                                button_isr, NULL); /* Setup interrupt */
```

See the register map table

Software Development using WICED Studio

Example: : I2C-Write (`i2cwrite`)
Part: 2/3

```
/* Main application */
void application_start( )
{
    wiced_init();       /* Initialize the WICED device */

    wiced_gpio_input_irq_enable(WICED_SH_MB1, IRQ_TRIGGER_FALLING_EDGE,
                        button_isr, NULL); /* Setup interrupt */

    /* Setup I2C master */
    const wiced_i2c_device_t i2cDevice = {
        .port = WICED_I2C_2,
        .address = I2C_ADDRESS, //The address of the device on the I2C bus
        .address_width = I2C_ADDRESS_WIDTH_7BIT,
        .speed_mode = I2C_STANDARD_SPEED_MODE
    };

    wiced_i2c_init(&i2cDevice); //Initializes an I2C interface

    /* Setup transmit buffer and message */
    /* We will always write an offset and then a single value,
     * so we need 2 bytes in the buffer */
    uint8_t tx_buffer[] = {0, 0};
    wiced_i2c_message_t msg;
    wiced_i2c_init_tx_message(&msg, tx_buffer, sizeof(tx_buffer),
                              RETRIES, DISABLE_DMA);
```

Setup I2C master

Prepare I2C message

Software Development using WICED Studio

Example: : I2C-Write (`i2cwrite`)
Part: 3/3

```
 /* Write a value of 0x01 to the control register to enable control of the
  * CapSense LEDs over I2C */
    tx_buffer[0] = CONTROL_REG;
    tx_buffer[1] = 0x01;
    wiced_i2c_transfer(&i2cDevice, &msg, NUM_MESSAGES);

    tx_buffer[0] = LED_REG; /* Set offset for the LED register (0x04) */

    while ( 1 )
    {
        if(buttonPress)
        {
                /* Send new I2C data */
                wiced_i2c_transfer(&i2cDevice, &msg, NUM_MESSAGES);
                tx_buffer[1] = tx_buffer[1] << 1; /* Shift to the next LED */
                if (tx_buffer[1] > 0x08) /* Reset after turning on LED3 */
                {
                        tx_buffer[1] = 0x01;
                }

                buttonPress = WICED_FALSE; // Reset flag for next button press
        }
    }
}
```

We set bit 1 in register 0x05 to allow the LED values register (0x04) to control the LEDs

Modify and send I2C message

I2C

❑ **Student Work:** Develop the following code:

- The four LEDs on the shield blink sequentially from left to right
- The duration of each LED's turn on time is 200ms
- When `WICED_SH_MB1` is pressed, the direction of LED blinking is changed

I2C

❑ **Example:** We are interested to implement the following application:

- **When the button on the base board is pressed, I2C is used to read the temperature, humidity, light, and PWM values from the analog co-processor on the shield board**
- **The values are printed to the UART**

- We need to set the offset to 0x07 to read the temperature
- We can do this just once and it will stay set for all future reads
- With an offset of 0x07 you can read 16 bytes to get the **temperature**, **humidity**, ambient **light**, and **potentiometer** values (4 bytes each)

Example: I2C-Read (`i2cread`)
Part: 1/3

```
#include "wiced.h"

#define I2C_ADDRESS (0x42)
#define RETRIES (1)
#define DISABLE_DMA (WICED_TRUE)
#define NUM_MESSAGES (1)

#define TEMPERATURE_REG  0x07

volatile wiced_bool_t buttonPress = WICED_FALSE;


/* Interrupt service routine for the button */
void button_isr(void* arg)
{
        buttonPress = WICED_TRUE;
}


/* Main application */
void application_start( )
{
    wiced_init();/* Initialize the WICED device */

    wiced_gpio_input_irq_enable(WICED_SH_MB1, IRQ_TRIGGER_FALLING_EDGE,
                                button_isr, NULL); /* Setup interrupt */
```

See the register map table (this is the offset of temperature register)

Example: I2C-Read (`i2cread`)
Part: 2/3

```
/* Setup I2C master */
const wiced_i2c_device_t i2cDevice = {
      .port = WICED_I2C_2,
      .address = I2C_ADDRESS,
      .address_width = I2C_ADDRESS_WIDTH_7BIT,
      .speed_mode = I2C_STANDARD_SPEED_MODE
};

wiced_i2c_init(&i2cDevice);

/* Tx buffer is used to set the offset */
uint8_t tx_buffer[] = {TEMPERATURE_REG};
wiced_i2c_message_t setOffset;
wiced_i2c_init_tx_message(&setOffset, tx_buffer,
                          sizeof(tx_buffer), RETRIES, DISABLE_DMA);



/* Initialize offset */
wiced_i2c_transfer(&i2cDevice, &setOffset, NUM_MESSAGES);
```

Preparing tx message for setting offset

Setting offset to 0x07

Software Development using WICED Studio

```
    /* Rx buffer is used to get temperature, humidity, light, and POT data –
     * 4 bytes each */
     struct {
          float temp;
          float humidity;
          float light;
          float pot;
     } rx_buffer;

     wiced_i2c_message_t msg;
     wiced_i2c_init_rx_message(&msg, &rx_buffer, sizeof(rx_buffer), RETRIES,
                                                                  DISABLE_DMA);

     while ( 1 ) {
          if(buttonPress) {
                    /* Get new data from I2C */
                    wiced_i2c_transfer(&i2cDevice, &msg, NUM_MESSAGES);

                    WPRINT_APP_INFO(("Temperature: %.1f\t Humidity: %.1f\t Light:
                              %.1f\t POT: %.1f\n", rx_buffer.temp
                                        rx_buffer.humidity, rx_buffer.light,
                                                             rx_buffer.pot));

                    /* Reset flag for next button press */
                    buttonPress = WICED_FALSE;
          }
     }
}
```

Preparing rx message

get data

Example: I2C-Read (`i2cread`)
Part: 3/3

```
Output: Temperature: 28.5  Humidity: 48.9   Light: 103.0   POT: 1.0
```

I2C

❑ **Example:** We are interested to implement an application that probes all I2C devices

- The I2C address is 7 bits
- 0x00 is a special "All Call" address
- All values above 124 (0x7C) are reserved for future purposes, so the only valid addresses are 1 – 123 (0x01 – 0x7B)

We need to use the following API:

```
wiced_bool_t wiced_i2c_probe_device
                ( const wiced_i2c_device_t* device, int retries );
```

Checks whether the device is available on a bus or not

```
 * @param[in]  device  : The i2c device to be probed
 * @param[in]  retries : The number of times to attempt to probe the
                         device
 *
 * @return    WICED_TRUE : device is found.
 * @return    WICED_FALSE: device is not found
```

Software Development using WICED Studio

Example: I2C-Probe (`i2cprobe`)
Part: 1/2

```
#include "wiced.h"
#define RETRIES (1)
#define MIN_I2C_ADDRESS (0x01)
#define MAX_I2C_ADDRESS (0x7B)
volatile wiced_bool_t buttonPress = WICED_FALSE;
/* Interrupt service routine for the button */
void button_isr(void* arg)
{
        buttonPress = WICED_TRUE;
}
/* Main application */
void application_start( )
{
    uint8_t i2cAddress = 0x00;

    wiced_init();/* Initialize the WICED device */

    wiced_gpio_input_irq_enable(WICED_SH_MB1, IRQ_TRIGGER_FALLING_EDGE,
                                button_isr, NULL); /* Setup interrupt */

    /* Setup I2C master device structure */
    wiced_i2c_device_t i2cDevice = {
        .port = WICED_I2C_2,
        .address = i2cAddress,
        .address_width = I2C_ADDRESS_WIDTH_7BIT,
        .speed_mode = I2C_STANDARD_SPEED_MODE
    };
```

Note that we probe `WICED_I2C_2`
Note: Using `WICED_I2C_1` we would find 0x33 (the onboard ADC)

Example: I2C-Probe (`i2cprobe`)
Part: 2/2

```
while ( 1 )
    {
        if(buttonPress)
        {
            WPRINT_APP_INFO(("\n"));
            /* Scan for I2C Devices */
            for(i2cAddress = MIN_I2C_ADDRESS;
                i2cAddress <= MAX_I2C_ADDRESS; i2cAddress++)
            {
                /* Put new address in the I2C device structure */
                i2cDevice.address = i2cAddress;
                /* Initialize I2C with the new address */
                wiced_i2c_init(&i2cDevice);

                /* Device was found at this address */
                if(wiced_i2c_probe_device(&i2cDevice, RETRIES) ==
                                                        WICED_TRUE)
                {
                        WPRINT_APP_INFO(("Device Found at: 0x%02X\n",
                                i2cAddress)); /* Print data to terminal */
                }
            }
                /* Reset flag for next button press */
                buttonPress = WICED_FALSE;
        }
    }
}
```

PWM

- **Example:** We are interested to implement an application that enables us to control the brightness of LED
- We need to configure a PWM to drive `WICED_SH_LED1` on the shield board instead of using the GPIO functions

- As `WICED_SH_LED1` is by default initialized to be controlled by GPIO driver, we need to call `wiced_gpio_deinit(…)` so that the PWM can drive the pin

```
wiced_result_t wiced_gpio_deinit( wiced_gpio_t gpio );

De-initializes a GPIO pin
 *
 * @param[in] gpio    : The gpio pin which should be de-initialized
 *
 * @return    WICED_SUCCESS : on success.
 * @return    WICED_ERROR   : if an error occurred with any step
 */
```

PWM

- `WICED_SH_LED1` is connected to `WICED_GPIO_17` so you need to find out which PWM is connected to that pin (look in the platform files, i.e., `platform.h`)

```
|WICED_SH_MB0   | WICED_GPIO_12   | D5              | Button MB0            | WICED_PWM_3          |
+---------------+-----------------+-----------------+-----------------------+----------------------+
|WICED_SH_MB1   | WICED_GPIO_3    | D3              | Button MB1            | WICED_PWM_6          |
+---------------+-----------------+-----------------+-----------------------+----------------------+
|WICED_SH_LED0  | WICED_GPIO_7    | D10             | LED0                  | WICED_PWM_2          |
+---------------+-----------------+-----------------+-----------------------+----------------------+
|WICED_SH_LED1  | WICED_GPIO_17   | D9              | LED1                  | WICED_PWM_5          |
+---------------+-----------------+-----------------+-----------------------+----------------------+
|WICED_ADC_0    | N/A             | A0              | Ambient Light Sensor  | N/A                  |
```

PWM

Initializes a PWM pin

```
 * Does not start the PWM output (use @ref wiced_pwm_start).
 * @param[in] pwm         : The PWM interface which should be initialized
 * @param[in] frequency   : Output signal frequency in Hertz
 * @param[in] duty_cycle  : Duty cycle of signal as a floating-point percentage
(0.0 to 100.0)
 *
 * @return    WICED_SUCCESS : on success.
 * @return    WICED_ERROR   : if an error occurred with any step
 */
wiced_result_t wiced_pwm_init( wiced_pwm_t pwm, uint32_t frequency,
                                                        float duty_cycle );
```

Starts PWM output on a PWM interface

```
 * @param[in] pwm         : The PWM interface which should be started
 *
 * @return    WICED_SUCCESS : on success.
 * @return    WICED_ERROR   : if an error occurred with any step
 */
wiced_result_t wiced_pwm_start( wiced_pwm_t pwm );
```

Example: PWM-LED (`pwm_led`)
Part: 1/1

```
#include "wiced.h"

#define  PWM_PIN  WICED_PWM_5

void application_start( )
{
    float duty_cycle = 0.0;

    wiced_init();/* Initialize the WICED device */

    // Need to de-init the GPIO if it is already set to drive the LED
    wiced_gpio_deinit(WICED_SH_LED1);

    while ( 1 )
    {
        wiced_pwm_init(PWM_PIN, 1000, duty_cycle);
        wiced_pwm_start(PWM_PIN);
        duty_cycle += 1.0;

        if(duty_cycle > 100.0)
        {
            duty_cycle = 0.0;
        }
        wiced_rtos_delay_milliseconds( 20 );
    }
}
```

Every 20ms we change the duty cycle of the 1KHz pulse generated

PWM

❑ **Student Work:** Develop the following code:

- The LED brightness is controlled by light intensity

Software Development using WICED Studio

- By using an RTOS you can separate the system functions into separate tasks (called **threads**) and develop them in a somewhat independent fashion

- In this section we focus on RTOS facilities such as **threads**, **semaphores**, **mutex**, and **queues**

Threads

- Threads are at the heart of an RTOS
- We can **create a new thread** by calling **`wiced_rtos_create_thread()`**
- Arguments are as follows:
  - **`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 only used by the debugger
    - You can give it any name or just use NULL if you don't want a specific name

Threads

- **`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

Threads

- Assume you want to create a thread that runs the function `mySpecialThread`

```
#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);
```

Pointer to a thread handle data structure

A function pointer to the function that is the thread

- **The thread function must take a single argument of type `wiced_thread_arg_t` and must have a `void` return**

Threads

- The body of a thread looks just like the “main” function of your application
- In fact, the main function (`application_start()`) 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

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

Threads

- 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 this case, after the other threads have started, the `application_start` function will just exist and will not take up any more CPU cycles

Threads

❑ **Example:** We are interested to implement the following application:

- Use a thread to blink an LED
- The blinking interval is received from the user as a 3 digit number

Software Development using WICED Studio

Example: Thread-LED-UART (`thread_led_uart`)
Part: 1/3

```
#include "wiced.h"
#define RX_BUFFER_SIZE (5)
/* Thread parameters */
#define THREAD_PRIORITY   (10)
#define THREAD_STACK_SIZE (1024)

volatile uint16_t interval; //LED blinking interval

// Define the thread function that will blink the LED on/off every “interval” ms
void ledThread(wiced_thread_arg_t arg) {
        wiced_bool_t led1 = WICED_FALSE;

        while(1) {
                /* Toggle LED1 */
                if ( led1 == WICED_TRUE ) {
                        wiced_gpio_output_low( WICED_SH_LED1 );
                        led1 = WICED_FALSE;
                }
                else {
                        wiced_gpio_output_high( WICED_SH_LED1 );
                        led1 = WICED_TRUE;
                }
                wiced_rtos_delay_milliseconds( interval );
        }
}
```

The thread wakes up every “interval” ms and runs the loop

Software Development using WICED Studio

Example: Thread-LED-UART (`thread_led_uart`)
Part: 2/3

```
void application_start( )
{
    uint32_t expected_data_size = 3; //Receive 3 digits from user
    char     receiveString[expected_data_size];
    interval = 250;

    wiced_thread_t  ledThreadHandle;

    wiced_init();/* Initialize the WICED device */

    /* Configure and start the UART. */
    wiced_ring_buffer_t   rx_buffer;
    uint8_t               rx_data[RX_BUFFER_SIZE];
    ring_buffer_init(&rx_buffer, rx_data, RX_BUFFER_SIZE );

    wiced_uart_config_t uart_config =
    {
        .baud_rate    = 115200,
        .data_width   = DATA_WIDTH_8BIT,
        .parity       = NO_PARITY,
        .stop_bits    = STOP_BITS_1,
        .flow_control = FLOW_CONTROL_DISABLED,
    };

    wiced_uart_init( STDIO_UART, &uart_config, &rx_buffer); /* Setup UART */
```

For the configuration to work, we need to use `GLOBAL_DEFINES := WICED_DISABLE_STDIO` in make file

Software Development using WICED Studio

Example: Thread-LED-UART (`thread_led_uart`)
Part: 3/3

```
    /* Initialize and start a new thread */
    wiced_rtos_create_thread(&ledThreadHandle, THREAD_PRIORITY,
                             "ledThread", ledThread, THREAD_STACK_SIZE, NULL);

     wiced_uart_transmit_bytes(WICED_UART_1,
                    "Enter a value between 000 and 999:\n", 35);

     while ( 1 ) {
         if ( wiced_uart_receive_bytes( STDIO_UART, &receiveString,
                    &expected_data_size, WICED_NEVER_TIMEOUT )
                                                    == WICED_SUCCESS )
         {
             interval = atoi(receiveString); //ASCI to integer conversion


             wiced_uart_transmit_bytes(WICED_UART_1,
                                  "\nNew Value entered: " , 21);
             wiced_uart_transmit_bytes(WICED_UART_1, &receiveString , 3);
             wiced_uart_transmit_bytes(WICED_UART_1, "\n" , 1);
         }
     }
}
```

Semaphore

- A **semaphore** is **a signaling mechanism between threads**
- 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
- You can use a semaphore to signal between threads that something is ready

- **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 indefinitely for the semaphore to be set, rather than continuing execution after a specific delay, then use `WICED_WAIT_FOREVER`

Semaphore

Initializes a semaphore

```
 * @param[in] semaphore : A pointer to the semaphore handle to be initialized
 * @return    WICED_SUCCESS : on success.
 * @return    WICED_ERROR   : if an error occurred
wiced_result_t wiced_rtos_init_semaphore( wiced_semaphore_t* semaphore );
```

Get (wait/decrement) a semaphore

```
 * Attempts to get (wait/decrement) a semaphore. If semaphore is at zero
 * already, then the calling thread will be suspended until another thread sets
 * the semaphore with @ref wiced_rtos_set_semaphore
 *
 * @param[in] semaphore : A pointer to the semaphore handle
 * @param[in] timeout_ms: The number of milliseconds to wait before returning
 *
 * @return    WICED_SUCCESS : on success.
 * @return    WICED_ERROR   : if an error occurred

wiced_result_t wiced_rtos_get_semaphore( wiced_semaphore_t* semaphore,
                                         uint32_t timeout_ms );
```

Semaphore

❑ **Example:** We are interested to implement the following application:

- **The main thread looks for a button press, then uses a semaphore to communicate to the toggle LED thread**

- You can use a pin interrupt to detect the button press and set the semaphore
- 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

Software Development using WICED Studio

Example: Semaphore-LED (`semaphore_led`)
Part: 1/2

```
#include "wiced.h"
#define THREAD_PRIORITY   (10)
#define THREAD_STACK_SIZE (1024)

static wiced_semaphore_t semaphoreHandle;

/* Interrupt service routine for the button */
void button_isr(void* arg) {
        wiced_rtos_set_semaphore(&semaphoreHandle); /* Set the semaphore */
}

void ledThread(wiced_thread_arg_t arg) // The thread function for LED toggle
{
    static wiced_bool_t led1 = WICED_FALSE;
    while(1)      {
     wiced_rtos_get_semaphore(&semaphoreHandle, WICED_WAIT_FOREVER);

        if ( led1 == WICED_TRUE ) {
                wiced_gpio_output_low( WICED_SH_LED1 );
                led1 = WICED_FALSE;
        }
        else {
                wiced_gpio_output_high( WICED_SH_LED1 );
                led1 = WICED_TRUE;
        }
        }
}
```

Check for the semaphore here
If it is not set, then this thread will suspend until the semaphore is set by the button thread

Software Development using WICED Studio

Example: Semaphore-LED (`semaphore_led`)
Part: 2/2

```
void application_start( )
{
    wiced_thread_t ledThreadHandle;

    wiced_init();/* Initialize the WICED device */

    /* Setup the semaphore which will be set by the button interrupt */
    wiced_rtos_init_semaphore(&semaphoreHandle);

    /* Initialize and start LED thread */
    wiced_rtos_create_thread(&ledThreadHandle, THREAD_PRIORITY,
                             "ledThread", ledThread, THREAD_STACK_SIZE, NULL);

    /* Setup button interrupt */
    wiced_gpio_input_irq_enable(WICED_SH_MB1, IRQ_TRIGGER_FALLING_EDGE,
                                                    button_isr, NULL);

    /* No while(1) here since everything is done by the new thread. */

}
```

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

Mutex

❑ **Example:** We are interested to implement the following application:

- Use a mutex to lock WPRINT_APP_INFO and avoid concurrent access of two threads to this function
- Note: The WPRINT_APP_INFO may go haywire if two threads write to it at the same time
- Use a delay of 250ms in one thread and a delay of 249ms in the other thread
- Add a WPRINT_APP_INFO to each of the threads with different messages

Software Development using WICED Studio

Example: MUTEX-UART (`mutex_uart`)
Part: 1/3

Comment out this line to see what happens without mutex

This thread function blinks LED1 on/off every 250ms

```
#include "wiced.h"
#define USE_MUTEX

#define THREAD_PRIORITY      (10)
#define THREAD_STACK_SIZE    (1024)

static wiced_mutex_t printMutexHandle;

void led1Thread(wiced_thread_arg_t arg) {
        wiced_bool_t led1 = WICED_FALSE;
        while(1) {
                #ifdef USE_MUTEX
                    wiced_rtos_lock_mutex(&printMutexHandle);
                #endif
                WPRINT_APP_INFO(("TOGGLE LED1\n"));
                #ifdef USE_MUTEX
                    wiced_rtos_unlock_mutex(&printMutexHandle);
                #endif

                /* Toggle LED1 */
                if ( led1 == WICED_TRUE ) {
                    wiced_gpio_output_low( WICED_SH_LED1 );   led1 = WICED_FALSE;
                }
                else {
                    wiced_gpio_output_high( WICED_SH_LED1 ); led1 = WICED_TRUE;
                }
                wiced_rtos_delay_milliseconds( 250 );
        }
}
```

Software Development using WICED Studio

Example: MUTEX-UART (`mutex_uart`)
Part: 2/3

This thread function blinks LED0 on/off every 249ms

```
void led0Thread(wiced_thread_arg_t arg)
{
        wiced_bool_t led0 = WICED_FALSE;

        while(1) {
                #ifdef USE_MUTEX
                        wiced_rtos_lock_mutex(&printMutexHandle);
                #endif
                WPRINT_APP_INFO(("TOGGLE LED0\n"));
                #ifdef USE_MUTEX
                        wiced_rtos_unlock_mutex(&printMutexHandle);
                #endif

                /* Toggle LED2 */
                if ( led0 == WICED_TRUE ) {
                    wiced_gpio_output_low( WICED_SH_LED0 ); led0 = WICED_FALSE;
                }
                else     {
                    wiced_gpio_output_high( WICED_SH_LED0 ); led0 = WICED_TRUE;
                }

                wiced_rtos_delay_milliseconds( 250 );
        }
}
```

Software Development using WICED Studio

Example: MUTEX-UART (`mutex_uart`)
Part: 3/3

```
void application_start( )
{
    wiced_thread_t led1ThreadHandle;
    wiced_thread_t led0ThreadHandle;


    wiced_init();/* Initialize the WICED device */

    /* Initialize the Mutex */
    wiced_rtos_init_mutex(&printMutexHandle);


    /* Initialize and start threads */
    wiced_rtos_create_thread(&led1ThreadHandle, THREAD_PRIORITY,
                             "led1Thread", led1Thread, THREAD_STACK_SIZE, NULL);

    wiced_rtos_create_thread(&led0ThreadHandle, THREAD_PRIORITY,
                             "led0Thread", led0Thread, THREAD_STACK_SIZE, NULL);

    /* No while(1) here since everything is done by the new threads. */
}
```

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

Queue

- 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 indefinitely for space in the queue rather than continuing execution after a specific delay then use `WICED_WAIT_FOREVER`
  - If you want the project to continue on immediately if there isn't room in the queue, then use `WICED_NO_WAIT`
- 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

Pushes an object onto a queue

```
 * @param[in] queue   : A pointer to the queue handle
 * @param[in] message: The object to be added to the queue
 * @param[in] timeout_ms : The number of milliseconds to wait before returning
 *
 * @return    WICED_SUCCESS : on success.
 * @return    WICED_ERROR   : if an error or timeout occurred
 */
wiced_result_t wiced_rtos_push_to_queue( wiced_queue_t* queue,
                                         void* message, uint32_t timeout_ms );
```

Queue

- You should **always initialize a queue before starting any threads that use it**
- 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 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**

Initializes a FIFO queue

```
 * @param[in] queue  : A pointer to the queue handle to be initialized
 * @param[in] name   : A text string name for the queue (NULL is allowed)
 * @param[in] message_size : Size in bytes of objects held in the queue
 * @param[in] number_of_messages : max number of objects in the queue
 * @return    WICED_SUCCESS : on success.
 * @return    WICED_ERROR   : if an error occurred

wiced_result_t wiced_rtos_init_queue( wiced_queue_t* queue, const char* name,
                                      uint32_t message_size, uint32_t number_of_messages );
```

Queue

❑ **Example:** We are interested to implement the following application:

- **Use a queue to send a message to indicate the number of times to blink an LED**

- 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
- Remember to use `WICED_NO_WAIT` for the timeout parameter in the ISR; Otherwise the push function will not work
- In the LED thread, pop the value from the queue to determine how many times to blink the LED

Example: Queue-LED (`queue_led`)
Part: 1/3

```
#include "wiced.h"

/* Thread parameters */
#define THREAD_PRIORITY   (10)
#define THREAD_STACK_SIZE (1024)
/* The queue messages will be 4 bytes each (a 32 bit integer) */
#define MESSAGE_SIZE    (4)
#define QUEUE_SIZE      (10)

static wiced_queue_t  queueHandle;
static wiced_thread_t ledThreadHandle;

wiced_bool_t buttonFlag = WICED_FALSE;

/* Interrupt service routine for the button */
void button_isr(void* arg)
{
        static uint32_t blinks = 0;

        blinks++;


        wiced_rtos_push_to_queue(&queueHandle, &blinks, WICED_NO_WAIT);
}
```

- **We will use `WICED_NO_WAIT` here so that the ISR won't lock execution if the queue is full**
- Note: `WICED_WAIT_FOREVER` is not allowed inside an ISR

Example: Queue-LED (`queue_led`) **Part:** 2/3

```
/* Define the thread function that will toggle the LED */
void ledThread(wiced_thread_arg_t arg)
{
    static uint32_t message;

    while(1)
    {
        uint32_t i; /* Loop Counter */


        wiced_rtos_pop_from_queue(&queueHandle, &message, WICED_WAIT_FOREVER);

        /* Blink LED1 the specified number of times */
        for(i=0; i < message; i++)
        {
                wiced_gpio_output_high( WICED_SH_LED1 );
                wiced_rtos_delay_milliseconds(150);
                wiced_gpio_output_low( WICED_SH_LED1 );
                wiced_rtos_delay_milliseconds(150);
        }

        // Wait 1 second between sets of blinks
        wiced_rtos_delay_milliseconds(1000);
    }
}
```

- If not empty, pull the value off the queue
- If empty, this will suspend the thread until a value is available

Example: Queue-LED (`queue_led`) **Part:** 3/3

```
void application_start( )
{

    wiced_init();/* Initialize the WICED device */


    /* Setup button interrupt */
    wiced_gpio_input_irq_enable(WICED_SH_MB1, IRQ_TRIGGER_FALLING_EDGE,
                                              button_isr, NULL);



    /* Initialize the queue */
    wiced_rtos_init_queue(&queueHandle, "blinkQueue", MESSAGE_SIZE, QUEUE_SIZE);



    /* Initialize and start LED thread */
    wiced_rtos_create_thread(&ledThreadHandle, THREAD_PRIORITY, "ledThread",
                                          ledThread, THREAD_STACK_SIZE, NULL);

    /* No while(1) here since everything is done by the new thread. */
}
```

The ISR pushes into the queue

ledThread pops from the queue

Timer

- An RTOS **timer** allows you **to schedule a function to run at a specified interval**

❑ Example: To send your data to the cloud every 10 seconds.

- When you setup the timer you specify the function you want to 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
- 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
- **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

Timer

Initializes a RTOS timer

```
 * Timer does not start running until @ref wiced_rtos_start_timer is called
 * @param[in] timer    : A pointer to the timer handle to be initialized
 * @param[in] time_ms  : Timer period in milliseconds
 * @param[in] function : The callback handler function that is called each
 *                       time the timer expires
 * @param[in] arg      : An argument that will be passed to the callback
 *                       function
 *
 * @return    WICED_SUCCESS : on success.
 * @return    WICED_ERROR   : if an error occurred

wiced_result_t wiced_rtos_init_timer( wiced_timer_t* timer,
                 uint32_t time_ms, timer_handler_t function, void* arg );
```

Starts a RTOS timer

```
 *
 * @param[in] timer    : A pointer to the timer handle to start
 *
 * @return    WICED_SUCCESS : on success.
 * @return    WICED_ERROR   : if an error occurred
 */
wiced_result_t wiced_rtos_start_timer( wiced_timer_t* timer );
```

Queue

❑ **Example:** We are interested to implement the following application:

- Make an LED blink using a timer

- 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
- Setup an RTOS timer that will call the LED function every 250ms

Example: Timer-LED (`timer_led`)
Part: 1/2

```
#include "wiced.h"

#define  TIMER_TIME  (250)

/* Define the  function that will blink the LED on/off */
void ledBlink(void* arg)
{
        static wiced_bool_t led1 = WICED_FALSE;

        /* Toggle LED1 */
        if ( led1 == WICED_TRUE )
        {
                wiced_gpio_output_low( WICED_SH_LED1 );
                led1 = WICED_FALSE;
        }
        else
        {
                wiced_gpio_output_high( WICED_SH_LED1 );
                led1 = WICED_TRUE;
        }
}
```

Software Development using WICED Studio

Example: Timer-LED (`timer_led`)
Part: 2/2

```
void application_start( )
{
    wiced_timer_t timerHandle;

    wiced_init();      /* Initialize the WICED device */

    /* Initialize and start a timer */
    wiced_rtos_init_timer(&timerHandle, TIMER_TIME, ledBlink, NULL);
    wiced_rtos_start_timer(&timerHandle);

    while ( 1 )
    {
        /* Nothing needed here since we only have one thread. */
    }
}
```

Software Development using Contiki

Installation on MAC OS

- Clone the github repository
  - `git clone git://github.com/contiki-os/contiki.git contiki`
- Enter Contiki root directory
  - `git init`
  - `git pull`
  - `git submodule init`
  - `git submodule update`
- Install ARM compiler:
  - `brew tap PX4/homebrew-px4`
  - `brew update`
  - `brew install gcc-arm-none-eabi-48`
- Install bin to hex converter
  - `brew install srecord`
- Install flash programmer
  - `http://software-dl.ti.com/ccs/esd/uniflash/uniflash_sl.4.1.1250.dmg`

Compiling and Flashing

- Move into `contiki/examples/cc26xx` folder
- In the `makefile`
  - `CONTIKI = path to the contiki folder`
- The `Makefile.target` includes
  - `TARGET = srf06-cc26xx`
  - `BOARD = sensortag/cc2650`

- In the project folder type
  - `make`
- Flash the bin file using Uniflash (downloadable from `ti.com`)

Eclipse Setup

- Download Eclipse installer from:
  `https://www.eclipse.org/downloads/`
- Install Eclipse CDT
- Clone Contiki into Eclipse through Import->git

- **Contiki documentation** is available at:
  `http://contiki.sourceforge.net/docs/2.6/index.html`

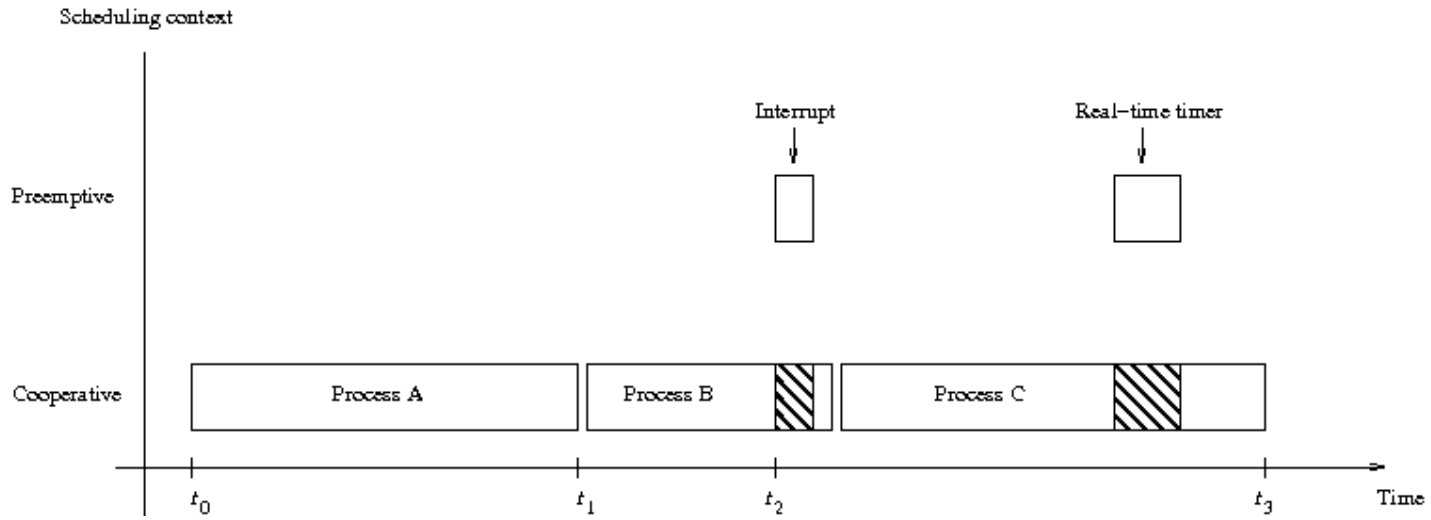
SDK Structure

- Contiki code structure:
  - **`apps`**: applications such as web browser, ping, shell, mqtt and telnet
  - **`core`**: source code for main core of Contiki
    - **`core/dev`**: source codes for devices such as LED, sensor, button…
    - **`core/net`**: folders for networking stack
  - **`cpu`**: source code for all computational units for sensor nodes
  - **`examples`**: implementation of sample applications
  - **`tools`**: tools for testing applications

Programming Concepts: Processes

- Code in Contiki runs in either of two execution contexts:
  - **Cooperative** code runs sequentially with respect to other cooperative code
  - **Preemptive** code temporarily stops the cooperative code

- **Contiki processes run in the cooperative context, whereas interrupts and real-time timers run in the preemptive context**



Programming Concepts: Processes

- **Processes always run in the cooperative context**
- The preemptive context is used by interrupt handlers in device drivers and by real-time tasks that have been scheduled for a specific deadline

Programming Concepts: Processes

The Process Thread

- A process thread contains the code of the process
- The process thread is a single protothread that is invoked from the process scheduler

An example process thread:

```
PROCESS_THREAD(hello_world_process, ev, data)
 {
   PROCESS_BEGIN();

   printf("Hello, world\n");

   PROCESS_END();
 }
```

Programming Concepts: Processes

`PROCESS_THREAD(name, process_event_t, process_data_t)`

- This function is used to define the **protothread** of a process
- The process is **called whenever an event occurs in the system**
- Each process in the module **requires an event handler**
- **`name`**: The variable name of the process structure to access the process
- **`process_event_t`**: The variable of type character (for debugging)
  - If this variable is same as `PROCESS_EVENT_EXIT` then `PROCESS_EXITHANDLER` is invoked
- Within the body of `PROCESS_THREAD` there are 3 major tasks:
  - **Initialization**: (1) allocate resources, (2) define variables, (3) begin process
  - **Infinite loop**: while(1) is used to create an infinite loop in which the actual event-handling response takes place
  - **Deallocation**: (1) end process, (2) deallocate resources

Programming Concepts: Processes

What is ”Prothread”?

- A protothread is a way to structure code in a way that allows the system to run other activities when the code is waiting for something to happen
- Protothread provides a way for C functions to work in a way that is similar to threads, without the memory overhead of threads
- Contiki process implement their own version of protothreads, that allow processes to wait for incoming events

Process-specific protothread macros that are used in Contiki processes:

```
PROCESS_BEGIN(); // Declares the beginning of a process' protothread.
PROCESS_END(); // Declares the end of a process' protothread.
PROCESS_EXIT(); // Exit the process.
PROCESS_WAIT_EVENT(); // Wait for any event.
PROCESS_WAIT_EVENT_UNTIL(); // Wait for an event, but with a condition.
PROCESS_YIELD(); // Wait for any event, equivalent to ROCESS_WAIT_EVENT().
PROCESS_WAIT_UNTIL(); // Wait for a given condition; may not yield the
process.
PROCESS_PAUSE(); // Temporarily yield the process.
```

Programming Concepts: Processes

- In Contiki, a process is run when it receives an event
- There are two types of events: **asynchronous events** and **synchronous events**

- **Asynchronous events** are delivered to the receiving process some time after they have been posted
- The events on the event queue are delivered to the receiving process by the kernel
- The kernel loops through the event queue and delivers the events to the processes on the queue by invoking the processes
- The receiver of an asynchronous event can either be a specific process, or all running processes
- Asynchronous events are posted with the `process_post()` function
  - It first checks the size of the current event queue to determine if there is room for the event on the queue; If not, the function returns an error
  - If there is room for the event on the queue, the function inserts the event at the end of the event queue and returns

Programming Concepts: Processes

- Unlike asynchronous events, **synchronous events are delivered directly when they are posted**
- Synchronous events can only be posted to a specific process
- Because synchronous events are delivered immediately, posting a **synchronous event is functionally equivalent to a function call**
- The process to which the event is delivered is directly invoked, and the process that posted the event is blocked until the receiving process has finished processing the event
- The receiving process is, however, not informed whether the event was posted synchronously or asynchronously

Programming Concepts: Processes

```
#define  PROCESS_WAIT_EVENT()  PROCESS_YIELD()
```

- Yield the currently running process until a condition occurs
- This macro blocks the currently running process until the process receives an event

```
#define  PROCESS_WAIT_EVENT_UNTIL(c)
                                  PROCESS_YIELD_UNTIL(c)
```

- Wait for an event to be posted to the process, with an extra condition
- This macro is similar to `PROCESS_WAIT_EVENT()` in that it blocks the currently running process until the process receives an event
- But `PROCESS_WAIT_EVENT_UNTIL()` takes an extra condition which must be true for the process to continue

Programming Concepts: Processes

Polling

- A poll request is a special type of event
- A process is polled by calling the function `process_poll()`
- Calling this function on a process causes the process to be scheduled as quickly as possible
- The process is passed a special event that informs the process that it has been polled

Programming Concepts: Processes

Event Identifiers

- Events are identified by an event identifier
- The event identifier is an 8-bit number that is passed to the process that receives an event
- The event identifier allows the receiving process to perform different actions depending on what type of event that was received
- Identifiers above 128 are managed by the kernel
- **Identifiers below 127 can be freely used within a user process**

Programming Concepts: Processes

Some important event identifiers

- **`PROCESS_EVENT_INIT`:** This event is sent to new processes when they are initiated
- **`PROCESS_EVENT_POLL`**: This event is sent to a process that is being polled
- **`PROCESS_EVENT_CONTINUE`**: This event is sent by the kernel to a process that is waiting in a `PROCESS_YIELD()` statement
- **`PROCESS_EVENT_MSG`**: This event is sent to a process that has received a communication message
  - It is **typically used by the IP stack** to inform a process that a message has arrived, but can also be used between processes as a generic event indicating that a message has arrived
- **`PROCESS_EVENT_TIMER`**: This event is sent to a process when an event timer (etimer) has expired

Programming Concepts: Processes

The Process Scheduler

- The purpose of the process scheduler is to invoke processes when it is their time to run
- The process scheduler invokes process by calling the function that implements the process' thread
- All process invocation in Contiki is done **in response to an event being posted to a process, or a poll has being requested for the process**
- The process scheduler passes the event identifier to the process that is being invoked

"Event-Printing" Application

```
1. #include "contiki.h" /* For contiki applications */

2. /* Contiki application : Declare the PROCESS */
3. PROCESS(name_of_your_process, "Process Name");

4. /* Start your PROCESS when Contiki starts*/
5. AUTOSTART_PROCESSES(&name_of_your_process);

6. /* Declare what the PROCESS DOES*/
7. PROCESS_THREAD(name_of_your_process, ev, data)
8. {
9.      PROCESS_BEGIN(); /* Begin the PROCESS*/
10.     while (1) {
11.             PROCESS_WAIT_EVENT()
12.             printf("Hello! – I got event number: %d\n");
13.     }
14.     PROCESS_END();   /* end the PROCESS */
15.}
```

Example: Event-Printing (`event_printing`)
Part: 1/1

Open a serial port tool, and you must see "`Hello! – I got event number: 129`" printed

Programming Concepts: Processes

- We begin the process with the `PROCESS_BEGIN()` declaration
- This declaration marks start of code belonging to the process thread
- **Code that is placed above this declaration will be (re)executed each time the process is scheduled to run**
- Code placed below the declaration will be executed based on the actual process thread control flow

- Contiki processes cannot start loops that never end, but in this case we are safe to do this because we wait for events below
- The process waits for events at line 11
- After the process wakes up, the `printf()` statement at line 12 is executed
- This line simply prints out the event number of the event that the process received
- **The event number is contained in the `ev` variable**
- **If a pointer was passed along with the event, this pointer is available in the `data` variable**

Programming Concepts: Timers

Contiki has one clock module and a set of timer modules:

- **timer**: timers use system clock ticks
- **stimer**: use seconds to allow much longer time periods
- **ctimer**: provides callback timers and are used to schedule calls to callback functions after a period of time
  - The callback timers are, among other things, used throughout the Rime protocol stack to handle communication timeout
- **etimer**: provides event timers and are used to schedule events to Contiki processes after a period of time
- **rtimer**: The rtimer library provides scheduling of real-time tasks
  - The rtimer library pre-empt any running Contiki process in order to let the real-time tasks execute at the scheduled time

- The **timer** and **stimer** libraries provide the simplest form of timers and are used to check if a time period has passed
  - The applications need to ask the timers if they have expired

Programming Concepts: Timers

The clock module provides two functions for blocking the CPU:

- `clock_delay()`: which blocks the CPU for a specified delay
- `clock_wait()`: which blocks the CPU for a specified number of clock ticks
- These functions are normally only used in low-level drivers where it sometimes is necessary to wait a short time without giving up the control over the CPU
- A timer is declared as a **`struct timer`** and all access to the timer is made by a pointer to the declared time

```
clock_time_t clock_time(); // Get the system time.
unsigned long clock_seconds(); // Get the system time in seconds.
void clock_delay(unsigned int delay); // Delay the CPU.
void clock_wait(int delay); // Delay the CPU for a number of clock ticks.
void clock_init(void); // Initialize the clock module.
CLOCK_SECOND; // The number of ticks per second.
```

Programming Concepts: Timers

```
// Start the timer.
void etimer_set(struct etimer *t, clock_time_t interval);

// Restart the timer from the previous expiration time.
void etimer_reset(struct etimer *t);

// Restart the timer from current time.
void etimer_restart(struct etimer *t);

void etimer_stop(struct etimer *t); // Stop the timer.

// Check if the timer has expired.
int etimer_expired(struct etimer *t);

// Check if there are any non-expired event timers.
int etimer_pending();

// Get the next event timer expiration time.
clock_time_t etimer_next_expiration_time();

// Inform the etimer library that the system clock has changed.
void etimer_request_poll();
```

Programming Concepts: Timers

- The Contiki **`ctimer`** library provides a timer mechanism that **calls a specified function when a callback timer expires**

```
void ctimer_set(struct ctimer *c, clock_time_t t,
                void(*f)(void *), void *ptr); // Start the timer.

// Restart the timer from the previous expiration time.
void ctimer_reset(struct ctimer *t);

// Restart the timer from current time.
void ctimer_restart(struct ctimer *t);

void ctimer_stop(struct ctimer *t); // Stop the timer.

// Check if the timer has expired.
int ctimer_expired(struct ctimer *t);
```

"blinking" Application

Example: Blinking (`blinking`)
Part: 1/2

```
#include "contiki.h"
#include "sys/etimer.h"
#include "sys/ctimer.h"
#include "dev/leds.h"

#define LOOP_INTERVAL   (70)

static struct etimer et;
static struct ctimer timer;

PROCESS(sensortag_led_experiment, "sensortag_led_experiment");
AUTOSTART_PROCESSES(&sensortag_led_experiment);


static void toggleLED2(void *ptr) {
        printf("Toggle green LED\n");
        leds_toggle(LEDS_GREEN);
        ctimer_reset(&timer);
 }
```

Function to be called when the callback timer fires

“blinking” Application

Example: Blinking (`blinking`)
Part: 2/2

```
PROCESS_THREAD(sensortag_led_experiment, ev, data)
{
        PROCESS_BEGIN();
        printf("CC26XX LED Experiment\n");
        etimer_set(&et, LOOP_INTERVAL);
        ctimer_set(&timer, LOOP_INTERVAL/2, toggleLED2, NULL);

        while(1) {

                PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));

                //returns the current system time in clock ticks
                printf("Clock time: %lu\n", clock_time());
                //returns the current system time in seconds
                printf("Clock seconds: %lu\n", clock_seconds());

                leds_toggle(LEDS_RED);
                printf("Toggle red LED\n");
                etimer_reset(&et);
        }

PROCESS_END();
}
```

Programming Concepts: Memory Allocation

- Contiki provides three ways to allocate and deallocate memory:
  - **`memb`** memory block allocator – most frequently used
  - **`mmem`** managed memory allocator – like malloc()
  - The standard C library **`malloc`** heap memory allocator – discouraged

- The **`memb`** library provides a set of memory block management functions
- Memory blocks are allocated as an array of objects of constant size and **are placed in static memory**

```
MEMB(name, structure, num); // Declare a memory block.
void memb_init(struct memb *m); // Initialize a memory block.
void *memb_alloc(struct memb *m); // Allocate a memory block.
int memb_free(struct memb *m, void *ptr); // Free a memory block.

// Check if an address is in a memory block.
int memb_inmemb(struct memb *m, void *ptr);
```

Programming Concepts: Memory Allocation

```
MEMB(name, structure, num);
```

- The `MEMB()` macro declares a memory block, which has the type `struct memb`
- Since the block is put into static memory, it is typically placed at the top of a C source file that uses memory blocks
- **`name`** identifies the memory block, and is later used as an argument to the other memory block functions
- The **`structure`** parameter specifies the C type of the memory block
- **`num`** represent the amount objects that the block accommodates

- After initializing a `struct memb`, we are ready to start allocating objects from it by using `memb_alloc()`
- `memb_alloc()` returns a pointer to the allocated object if the operation was successful, or NULL if the memory block has no free object

Programming Concepts: Memory Allocation

```
#include "contiki.h"
#include "lib/memb.h"

struct connection {
   int socket;
};

MEMB(connections, struct connection, 16);

struct connection *open_connection(int socket)
{
   struct connection *conn;

   conn = memb_alloc(&connections);
   if(conn == NULL) {
     return NULL;
   }
   conn->socket = socket;
   return conn;
}

void close_connection(struct connection *conn)
 {
   memb_free(&connections, conn);
 }
```

Define a memory block

Allocate memory

The `open_connection()` function allocates a new struct `connection` variable for each new connection identified by `socket`

Programming Concepts: Input/Output

- The type of I/O devices available to the programmer is highly dependent on the platform in use
- Contiki provides two basic interfaces that most platforms share: serial I/O and LEDs

"UART-LED" Application

Example: LED-UART (`led_uart`)
Part: 1/2

```
#include "contiki.h"
#include "sys/etimer.h"
#include "dev/leds.h"
#include "dev/serial-line.h"
#include "cc26xx-cc13xx/dev/uart1.h"
#include "button-sensor.h"
#include <stdio.h>

PROCESS(sensortag_serial, "sensortag_serial");
AUTOSTART_PROCESSES(&sensortag_serial);

int parse_uart_cmd(char* msg)
{
        if (!strcmp(msg, "GREENLED")) {
                leds_toggle(LEDS_YELLOW);
        } else if (!strcmp(msg, "REDLED")) {
                leds_toggle(LEDS_RED);
        }
        return 0;
}
```

"UART-LED" Application

Example: LED-UART (`led_uart`)
Part: 2/2

```
PROCESS_THREAD(sensortag_serial, ev, data)
{
        PROCESS_BEGIN();

      //Assigns a callback to be called when the UART receives a byte
       cc26xx_uart_set_input(serial_line_input_byte);

        printf ("Sensortag Status: Ready\n");

        while(1) {

                PROCESS_WAIT_EVENT ();

                if(ev == serial_line_event_message) {
                        printf("received line: %s\n", (char *)data);
                         parse_uart_cmd(data);
                }
        }

        PROCESS_END();
}
```

References

1. http://www.cypress.com/part/cyw943907aeval1f
2. http://www.cypress.com/documentation/development-kitsboards/cyw943907aeval1f-evaluation-kit
3. Express Logic Inc. "ThreadX," [Online]. Available: http://rtos.com/ products/threadx/
4. http://www.freertos.org
5. O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes, "Operating Systems for Low-End Devices in the Internet of Things: A Survey," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 720–734, 2016.
6. A. Dunkels, B. Grönvall, and T. Voigt, "Contiki–A lightweight and flexible operating system for tiny networked sensors," in *Proc. 29th Annu. Int. Conf. Local Comput. Netw. (LCN)*, 2004, pp. 455–462 [Online]. Available: http://dblp.uni-trier.de/db/conf/lcn/lcn2004.html# DunkelsGV04
7. "Contiki operating system," [Online]. Available: http://www.contiki- os.org.
8. Freescale. "The MC13224V SoC," [Online]. Available: http://www. freescale.com/webapp/sps/site/prod_summary.jsp?code=MC13224V
9. ST Microelectronics. "STM32F100VC microcontroller," [Online]. Available: http://www.st.com/web/catalog/mmc/FM141/SC1169/ SS1031/LN775/PF216851
10. W. Dong, C. Chen, X. Liu, and J. Bu, "Providing os support for wire- less sensor networks: Challenges and approaches," *IEEE Commun. Surv. Tuts.*, vol. 12, no. 4, pp. 519–530, 2010.
11. P. Rosenkranz, M. Wählisch, E. Baccelli, and L. Ortmann, "A distributed test system architecture for open-source IoT software," in *Proc. ACM MobiSys Workshop IoT Challenges Mobile Ind. Syst. (IoT-Sys)*, May 2015.
12. "Contiki operating system," [Online]. Available: http://www.contiki- os.org.