Chapter 3: Peripherals

After completing this chapter, you should be able to write firmware for the MCU peripherals (GPIOs, UARTs, Timers, PWMs, NVRAM, I²C, ADC and RTC). In addition, you will understand the role of the critical files related to the kit hardware platform and you will know how to re-map pin functions to different peripherals.

Table of contents

3.1 Documentation 3
3.1.1 Context Sensitive Help 5
3.1.2 Intellisense 5
3.2 Peripherals 6
3.2.1 GPIO 6
3.2.2 Debug Printing 8
3.2.3 PUART (Peripheral UART) 10
3.2.4 Timers 11
3.2.5 PWM 12
3.2.6 NVRAM 14
3.2.7 ADC 15
3.2.8 RTC (Real Time Clock) 16
3.3 WICED_RESULT_T 17
3.4 Exercises 18
Exercise 1: (GPIO) Blink an LED 18
Exercise 2: (GPIO) Add Debug Printing to the LED Blink Application 21
Exercise 3: (GPIO) Read the State of a Mechanical Button 22
Exercise 4: (GPIO) Use an Interrupt to Toggle the State of an LED 22
Exercise 5: (Timer) Use a Timer to Toggle an LED 23
Exercise 6: (PWM) LED brightness 23
Exercise 7: (PWM) LED toggling at specific frequency and duty cycle 24
Exercise 8: (NVRAM) Write and Read Data in the NVRAM 24
Exercise 9: (ADC) Calculate the resistance of a thermistor 26
Exercise 10: (UART) Send a value using the standard UART functions 27
Exercise 11: (UART) Get a value using the standard UART functions 27
Exercise 12: (RTC) Display Time and Date Data on the UART 28
3.5 Appendix 29
3.5.1 Exercise 1 Answers 29
3.5.2 Exercise 8 Answers 29

Document conventions

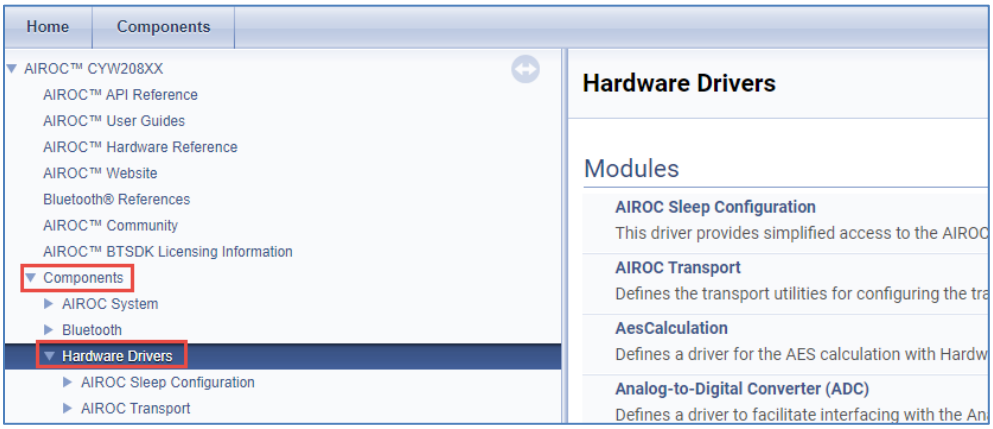
Convention	Usage	Example
`Courier New`	Displays code and text commands	`CY_ISR_PROTO(MyISR);` `make build`
Italics	Displays file names and paths	*sourcefile.hex*

Convention	Usage	Example
[**bracketed, bold**]	Displays keyboard commands in procedures	[**Enter**] or [**Ctrl**] [**C**]
Menu > Selection	Represents menu paths	**File > New Project > Clone**
Bold	Displays GUI commands, menu paths and selections, and icon names in procedures	Click the **Debugger** icon, and then click **Next**.

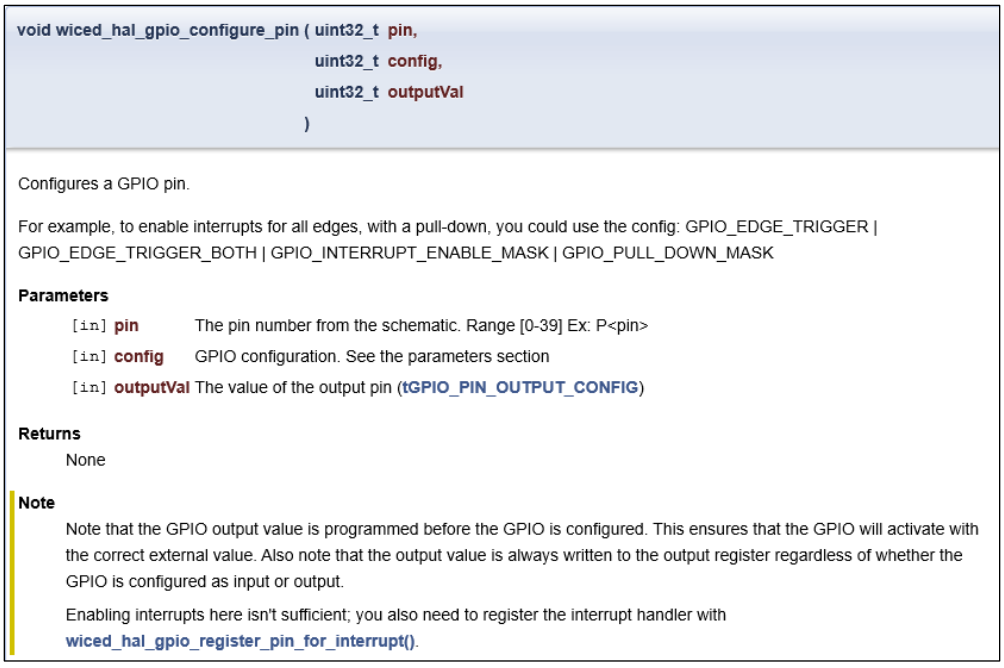
3.1 Documentation

You can find MCU peripheral documentation in the API guide, which is located in the Eclipse IDE for ModusToolbox™ Quick Panel: **AIROC™ Bluetooth® SDK Documentation > CYW208XX**). You can also open the *index.html* file from the *libs* directory of the application.

The peripheral APIs are presented under **Components > Hardware Drivers**. We will be using GPIO, Pulse Width Modulation (PWM), Peripheral UART (PUART), I²C, Real-Time Clock (RTC), and ADC.



As an example, click on or expand **GPIO** to see the list of GPIO APIs, and then click on `wiced_hal_gpio_configure_pin` for a description of the pin configuration function.



```
void wiced_hal_gpio_configure_pin ( uint32_t pin,
                                    uint32_t config,
                                    uint32_t outputVal
                                  )
```

Configures a GPIO pin.

For example, to enable interrupts for all edges, with a pull-down, you could use the config: GPIO_EDGE_TRIGGER | GPIO_EDGE_TRIGGER_BOTH | GPIO_INTERRUPT_ENABLE_MASK | GPIO_PULL_DOWN_MASK

Parameters

[in]	**pin**	The pin number from the schematic. Range [0-39] Ex: P<pin>
[in]	**config**	GPIO configuration. See the parameters section
[in]	**outputVal**	The value of the output pin (**tGPIO_PIN_OUTPUT_CONFIG**)

Returns

None

Note

Note that the GPIO output value is programmed before the GPIO is configured. This ensures that the GPIO will activate with the correct external value. Also note that the output value is always written to the output register regardless of whether the GPIO is configured as input or output.

Enabling interrupts here isn't sufficient; you also need to register the interrupt handler with **wiced_hal_gpio_register_pin_for_interrupt()**.

The description tells you what the function does, but it does not give complete information on the possible configuration values. To find that information, either look in the Enumeration section of the documentation (it's near the top of the GPIO page below Macros and Typedefs), or once you create an application in the Eclipse IDE, you can highlight the function in the C code, right click, and select **Open Declaration**.

Note: *You may have to clean and re-build the application and sometimes use the **Index > Refresh** menu item by right-clicking on a project in the Project Explorer window for this to work correctly.*

This will take you to the function declaration in the file *wiced_hal_gpio.h*. If you scroll to the top of this file, you will find a list of allowed choices. A subset of the choices is shown below:

```
/* Interrupt Enable
 * GPIO configuration bit 3, interrupt enable/disable defines
 */
GPIO_INTERRUPT_ENABLE_MASK    = 0x0008, /**< GPIO configuration bit 3 mask */
GPIO_INTERRUPT_ENABLE         = 0x0008, /**< Interrupt Enabled */
GPIO_INTERRUPT_DISABLE        = 0x0000, /**< Interrupt Disabled */

/* Interrupt Config
 * GPIO configuration bit 0:3, Summary of Interrupt enabling type
 */
GPIO_EN_INT_MASK              = GPIO_EDGE_TRIGGER_MASK | GPIO_TRIGGER_POLARITY_MASK | GPIO_DUAL_EDGE_TRIGGER_MASK | GPIO_INTERRUPT_ENABLE_MASK,
GPIO_EN_INT_LEVEL_HIGH        = GPIO_INTERRUPT_ENABLE | GPIO_LEVEL_TRIGGER, /**< Interrupt on level HIGH */
GPIO_EN_INT_LEVEL_LOW         = GPIO_INTERRUPT_ENABLE | GPIO_LEVEL_TRIGGER | GPIO_TRIGGER_NEG, /**< Interrupt on level LOW */
GPIO_EN_INT_RISING_EDGE       = GPIO_INTERRUPT_ENABLE | GPIO_EDGE_TRIGGER, /**< Interrupt on rising edge */
GPIO_EN_INT_FALLING_EDGE      = GPIO_INTERRUPT_ENABLE | GPIO_EDGE_TRIGGER | GPIO_TRIGGER_NEG, /**< Interrupt on falling edge */
GPIO_EN_INT_BOTH_EDGE         = GPIO_INTERRUPT_ENABLE | GPIO_EDGE_TRIGGER | GPIO_EDGE_TRIGGER_BOTH, /**< Interrupt on both edges */

/* GPIO Output Buffer Control and Output Value Multiplexing Control
 * GPIO configuration bit 4:5, and 14 output enable control and
 * muxing control
 */
GPIO_INPUT_ENABLE             = 0x0000, /**< Input enable */
GPIO_OUTPUT_DISABLE           = 0x0000, /**< Output disable */
GPIO_OUTPUT_ENABLE            = 0x4000, /**< Output enable */
GPIO_KS_OUTPUT_ENABLE         = 0x0001, /**< Keyscan output enable*/
GPIO_OUTPUT_FN_SEL_MASK       = 0x0000, /**< Output function select mask*/
GPIO_OUTPUT_FN_SEL_SHIFT      = 0,


/* Global Input Disable
 * GPIO configuration bit 6, "Global_input_disable" Disable bit
 * This bit when set to "1" , P0 input_disable signal will control
 * ALL GPIOs. Default value (after power up or a reset event) is "0".
 */
GPIO_GLOBAL_INPUT_ENABLE      = 0x0000, /**< Global input enable */
GPIO_GLOBAL_INPUT_DISABLE     = 0x0040, /**< Global input disable */

/* Pull-up/Pull-down
 * GPIO configuration bit 9 and bit 10, pull-up and pull-down enable
 * Default value is [0,0]--means no pull resistor.
 */
GPIO_PULL_UP_DOWN_NONE        = 0x0000, /**< No pull [0,0] */
GPIO_PULL_UP                  = 0x0400, /**< Pull up [1,0] */
GPIO_PULL_DOWN                = 0x0200, /**< Pull down [0,1] */
GPIO_INPUT_DISABLE            = 0x0600, /**< Input disable [1,1] (input disabled the GPIO) */
```

For example:

- An input pin with an active-low button would typically have the config set to:

  `GPIO_INPUT_ENABLE | GPIO_PULL_UP`

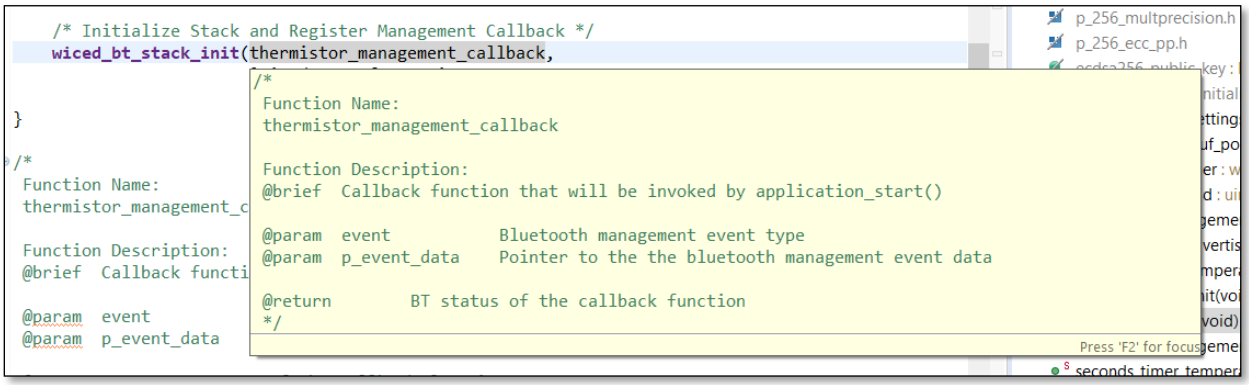
- An output pin driving an active-low LED would typically have the config set to:

  `GPIO_OUTPUT_ENABLE`

3.1.1 Context Sensitive Help

Right-clicking and selecting **Open Declaration** on function names and data-types inside the Eclipse IDE is often very useful for finding information on how to use functions and what values are allowed for parameters. Again, cleaning, building and/or refreshing the index may be necessary for this to work.

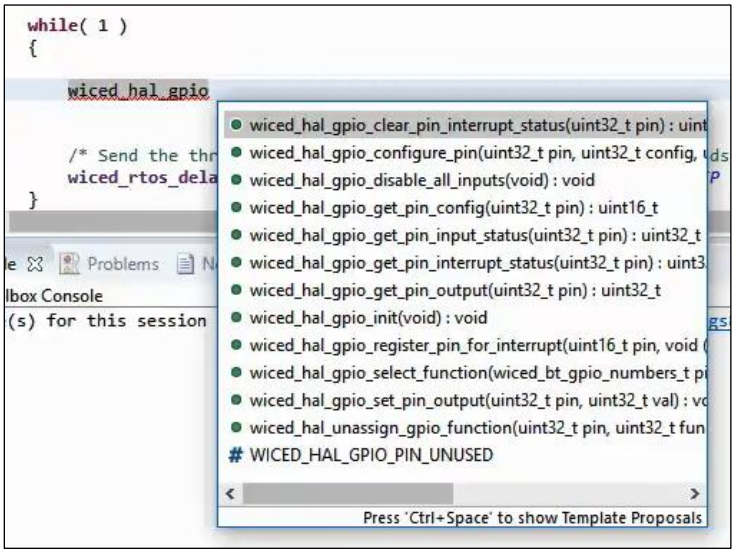
You can also just hover over a function name and get information like this:



3.1.2 Intellisense

Another very useful tool is to type [**Control**]+[**Space**] when you are in the code editor inside the Eclipse IDE. This will list possible completions for what you have already typed so you can select from the list. You can type the start of a function name, the start of a macro, the start of a variable, etc.

For example, if you type "`wiced_hal_gpio`" and press [**Control**]+[**Space**], you will get this list of all the matching items that the tool can find:



As with references, cleaning, building and/or rebuilding the index may make this work better.

3.2 Peripherals

3.2.1 GPIO

You will use this in Exercise 1: , Exercise 2: , and Exercise 4:

As explained previously, GPIOs must be configured using the function `wiced_hal_gpio_configure_pin`. The I/Os on the kit that are connected to specific peripherals such as LEDs and buttons are usually configured for you as part of the BSP, so you don't need to configure them explicitly in your applications unless you want to change a setting (for example to enable an interrupt on a button pin).

The configuration function takes two arguments: the first is the pin config and the second is the pin's initial drive state. The config is a `uint32` where each bit specifies an option. These include input and output buffer settings, resistive pull-up and pull-downs, drive strength, and interrupt settings. The initial drive state is either `GPIO_PIN_OUTPUT_HIGH` or `GPIO_PIN_OUTPUT_LOW`.

There is an enumerated set of values that can be used in combination to specify the configuration you want. These can be found in the GPIO documentation or the file *wiced_hal_gpio.h*. A partial list is shown here:

GPIO_EN_INT_LEVEL_HIGH	Interrupt on level HIGH.
GPIO_EN_INT_LEVEL_LOW	Interrupt on level LOW.
GPIO_EN_INT_RISING_EDGE	Interrupt on rising edge.
GPIO_EN_INT_FALLING_EDGE	Interrupt on falling edge.
GPIO_EN_INT_BOTH_EDGE	Interrupt on both edges.
GPIO_INPUT_ENABLE	Input enable.
GPIO_OUTPUT_DISABLE	Output disable.
GPIO_OUTPUT_ENABLE	Output enable.
GPIO_KS_OUTPUT_ENABLE	Keyscan output enable.
GPIO_OUTPUT_FN_SEL_MASK	Output function select mask.
GPIO_GLOBAL_INPUT_ENABLE	Global input enable.
GPIO_GLOBAL_INPUT_DISABLE	Global input disable.
GPIO_PULL_UP_DOWN_NONE	No pull [0,0].
GPIO_PULL_UP	Pull up [1,0].
GPIO_PULL_DOWN	Pull down [0,1].

As an example, if you have a pin connected to a mechanical button that pulls the pin low when pressed, you would configure it as an input with a resistive pullup. You would also set the initial drive state to 1 so that the resistive pullup path is activated:

```
wiced_hal_gpio_configure_pin(
      USER_BUTTON1,
      (GPIO_INPUT_ENABLE | GPIO_PULL_UP),
      GPIO_PIN_OUTPUT_HIGH );
```

If you want a falling edge interrupt on that same pin, it would look like this:

```
wiced_hal_gpio_configure_pin(
      USER_BUTTON1,
(GPIO_INPUT_ENABLE | GPIO_PULL_UP | GPIO_EN_INT_FALLING_EDGE),
      GPIO_PIN_OUTPUT_HIGH );
```

Once configured, input pins can be read using `wiced_hal_gpio_get_pin_input_status` and outputs can be driven using *`wiced_hal_gpio_set_pin_output`*. You can also get the state that an output pin is set to (not necessarily the actual value on the pin) using `wiced_hal_gpio_get_pin_output`.

As you learned in the previous chapter, the parameter for these functions is the pin name such as `WICED_P27` (from *wiced_hal_gpio.h*) or a functional name from your BSP such as `LED1` (from *cycfg_pin.h*) or `WICED_GPIO_PIN_LED_1` (from *wiced_platform.h*).

For example:

```
btnState = wiced_hal_gpio_get_pin_input_status( USER_BUTTON1 ); /* Get pin state */
wiced_hal_gpio_set_pin_output(LED2, 0); /* Set pin low */
wiced_hal_gpio_set_pin_output(LED2,
     !wiced_hal_gpio_get_pin_output(LED2) );  /* Invert desired pin state */
```

GPIO interrupts are enabled or disabled during pin configuration. For pins with interrupts enabled, the interrupt callback function (i.e. interrupt service routine or interrupt handler) is registered using `wiced_hal_gpio_register_pin_for_interrupt`. For example, the following would enable a falling edge interrupt on `BUTTON1` with a callback function called `my_interrupt_callback`.

```
wiced_hal_gpio_register_pin_for_interrupt( USER_BUTTON1,
   my_interrupt_callback, NULL);

wiced_hal_gpio_configure_pin( USER_BUTTON1,
  (GPIO_INPUT_ENABLE | GPIO_PULL_UP | GPIO_EN_INT_FALLING_EDGE),
  GPIO_PIN_OUTPUT_HIGH);
```

The interrupt callback function is passed user data (optional) and the number of the pin that generated the interrupt. The callback function should clear the interrupt using `wiced_hal_gpio_clear_pin_interrupt_status`. For example:

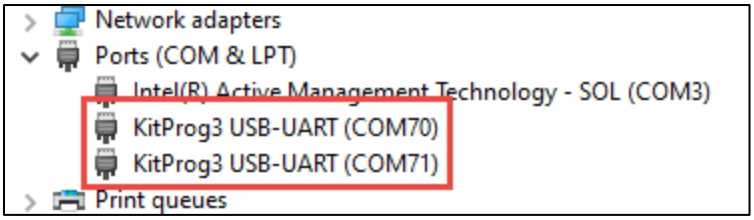
```
void my_interrupt_callback(void *data, uint8_t port_pin)
{
   /* Clear the gpio interrupt */
   wiced_hal_gpio_clear_pin_interrupt_status( port_pin );

   /* Add other interrupt functionality here */
}
```

Note: *The call to `wiced_hal_gpio_clear_pin_interrupt_status` is shown in the code above for completeness. For most peripherals it is necessary to clear the interrupt in the callback function. However, for GPIOs this is done automatically before the callback is executed and so it is not strictly necessary.*

3.2.2 Debug Printing

You will use this in Exercise 2:

As mentioned previously, the kit has two separate UART interfaces: the HCI UART (Host Controller Interface UART) and the PUART (peripheral UART). The HCI UART interface is used for programming the kit and is sometimes used by a host microcontroller to communicate with the Bluetooth® LE device. It will be discussed in more detail later. The PUART is not used for any other specific functions so it is useful for general debug messages. When you plug a kit into your USB port both UART channels appear as COM ports. If you attach a kit to a Windows machine, the Device Manager will display entries that look something like this:



The PUART is usually the smaller of the two COM port numbers.

There are 3 things required to allow debug print messages:

1. Make sure that the symbol `WICED_BT_TRACE_ENABLE` is defined in the project's *makefile*. For example:

   ```
   CY_APP_DEFINES+= -DWICED_BT_TRACE_ENABLE
   ```

   The provided starter templates all set this up automatically, so editing the *makefile* for this is not usually necessary.

2. Include the following header in the top-level C file if it's not already there:

   ```
   #include "wiced_bt_trace.h"
   ```

3. Indicate which interface you want to use by choosing one of the following. Typically, we will use the PUART (the 2nd one in the list).

   ```
   wiced_set_debug_uart( WICED_ROUTE_DEBUG_NONE);
   wiced_set_debug_uart( WICED_ROUTE_DEBUG_TO_PUART );
   wiced_set_debug_uart( WICED_ROUTE_DEBUG_TO_HCI_UART );
   wiced_set_debug_uart( WICED_ROUTE_DEBUG_TO_WICED_UART);
   ```

The last of these is used for sending formatted debug strings over the HCI interface specifically for use with the BTSpy application. The BTSpy application will be discussed in detail in the debugging chapter.

Once the appropriate debug UART is selected, messages can be sent using `printf`-type formatting using the `WICED_BT_TRACE` function. For example:

```
WICED_BT_TRACE( "Hello - this is a debug message \n");
WICED_BT_TRACE("The value of X is: %d\n", x);
```

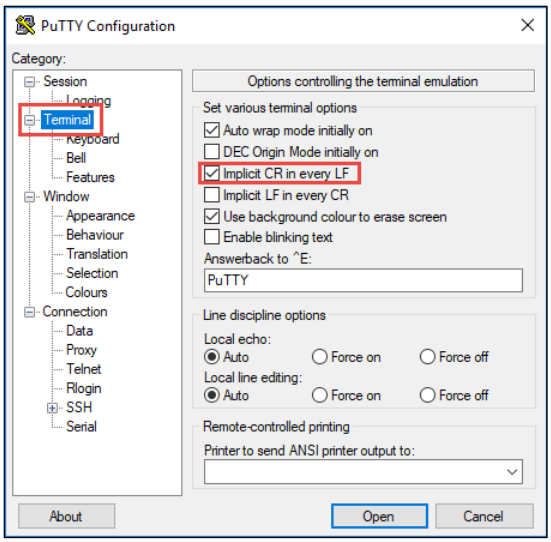
Note: *This function does NOT support floating point values (i.e. %f).*

You can easily print arrays using `WICED_BT_TRACE_ARRAY`. The usage is:

```
WICED_BT_TRACE_ARRAY(arrayName, arrayLength, "String to be printed before the
array data: ");
```

The `WICED_BT_TRACE_ARRAY` function automatically adds a newline (\n) to the end of the printed string.

We typically put "\n" at the end of strings to be printed but not "\r" to save flash and ram space. Therefore, you will want to set up your terminal window to automatically generate a carriage return for every line feed so that each debug message will start at the beginning of the line. In PuTTY, the setting is under **Terminal > Implicit CR in every LF**. You can save this to the default settings if you want it to be on by default (you can also save the default speed to be 115200).



3.2.2.1 Printing to a String

You can print to a string (like `snprintf`) by using the function `wiced_printf`. This can be useful if you are using an external display such as an OLED. The arguments are:

1. A pointer to a `uint8_t` array to hold the output string.
2. The max number of characters to output.
3. The formatting template string just like in `snprintf`.
4. Any values to be substituted into the formatting template, just like `snprintf`.

For example:

```
uint8_t message[50];
wiced_printf (message, 50, "The value of X is: %d", x);
```

As with debug traces, this function does not support floating point values (i.e. %f).

If you want to operate on floating point values or cast floats to decimals, you must add the following to the project's *makefile*:

```
CY_RECIPE_EXTRA_LIBS+=-lgcc
```

3.2.3 PUART (Peripheral UART)

You will use this in Exercise 10: and Exercise 11:

In addition to the debug printing functions, the PUART can be used as a generic Tx/Rx UART block. To use it, first include the header file in your top-level C file:

```
#include "wiced_hal_puart.h"
```

Next, initialize the block and configure the baud rate, parity, stop bits, and flow control. For example:

```
wiced_hal_puart_init( );
wiced_hal_puart_configuration(115200, PARITY_NONE, STOP_BIT_1 );
wiced_hal_puart_flow_off( );
```

For transmitting data, enable Tx, and then use the desired functions for sending strings (print), single bytes (write), or an array of bytes (synchronous_write).

```
wiced_hal_puart_enable_tx( );
wiced_hal_puart_print("Hello World!\n");
/* Print value to the screen */
wiced_hal_puart_print("Value = ");
/* Add '0' to the value to get the ASCII equivalent of the number */
wiced_hal_puart_write(value+'0');
wiced_hal_puart_print("\n");
```

For receiving data, register an interrupt callback function, set the watermark to determine how many bytes should be received before an interrupt is triggered, and enable Rx.

```
wiced_hal_puart_register_interrupt(rx_interrupt_callback);
/* Set watermark level to 1 to receive interrupt up on receiving each byte */
wiced_hal_puart_set_watermark_level(1);
wiced_hal_puart_enable_rx();
```

The Rx processing is done inside the interrupt callback function. You must clear the interrupt inside the callback function so that additional characters can be received.

```
void rx_interrupt_callback(void* unused)
{
    uint8_t readbyte;

    /* Read one byte from the buffer and then clear the interrupt */
    wiced_hal_puart_read( &readbyte );
    wiced_hal_puart_reset_puart_interrupt();

    /* Add your processing here */
}
```

3.2.4 Timers

You will use this in Exercise 5:

A timer allows you to schedule a function to run at a specified interval; e.g., send data every 10 seconds.

First, you must include *wiced_timer.h* in your source code. Then, you initialize the timer using `wiced_init_timer` with a pointer to a timer structure, the function you want to run, an argument to the function (or `NULL` if you don't need it), and the timer type. There are four types of timer. The first two are one-shot timers, while the last two will run continuously:

WICED_SECONDS_TIMER
WICED_MILLI_SECONDS_TIMER
WICED_SECONDS_PERIODIC_TIMER
WICED_MILLI_SECONDS_PERIODIC_TIMER

The function that you specify takes a single argument: `uint32_t arg`. If the function doesn't require any arguments, you can specify 0 in the timer initialization function, but the function itself must still have the `uint32_t arg` argument in its definition.

After you initialize the timer, you then start it using `wiced_start_timer`. This function takes a pointer to the timer structure and the actual time interval for the timer (either in seconds or milliseconds depending on the timer chosen).

Note: *This is an interrupt function for when the timer expires rather than a continually executing thread so the function should NOT have a while(1) loop; it should just run and exit each time the timer calls it.*

For example, to set up a timer that runs a function called `myTimer` every 100ms, you would do something like this:

```
wiced_timer_t my_timer_handle; /* Typically defined as a global */
.
.
.
.
/* Typically inside the BTM_ENABLED_EVT */
wiced_init_timer(&my_timer_handle, myTimer, 0,
WICED_MILLI_SECONDS_PERIODIC_TIMER);
wiced_start_timer(&my_timer_handle, 100);
.
.
.
.
/* The timer function */
void myTimer( uint32_t arg )
{
    /* Put timer code here */
}
```

3.2.5 PWM

You will use this in Exercise 6: and Exercise 7:

There are 6 PWM blocks (PWM0 – PWM5) on the device, each of which can be routed to any GPIO pin. The PWMs are 16 bits (i.e. they count from 0 to 0xFFFF).

The PWMs can use either the LHL_CLK (which is 32 kHz) or PMU_CLK (aka ACLK1) which is configurable.

First, you must include the PWM header file to use the PWM API functions:

```
#include "wiced_hal_pwm.h"
```

Then, you can enable the PWM and assign its output to one of the LED pins using the following function:

```
wiced_hal_gpio_select_function(LED2, WICED_PWM0);
```

The pin name used above is defined by the Device Configurator, and the peripheral name is defined in *wiced_hal_gpio.h* (in the *wiced_btsdk/dev-kit/baselib/<device>/<version>/COMPONENT_<device>/include/hal* directory).

The function that you want to select (`WICED_PWM0` in this case) is a selection from a mux that allows different on-chip peripherals to connect to the GPIO pins. The possible values come from an enumeration of type `wiced_bt_gpio_function_t`, which is found in `wiced_hal_gpio.h`.

When you want to start the PWM, just call the start function like this:

```
wiced_hal_pwm_start( PWM0, LHL_CLK, toggleCount, initCount, 0 );
```

The value of `PWM0` is the name of the PWM you are starting, which is specified in *wiced_hal_pwm.h*.

The `initCount` parameter is the value that the PWM will reset to each time it wraps around. For example, if you set `initCount` to (0xFFFF – 99), then the PWM will provide a period of 100 counts.

The `toggleCount` parameter is the value at which the PWM will switch its output from high to low. That is, it will be high when the count is less than the `toggleCount` and will be low when the count is greater than the `toggleCount`. For example, if you set the `toggleCount` to (0xFFFF-50) with the period set as above, then you will get a duty cycle of 50%.

You can invert the PWM output (i.e. it will start low and then transition high at the `toggleCount`) by setting the last parameter to 1 instead of 0.

Similarly, to change the PWM parameters while it is running, use this function:

```
wiced_hal_pwm_change_values( PWM0, toggleCount, initCount );
```

There is a helper function that will calculate the `initCount` and `toggleCount` required to achieve a desired frequency and duty cycle. That function's prototype is:

```
wiced_hal_pwm_get_params( uint32_t clock_frequency_in, uint32_t duty_cycle,
uint32_t pwm_frequency_out, pwm_config_t * params_out);
```

You provide the input clock to the PWM, the desired duty cycle (0 – 99), and desired frequency. The function will fill the `params_out` structure with the values needed for the PWM functions to start or change values. The structure looks like this:

```
typedef struct{
    UINT32 init_count;
    UINT32 toggle_count;
} pwm_config_t;
```

If you want a specific clock frequency for the PWM, you must first configure the PMU_CLK clock and then specify it in the PWM start function.

First, you have to include an additional header file:

```
#include "wiced_hal_aclk.h"
```

Then, if you (for example) want a 1 kHz clock for the PWM, you could do the following:

```
#define CLK_FREQ    (1000)

wiced_hal_aclk_enable(CLK_FREQ, ACLK1, ACLK_FREQ_24_MHZ );
wiced_hal_pwm_start(PWM0, PMU_CLK, toggleCount, initCount, 0);
```

Note: *There is only 1 PMU clock available. If you use it, you will get the same clock frequency for all PWMs that use it as the source.*

There are additional functions to enable, disable, get the init value, and get the toggle count. See the documentation for details on each of these functions.

3.2.6 NVRAM

You will use this in Exercise 8:

There are many situations in a Bluetooth® system where a non-volatile memory is required. One example of that is Bonding – which we will discuss in detail later - where you are required to save the Link Keys for future use. The BTSDK provides an abstraction called the "NVRAM" (it is really just an area of flash memory that is set aside) for this purpose. The chip configuration typically allocates 4 kB to 8 kB for the NVRAM, but it is user-modifiable. The API and programming model remain the same regardless of the total NVRAM size.

The NVRAM is broken into multiple sections that are up to 512 bytes long (on the 20835 device), of which 500 are available for user data. To use the NVRAM, you write to/read from a number called the VSID (Virtual System Identifier). This number is not an offset or a block number within the memory. Rather, it is an ID that the API uses to locate the actual memory.

Physical addresses are not used because the NVRAM has a wear-leveling scheme built in that moves the data around to avoid wearing out the memory. The VSID allows you to use the NVRAM without needing to worry about the physical location of data and the wear-leveling scheme. The only cost of this implementation is that reads and writes to NVRAM take a variable amount of time.

Note: *The wear-leveling scheme also has a "defragmentation" algorithm that runs during chip boot-up.*

As the developer, you are responsible for managing what the VSIDs are used for in your application.

The API can be included in your application with `#include "wiced_hal_nvram.h"`, which also `#defines` the first VSID to be `WICED_NVRAM_VSID_START` and last VSID to be `WICED_NVRAM_VSID_END`.

The write function for the NVRAM is:

```
uint16_t wiced_hal_write_nvram( uint16_t vs_id, uint16_t data_length, uint8_t *p_data,
         wiced_result_t * p_status);
```

The return value is the number of bytes written. You need to pass a pointer to a `wiced_result_t`, which will give you the success or failure of the write operation.

The read function for the NVRAM looks just like the write function:

```
uint16_t wiced_hal_read_nvram( uint16_t vs_id,uint16_t data_length, uint8_t * p_data,
       wiced_result_t * p_status);
```

The return value is the number of bytes read into your buffer, and `p_status` tells you if the read succeeded.

Note: *If you read from a VSID that has not been written to (since the device was programmed), the read will return with a failing error code. This is useful to determine if a device already has information (such as bonding information) stored.*

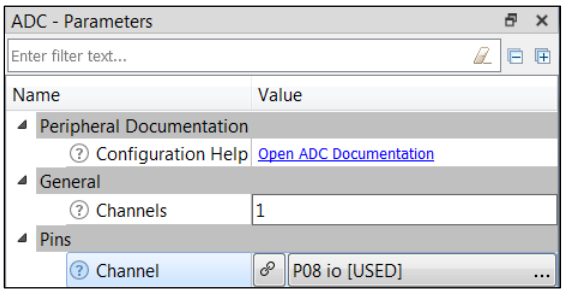
3.2.7 ADC

You will use this in Exercise 9:

The device contains a 16-bit signed ADC (-32768 to +32767). The ADC has 32 input channels, all of which have fixed connections to pins or on-chip voltages such as Vddio. When setting up the ADC in the Device Configurator, you choose the number of channels and assign each one to the desired physical pin.

Note: *This is not routing signals on the device, but merely assigning virtual channel numbers to the pins.*

The voltage channels, like Vddio, are configured automatically for you. The ADC is enabled and set to P08 (thermistor) by default.



We will be measuring P08 (the thermistor) in the exercises, so we won't need to make any Device Configurator changes.

In either case, you must include the ADC header file to use the ADC functions:

```
#include "wiced_hal_adc.h"
```

To initialize the ADC block, you need to call the initialization function and select the input range based on your board supply. When you read a sample, you must specify which channel to read from. The BSP provides enums of the form `ADC_INPUT_*` for the channels in *wiced_hal_adc.h*. Examples of channel defines are `ADC_INPUT_P8` (ADC channel 9 is connected to pin P08 on the device) and `ADC_INPUT_VDDIO`.

There is one function that will return a count value and another function that will return a voltage value in millivolts. For example, to read the count and voltage from a sensor which is connected to GPIO `WICED_P10`, you would do the following:

```
#define ADC_CHANNEL      (ADC_INPUT_P10)
wiced_hal_adc_init();
wiced_hal_adc_set_input_range( ADC_RANGE_0_3P6V );
raw_val = wiced_hal_adc_read_raw_sample( ADC_CHANNEL, 0 );
voltage_val = wiced_hal_adc_read_voltage( ADC_CHANNEL );
```

3.2.8 RTC (Real Time Clock)

You will use this in Exercise 12:

The AIROC™ CYW20835 device supports a 48-bit RTC timer referenced to a 32-kHz crystal (XTAL32K) LPO (low power oscillator). The LPO can be either external or internal. If an external LPO is not connected to the device, the firmware takes the clock input from the internal LPO for the RTC. The device supports both 32-kHz and 128-kHz LPOs, but the internal defaults to 32-kHz.

The BTSDK provides API functions to set the current time, get the current time, and convert the current time value to a string. By default, the date and time are set to January 1, 2010 with a time of 00:00:00 denoting HH:MM:SS.

You must include *wiced_rtc.h* and the following code to initialize the RTC for use:

```
wiced_rtc_init();
wiced_rtc_time_t newTime = { 0, 30, 12, 30, 11, 1999 }; // s,m,h,d,m,y

wiced_set_rtc_time( &newTime );
```

Note that the day and month are 0-based and so, for example, January is 0 and December is 11. Likewise, the first day of the month is 0.

After the RTC is initialized, you can use `wiced_rtc_get_time` to get the time and `wiced_rtc_ctime` to convert the result into a printable string.

3.3 WICED_RESULT_T

A value is returned from many of the API functions to tell you what happened. The return value is of the type `wiced_result_t`, which is a giant enumeration. You can find the values in the documentation in the section **Components > AIROC System > AIROC Result Codes**. Alternately, if you right-click on `wiced_result_t` from a variable declaration, select **Open Declaration**, you will see this:

```
/** WICED result */
typedef enum
{
    WICED_RESULT_LIST(WICED_)
    BT_RESULT_LIST      (  WICED_BT_      )  /**< 8000 - 8999 */
} wiced_result_t;
```

As you can see, there are two sets of codes: one for general function results and one for Bluetooth® function results. To see general return codes (`WICED_*`), right-click and choose **Open Declaration** on `WICED_RESULT_LIST`. For Bluetooth® specific return codes (`WICED_BT_*`), right-click and choose **Open Declaration** on `BT_RESULT_LIST`. The partial lists look like this:

`WICED_RESULT_LIST:`

```
/** WICED result list */
#define WICED_RESULT_LIST( prefix ) \
    RESULT_ENUM( prefix, SUCCESS,                    0x00 ),   /**< Success */
    RESULT_ENUM( prefix, DELETED                    ,0x01 ),   \
    RESULT_ENUM( prefix, NO_MEMORY                  ,0x10 ),   \
    RESULT_ENUM( prefix, POOL_ERROR                 ,0x02 ),   \
    RESULT_ENUM( prefix, PTR_ERROR                  ,0x03 ),   \
    RESULT_ENUM( prefix, WAIT_ERROR                 ,0x04 ),   \
    RESULT_ENUM( prefix, SIZE_ERROR                 ,0x05 ),   \
    RESULT_ENUM( prefix, GROUP_ERROR                ,0x06 ),   \
    RESULT_ENUM( prefix, NO_EVENTS                  ,0x07 ),   \
    RESULT_ENUM( prefix, OPTION_ERROR               ,0x08 ),   \
    RESULT_ENUM( prefix, QUEUE_ERROR                ,0x09 ),   \
    RESULT_ENUM( prefix, QUEUE_EMPTY                ,0x0A ),   \
    RESULT_ENUM( prefix, QUEUE_FULL                 ,0x0B ),   \
    RESULT_ENUM( prefix, SEMAPHORE_ERROR            ,0x0C ),   \
    RESULT_ENUM( prefix, NO_INSTANCE                ,0x0D ),   \
    RESULT_ENUM( prefix, THREAD_ERROR               ,0x0E ),   \
    RESULT_ENUM( prefix, PRIORITY_ERROR             ,0x0F ),   \
    RESULT_ENUM( prefix, START_ERROR                ,0x10 ),   \
```

`BT_RESULT_LIST:`

```
#define BT_RESULT_LIST( prefix ) \
        RESULT_ENUM( prefix, SUCCESS,                     0 ),   /**< Success */
        RESULT_ENUM( prefix, PARTIAL_RESULTS,             3 ),   /**< Partial results */
        RESULT_ENUM( prefix, BADARG,                      5 ),   /**< Bad Arguments */
        RESULT_ENUM( prefix, BADOPTION,                   6 ),   /**< Mode not supported */
        RESULT_ENUM( prefix, OUT_OF_HEAP_SPACE,           8 ),   /**< Dynamic memory space exhausted */
        RESULT_ENUM( prefix, UNKNOWN_EVENT,            8029 ),   /**< Unknown event is received */
        RESULT_ENUM( prefix, LIST_EMPTY,               8010 ),   /**< List is empty */
        RESULT_ENUM( prefix, ITEM_NOT_IN_LIST,         8011 ),   /**< Item not found in the list */
        RESULT_ENUM( prefix, PACKET_DATA_OVERFLOW,     8012 ),   /**< Data overflow beyond the packet end
        RESULT_ENUM( prefix, PACKET_POOL_EXHAUSTED,    8013 ),   /**< All packets in the pool is in use *
        RESULT_ENUM( prefix, PACKET_POOL_FATAL_ERROR,  8014 ),   /**< Packet pool fatal error such as per
        RESULT_ENUM( prefix, UNKNOWN_PACKET,           8015 ),   /**< Unknown packet */
        RESULT_ENUM( prefix, PACKET_WRONG_OWNER,       8016 ),   /**< Packet is owned by another entity *
        RESULT_ENUM( prefix, BUS_UNINITIALISED,        8017 ),   /**< Bluetooth bus isn't initialised */
        RESULT_ENUM( prefix, MPAF_UNINITIALISED,       8018 ),   /**< MPAF framework isn't initialised */
        RESULT_ENUM( prefix, RFCOMM_UNINITIALISED,     8019 ),   /**< RFCOMM protocol isn't initialised *
        RESULT_ENUM( prefix, STACK_UNINITIALISED,      8020 ),   /**< SmartBridge isn't initialised */
        RESULT_ENUM( prefix, SMARTBRIDGE_UNINITIALISED, 8021 ),   /**< Bluetooth stack isn't initialised *
        RESULT_ENUM( prefix, ATT_CACHE_UNINITIALISED,  8022 ),   /**< Attribute cache isn't initialised *
        RESULT_ENUM( prefix, MAX_CONNECTIONS_REACHED,  8023 ),   /**< Maximum number of connections is re
        RESULT_ENUM( prefix, SOCKET_IN_USE,            8024 ),   /**< Socket specified is in use */
        RESULT_ENUM( prefix, SOCKET_NOT_CONNECTED,     8025 ),   /**< Socket is not connected or connecti
        RESULT_ENUM( prefix, ENCRYPTION_FAILED,        8026 ),   /**< Encryption failed */
```

Note: The file containing the return code list shown above can be found at:

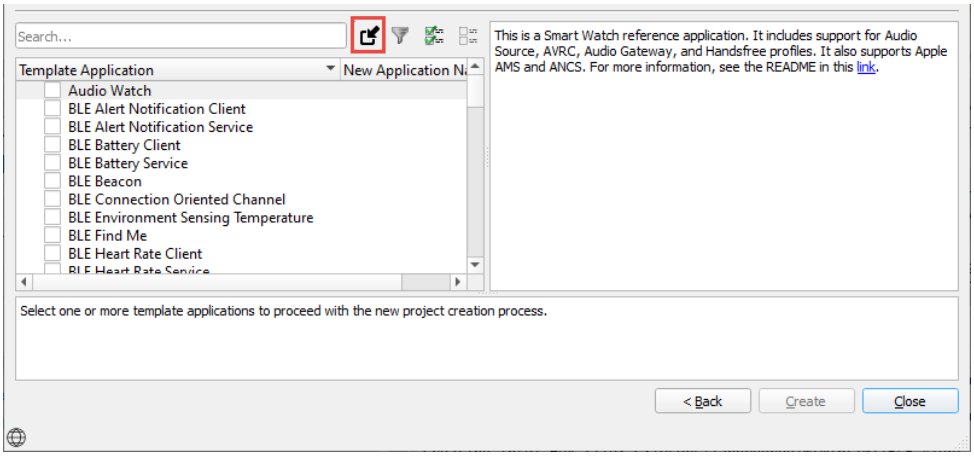
wiced_btsdk/dev-kit/baselib/<device>/<version>/COMPONENT_<device>/include/wiced_result.h.

3.4 Exercises

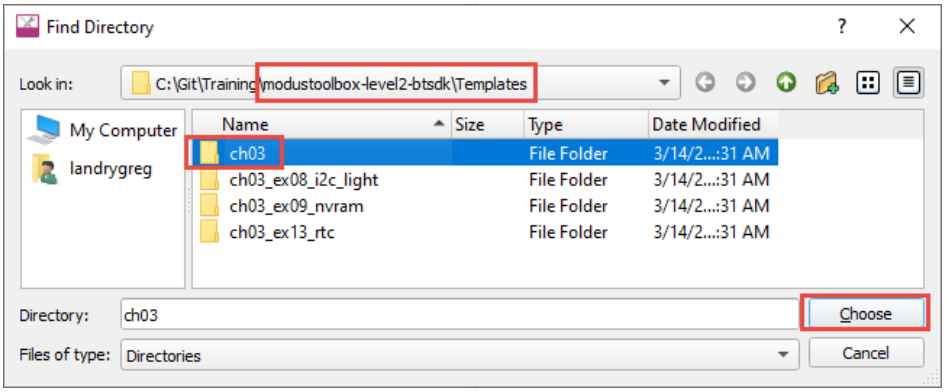
Exercise 1: (GPIO) Blink an LED

In this exercise, you will create an application to blink LED2 on the kit at 2 Hz. This material is covered in 3.2.1.

1. Launch the Project Creator tool either in stand-alone mode or from the Eclipse IDE Quick Panel.
2. Select the CYW920835M2EVB-01 kit and click **Next >**.
3. In this case, we will provide a template for you to use called *ch03* instead of one of the built-in templates. Click the **Import** button and navigate to the course materials *templates/ch03* directory.



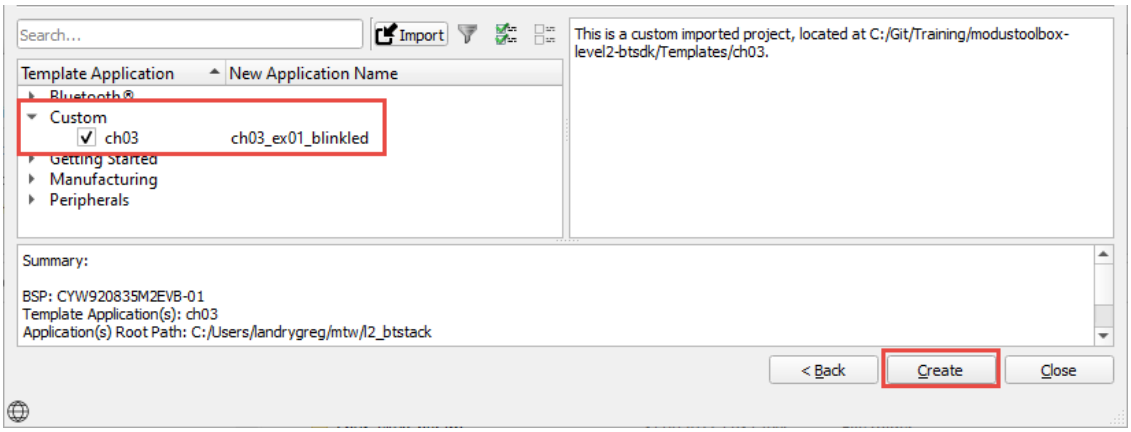
4. Click **Choose**.



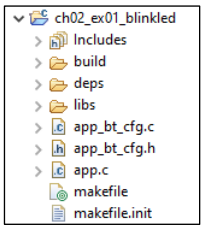
5. The new template will now appear in the list in the **Custom** category. Check the box next to it.

6. Change the application name to **ch03_ex01_blinkled** and click **Create**.

Note: *You can call the application anything you like but it will really help if you maintain an alphabetically sortable naming scheme – you are going to create quite a few applications in this course.*



7. After the application has been created it will be imported into the IDE. When it is finished, the project directory will look like the following:



8. Examine *app.c* to make sure you understand what it does.

   All WICED Bluetooth® LE applications are multi-threaded (the Bluetooth® LE stack requires it). There is an operating system (RTOS) that gets launched from the device startup code and you can use it to create your own threads. Each thread has a function that runs almost as though it is the only software in the system – the RTOS allocates time for all threads to execute when they need to. This makes it easier to write your programs without a lot of extra code in your main loop. The details of how to use the RTOS effectively are covered in the next chapter. However, in these exercises, we will show you how to create a thread and associate it with a function for the code you will write (look in `app_bt_management_callback()`).

9. Add code in the `app_task` thread function to do the following:

   Read the state of LED1. Remember, it is an output pin, not an input pin.

Note: *Go back to the section on GPIOs if you need a reminder on using pins.*

   Drive the state of LED1 to the opposite value.

10. In the Quick Panel, click the **ch03_ex01_blinkled Program** link in the "Launches" section.

Questions

1. What is the name of the first user application function that is executed? What does it do?

2. What is the purpose of the function `app_bt_management_callback`? When does the `BTM_ENABLED_EVT` case occur?

3. What controls the rate of the LED blinking?

Exercise 2: (GPIO) Add Debug Printing to the LED Blink Application

For this exercise, you will add a message that is printed to a UART terminal each time the LED changes state. This material is covered in 3.2.2.

☐ 1. Instead of starting from scratch, crate a new application called **ch03_ex02_blinkled_print** by importing the previously completed ch03_ex01_blinkled exercise as the template application.

Note: *If you did not complete the previous exercise, you can import the key_ch03_ex01_blinkled solution project instead.*

☐ 2. Add `WICED_BT_TRACE` calls to display "LED LOW" and "LED HIGH" at the appropriate times.

Note: *Go back to the section on Debug Printing if you need a refresher.*

Note: *Remember to set the debug `UART` to `WICED_ROUTE_DEBUG_TO_PUART`. Although you will see comments in the template code encouraging you to put initialization code in the `app_bt_management_callback` function so that it runs when the Bluetooth LE stack starts up, we recommend doing it in `application_start` instead. This is because you may want to print messages before even trying to start the stack!*

Note: *Remember to use `\n\r` (or at least `\n`) to create a new line so that information is printed on a new line each time the LED changes.*

☐ 3. Program your application to the board.

☐ 4. Open a terminal window (e.g. PuTTY or TeraTerm) with a baud rate of 115200 and observe the messages being printed.

Note: *If you look at the output during programming, you can determine which UART is used for programming. The PUART will be the other one of the two KitProg3 COM ports.*

Note: *If you don't have terminal emulator software installed, you can use putty.exe which is free to download. To configure putty:*

- Go to the **Serial** tab, select the correct COM port (you can get this from the device manager under "Ports (COM & LPT)" as *"WICED USB Serial Port"*), and set the speed to 115200.
- Go to the **Session** tab, select the **Serial** button, and click on **Open**.
- If you want an automatic carriage return with a line feed in putty (i.e. add a \r for every \n) check the box next to **Terminal > Implicit CR in every LF**

Exercise 3: (GPIO) Read the State of a Mechanical Button

In this exercise, you will control an LED by monitoring the state of a mechanical button on the kit. This material is covered in 3.2.1.

- [ ] 1. Create a new application called **ch03_ex03_button** again using the completed ch03_ex01_blinkled exercise as the template.
- [ ] 2. In the C file:
    a. Change the thread sleep time to 100ms.
    b. In the thread function, check the state of mechanical button input (use `USER_BUTTON1`). Turn on LED1 if the button is pressed and turn it off if the button is not pressed.
- [ ] 3. Program your application to the board and test it.

Exercise 4: (GPIO) Use an Interrupt to Toggle the State of an LED

In this exercise, rather than polling the state of the button in a thread, you will use an interrupt so that your firmware is notified every time the button is pressed. In the interrupt callback function, you will toggle the state of the LED. This material is covered in 3.2.1.

- [ ] 1. Create a new application called **ch03_ex04_interrupt** with the ch03 template.
- [ ] 2. In the *app.c* file:
    a. Remove the calls to `wiced_rtos_create_thread` and `wiced_rtos_init_thread`.
    b. Delete the thread function.
- [ ] 3. In the `BTM_ENABLED_EVT`, set up a falling edge interrupt for the GPIO connected to the button and register the callback function.

Note: You will need to call wiced_hal_gpio_configure_pin and wiced_hal_gpio_register_pin_for_interrupt.

- [ ] 4. Create the interrupt callback function so that it toggles the state of the LED each time the button is pressed.
- [ ] 5. Program your application to the board and test it.

Exercise 5: (Timer) Use a Timer to Toggle an LED

In this exercise, you use a timer to blink an LED. This material is covered in 3.2.4.

1. Create a new application called **ch03_ex05_timer.** This time, use the completed ch03_ex04_interrupt exercise as the template.
2. In the C file:
    a. Add an include for *wiced_timer.h*.
    b. In the `BTM_ENABLED_EVT`, add calls to initialize and start a periodic timer with a 250ms interval.
    c. Modify the interrupt callback function to serve as the timer callback.

Note: *The body of the timer function is the same as the code you wrote for the interrupt callback, but the function argument list is slightly different.*

3. Program your application to the board and test it.

Exercise 6: (PWM) LED brightness

In this exercise, you will control an LED using a PWM instead of a GPIO. The PWM will toggle the LED too fast for the eye to see, but by controlling the duty cycle you will vary the apparent brightness of the LED. This material is covered in 3.2.5.

1. Create a new application called **ch03_ex06_pwm** using the ch03 template.
2. In the app.c file:
    a. Add a `#include` for the PWM functions - `"wiced_hal_pwm.h"`
    b. Configure the GPIO mux to make a connection from `LED1` to `WICED_PWM0`.
    c. Configure `PWM0` with an initial frequency of 500 Hz and a duty cycle of 50%.

Note: *This can be done in the `BTM_ENABLED_EVT` event but be sure to define the PWM configuration structure as a global so it can be used in the application thread.*

Note: *Use LHL_CLK as the source clock since the exact period of the PWM doesn't matter as long as it is faster than the human eye can see (~50 Hz). The LHL clock is 32000 Hz.*

3. Update the duty cycle in the thread function so that the LED gradually cycles through intensity values from 0 to 100%.

Note: *Change the delay in the thread function to 10ms so that the brightness changes relatively quickly.*

4. Program the application to the board and test it.

Exercise 7: (PWM) LED toggling at specific frequency and duty cycle

In this exercise, you will use a PWM with a period of 1 second and a duty cycle of 10% so that the LED will blink at a 1 Hz rate but will only be on for 100ms each second. This material is covered in 3.2.5.

1. Create a new application called **ch03_ex07_pwm_blink** using the completed ch03_ex06_pwm exercise as the template.
2. In the *app.c* file:
    a. Remove the code from the thread function that updates the PWM duty cycle. The thread function should only have an RTOS delay in it.
    b. Initialize the `aclk` with a frequency of 1 kHz.
    c. Change the PWM0 configuration to use `PMU_CLK` as the source. Set the frequency to 1 Hz and set the duty cycle to 10%.

Note: Don't forget to include the header file for the ACLK functions.

3. Program the application to the board and test it.

Exercise 8: (NVRAM) Write and Read Data in the NVRAM

In this exercise, you will store a 1-byte value in the NVRAM. The button `USER_BUTTON1` will increment the value each time it is pressed and print the new value. This material is covered in 3.2.6.

1. Create another new application called **ch03_ex08_nvram** using the template in *templates/**ch03_ex08_nvram*** (NOT the I2C template).
2. Add a `#include` for the NVRAM API functions.
3. Call `wiced_set_debug_uart` with the appropriate parameter to use the PUART.
4. In the button interrupt callback function:
    a. Read the value from NVRAM at location `WICED_NVRAM_VSID_START`.
    b. Print out the value, the number of bytes read, and the status of the read operation (not the return value).
    c. Increment the value.
    d. Write the new value into NVRAM at location `WICED_NVRAM_VSID_START`.
    e. Print out the value, the number of bytes written, and the status of the write operation.
5. Open a terminal window and program the kit. Wait a few seconds and then press Button 1 a few times to observe the results.
6. Reset the kit. Press the button and notice that the previously stored value is retained.
7. Unplug the kit, plug it back in, and reset the terminal. Press the button and notice that the previously stored value is retained.

Questions

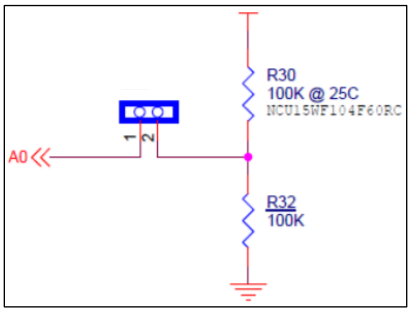
1. How many bytes does the NVRAM read function get when you press the button the first time?

2. What is the return status value when you press the button the first time?

3. What does the return value mean?

Exercise 9: (ADC) Calculate the resistance of a thermistor

In this exercise you will measure the voltages, in millivolts, of Vddio (supply) and across the thermistor balance resistor (100kOhm). Then you will use that data to calculate the resistance of the thermistor. For reference, here is the thermistor circuit. This material is covered in 3.2.7.



The thermistor shares pin P08 with A0 on the Arduino header of the CYW920835M2EVB-01. You need to make sure the THERMISTOR_ENABLE jumper is attached to read the voltage.

1. Create a new application called **ch03_ex09_adc** using the template in templates/**ch03** (this is the original template again).
2. Add in the include file for the ADC functions.
3. Add the call to `wiced_set_debug_uart` to enable printing to the PUART.
4. In the management callback function, initialize the ADC.
5. In the application thread function, use the ADC to read Vddio and the voltages across the balance resistor.

Note: The ADC has a dedicated channel to read the supply voltage called `ADC_INPUT_VDDIO`.

6. Calculate and print the resistance of the thermistor periodically.

Note: The formula is as follows.

$$Rtherm = \frac{(Vddio - Vmeas) * BALANCE\_RESISTANCE}{Vmeas}$$

7. Program the board and open a terminal window with a baud rate of 115200.
8. Alternately place and remove your finger from the thermistor (next to the THERMISTOR_ENABLE jumper) and observe the value displayed in the terminal.

Exercise 10: (UART) Send a value using the standard UART functions

In this exercise, you will use the standard UART functions to send a value to a terminal window. The value will increment each time a mechanical button on the kit is pressed. This material is covered in 3.2.3.

1. Create a new application called **ch03_ex10_uartsend** using the completed ch03_ex04_interrupt exercise as the template.
2. In *app.c*, remove the call to `wiced_set_debug_uart` if your interrupt exercise used the PUART.
3. Initialize the UART with Tx enabled, baud rate of 115200, and no flow control.
4. Modify the interrupt callback so that each time the button is pressed a variable is incremented and the value is sent out over the UART. 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.

   Note: Use a "static" variable in the callback function to remember the last number printed.

5. Program the board and open a terminal window with a baud rate of 115200. Press the button and observe the value displayed in the terminal.

Exercise 11: (UART) Get a value using the standard UART functions

In this exercise, you will learn how to read a value from the UART rather than sending a value like in the previous exercise. The value entered will be used to control an LED on the board (0 = OFF, 1 = ON). This material is covered in 3.2.3.

1. Create a new application called **ch03_ex11_uartreceive** using the completed ch03_ex10_uartsend exercise as the template.
2. Update the code to initialize the UART with Rx enabled, baud rate of 115200, no flow control, and an interrupt generated on every byte received.

   Note: You can remove the code for the button press and its interrupt, but you will need to register a UART Rx interrupt callback instead.

3. In the interrupt callback, read the byte. If the byte is a 1, turn on LED2. If the byte is a 0, turn off the LED. Ignore any other characters.
4. Program your application to the board.
5. Open a terminal window with a baud rate of 115200.
6. Press the 1 and 0 keys on the keyboard and observe the LED turn on/off.

Exercise 12: (RTC) Display Time and Date Data on the UART

In this exercise you will set the time and date after reset and thereafter display a clock on the UART. This material is covered in 3.2.8.

1. Create a new application called **ch03_ex12_rtc** using the template in *templates/**ch03_ex12_rtc*** (different template).
2. In *app.c*, look for TODO comments and add code to control the RTC as follows.
    a. Initialize the RTC.
    b. Set the time and date. Make sure you understand how the `getDateTimeEntry` function works and gets used.

Note: *The template code includes `ring_pop` and `ring_push` functions that implement a ring buffer for the UART – the UART interrupt code pushes received characters into the buffer and the code that sets the time and date pulls characters from it.*

    c. Display the clock with a precision of 1s.

Note: *Use the `wiced_hal_puart_print` function to print the time once you have converted it to a string.*

3. Program your application to the board and test it.

Note: *Enter the time and date as indicated. Once you are done, the clock will start running.*

3.5 Appendix

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

3.5.1 Exercise 1 Answers

1. What is the name of the first user application function that is executed? What does it do?

   The first user function is `application_start`.

   It just initializes the Bluetooth® stack and registers the Bluetooth® stack callback function.

2. What is the purpose of the function `app_bt_management_callback`? When does the `BTM_ENABLED_EVT` case occur?

   It is the Bluetooth® stack management callback function. It is called whenever there is a management event from the stack.

   The `BTM_ENABLED_EVT` case occurs once the stack has completed initialization.

3. What controls the rate of the LED blinking?

   The first parameter to the RTOS delay function `wiced_rtos_delay_milliseconds` specifies the delay which controls the rate of the LED blinking.

3.5.2 Exercise 8 Answers

1. How many bytes does the NVRAM read function get when you press the button the first time?

   Zero because there is no data in the NVRAM yet.

2. What is the return status value when you press the button the first time?

   The return value is 40 (0x28).

3. What does the return value mean?

   The return value of 40 (0x28) means ERROR. This occurs because there is no data in the NVRAM to read yet. This is defined in the file *wiced_result.h*.

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Published by

Infineon Technologies AG

81726 Munich, Germany

IMPORTANT NOTICE

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics (“Beschaffenheitsgarantie”) .

With respect to any examples, hints or any typical values stated herein and/or any information regarding the application of the product, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party.

In addition, any information given in this document is subject to customer’s compliance with its obligations stated in this document and any applicable legal requirements, norms and standards concerning customer’s products and any use of the product of Infineon Technologies in customer’s applications.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer’s technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

For further information on the product, technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies office (**www.infineon.com**).

WARNINGS

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies’ products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.