Chapter 3: Using the BT SDK to Connect Inputs and Outputs to MCU Peripherals

At the end of 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 INTRODUCTION TO BLUETOOTH SOC PERIPHERALS

The Bluetooth devices include a useful set of MCU peripherals, such as UART, I2C and SPI communications, plus PWM, ADC and GPIO, which we are going to use to blink LEDs, print messages, measure acceleration and light levels, and so on. The Device Configurator is a graphical tool that helps you extend the BSP for your hardware, for example by choosing to drive an LED from a PWM instead of firmware.

Be aware, though, that the WICED example apps and snips predate the availability of the configurator and often rely on firmware-only configuration. As a result, you cannot assume that a peripheral or GPIO that is disabled in the configurator is not actually in use by the application. As ModusToolbox and its associated examples mature you will see a greater utilization of the configurator-generated code and more powerful set of configuration options (e.g. PWM duty cycle and UART baud rate).

3.2 BOARD SUPPORT PACKAGE

Every ModusToolbox application makes use of a Board Support Package (BSP), which is chosen in the first step of the New Application wizard (every kit has exactly one BSP). The BSP makes it easier to work with the peripherals on the kit. A BSP is a collection of files that specify which device is on the board, how it is programmed, what peripherals are available, which pins they are mapped to, etc.

In case you design your own hardware, you can either create a new BSP (beyond the scope of this course) or create an application with the BSP that most closely matches your hardware then modify the files as necessary. For example, you may have buttons and LEDs connected to different pins, so you would update the appropriate file to make those changes for your hardware.

Most of the configurable content of the BSP is managed by the design.modus file, which is the database for the Device Configurator. It generates the GeneratedSource/cycfg_pins.c file, which contains constant arrays that are used to configure the peripherals and to initialize the pins to the correct state. For example, LED pins are initialized as outputs, and the button pins are initialized as inputs with a resistive pullup. For example, for LED1:

```
#define LED1_config \
{\
    .gpio =
(wiced_bt_gpio_numbers_t*)&platform_gpio_pins[PLATFORM_GPIO_3].gp
io_pin, \
    .config = GPIO_OUTPUT_ENABLE | GPIO_PULL_UP, \
    .default_state = GPIO_PIN_OUTPUT_HIGH, \
}
```

The other key file here is wiced_platform.h, which contains #define and type definitions used to set up and access the various kit peripherals. For example, the CYW920819EVB-02 kit contains two LEDs and one mechanical button. These are identified in wiced_platform.h using the names WICED_GPIO_PIN_BUTTON_1, WICED_GPIO_PIN_LED_1 and WICED_GPIO_PIN_LED_2:

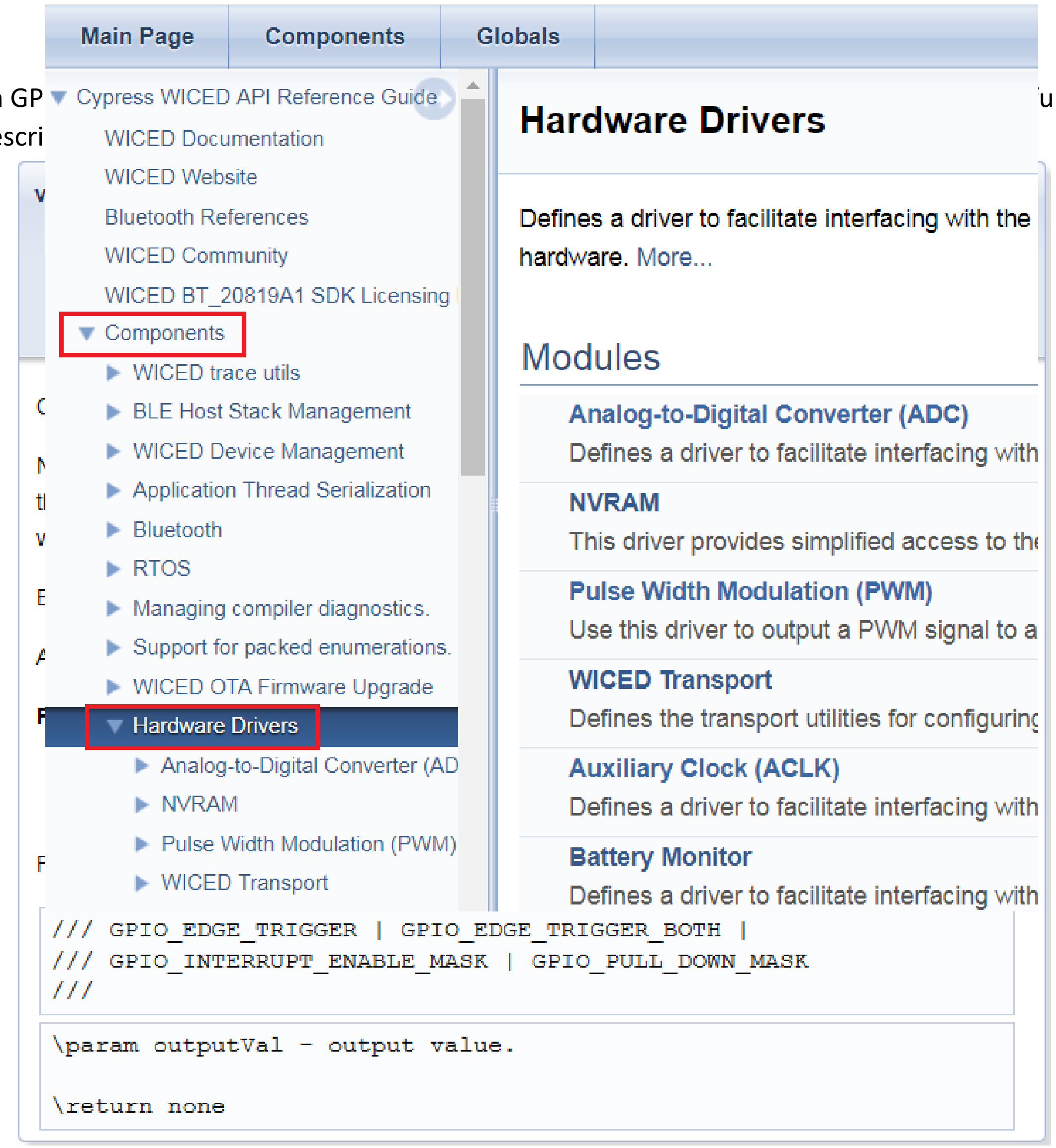
```
/*! pin for button 1 */
#define WICED_GPIO_PIN_BUTTON_1        WICED_P00
#define WICED_GPIO_PIN_BUTTON          WICED_GPIO_PIN_BUTTON_1

/*! pin for LED 1 */
#define WICED_GPIO_PIN_LED_1     WICED_P27
/*! pin for LED 2 */
```

```
    #define WICED_GPIO_PIN_LED_2      WICED_P26
```

3.3 DOCUMENTATION

CPU peripheral documentation can be found in the Documents tab in the lower left panel (next to the Quick Panel) or in the Help menu under *ModusToolbox API Reference->WICED API Reference*. The peripheral APIs are presented under Components->Hardware Drivers as shown below. We will be using GPIO, Pulse Width Modulation (PWM), Peripheral UART (PUART), I2C, and Real-Time Clock (RTC).

Click on GP unction for a descri



Main Page
Components
Globals
Cypress WICED API Reference Guide
WICED Documentation
WICED Website
Bluetooth References
WICED Community
WICED BT_20819A1 SDK Licensing
Components
WICED trace utils
BLE Host Stack Management
WICED Device Management
Application Thread Serialization
Bluetooth
RTOS
Managing compiler diagnostics.
Support for packed enumerations.
WICED OTA Firmware Upgrade
Hardware Drivers
Analog-to-Digital Converter (AD
NVRAM
Pulse Width Modulation (PWM)
WICED Transport

Hardware Drivers

Defines a driver to facilitate interfacing with the hardware. More...

Modules

Analog-to-Digital Converter (ADC)
Defines a driver to facilitate interfacing with

NVRAM
This driver provides simplified access to the

Pulse Width Modulation (PWM)
Use this driver to output a PWM signal to a

WICED Transport
Defines the transport utilities for configuring

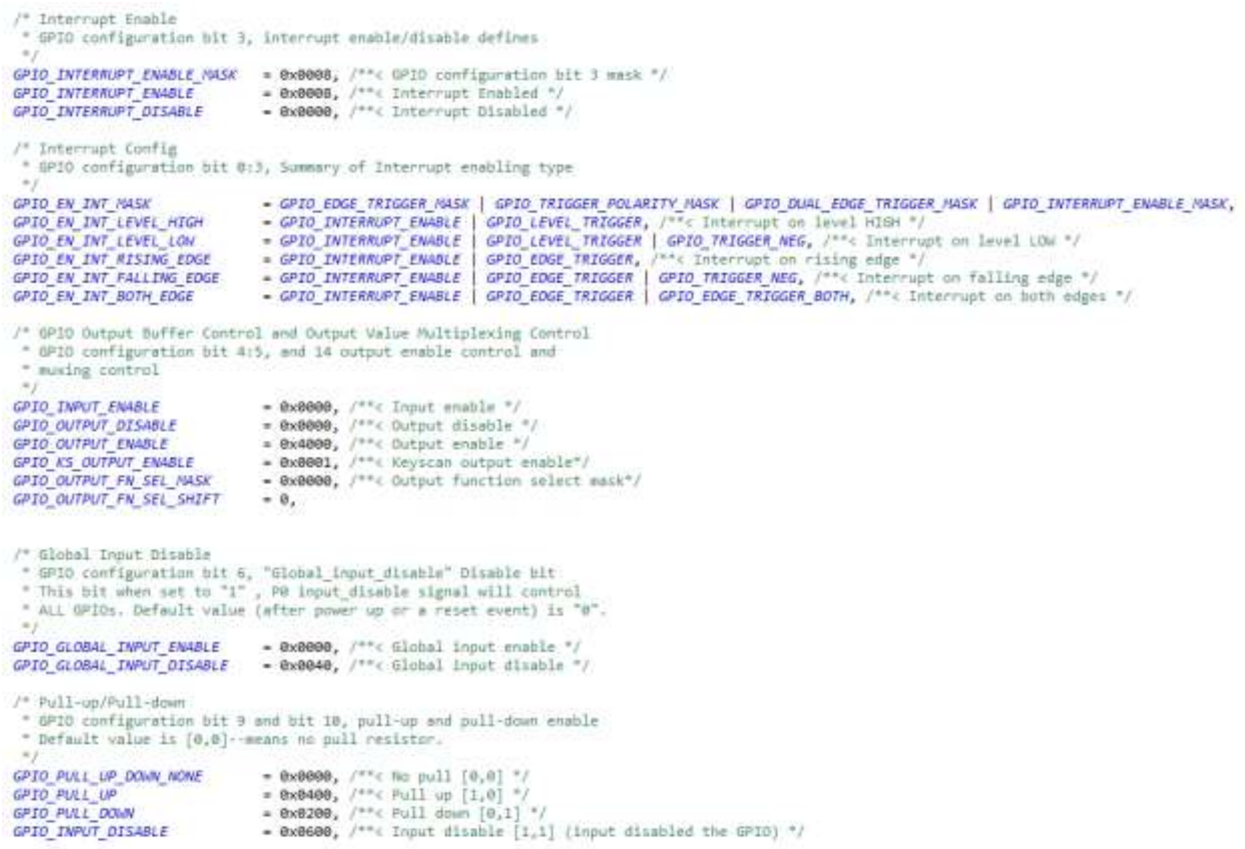
Auxiliary Clock (ACLK)
Defines a driver to facilitate interfacing with

Battery Monitor
Defines a driver to facilitate interfacing with

```
/// GPIO_EDGE_TRIGGER | GPIO_EDGE_TRIGGER_BOTH |
/// GPIO_INTERRUPT_ENABLE_MASK | GPIO_PULL_DOWN_MASK
///
```

```
\param outputVal - output value.

\return none
```

The description tells you what the function does but does not give complete information on the configuration value that is required. To find that information, once you create a project in ModusToolbox IDE, you can highlight the function in the C code, right click, and select "Open Declaration" (F3). 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 here:



```
/* 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:

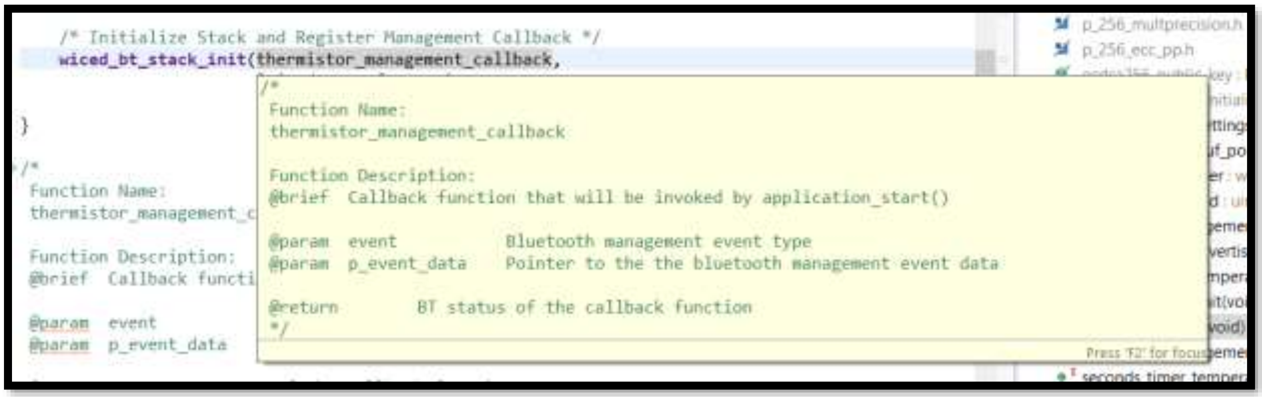
`GPIO_INPUT_ENABLE | GPIO_PULL_UP`

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

`GPIO_OUTPUT_ENABLE | GPIO_PULL_UP`

3.3-1 CONTEXT SENSITIVE HELP

Right-clicking and selecting "Open Declaration" on function names and data-types inside ModusToolbox IDE is often very useful in finding information on how to use functions and what values are allowed for parameters.
You can also just hover over a function name and get information like this:

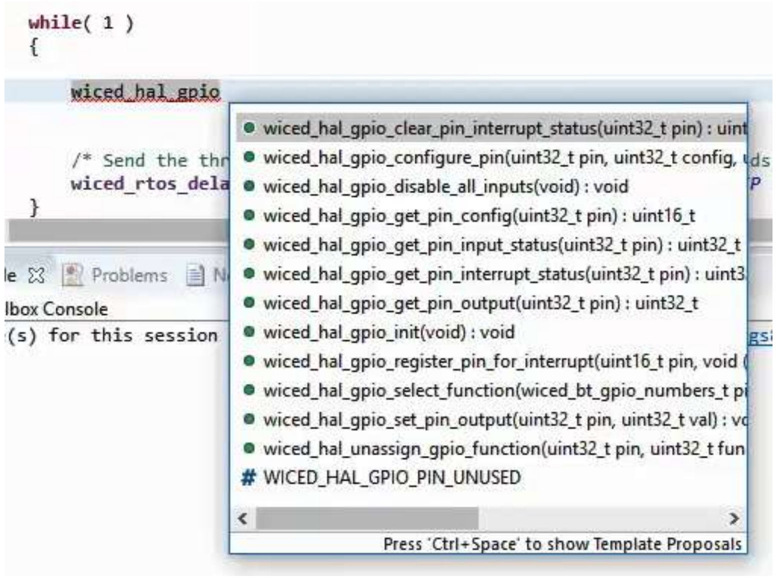


```
    /* Initialize Stack and Register Management Callback */
    wiced_bt_stack_init(thermistor_management_callback,

}

/*
 Function Name:
 thermistor_management_c

 Function Description:
 @brief  Callback functi

 @param  event
 @param  p_event_data
```

```
/*
 Function Name:
 thermistor_management_callback

 Function Description:
 @brief  Callback function that will be invoked by application_start()

 @param  event          Bluetooth management event type
 @param  p_event_data   Pointer to the the bluetooth management event data

 @return        BT status of the callback function
 */
```

3.3-2 INTELLISENSE

Another very useful tool is to type Control-Space when you are in the code editor. This will fill in possible completions for whatever you have already typed so that you can select what you want 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:



```
while( 1 )
{

    wiced_hal_gpio

    /* Send the thr
    wiced_rtos_dela
}
```

```
● wiced_hal_gpio_clear_pin_interrupt_status(uint32_t pin) : uint
● wiced_hal_gpio_configure_pin(uint32_t pin, uint32_t config,
● wiced_hal_gpio_disable_all_inputs(void) : void
● wiced_hal_gpio_get_pin_config(uint32_t pin) : uint16_t
● wiced_hal_gpio_get_pin_input_status(uint32_t pin) : uint32_t
● wiced_hal_gpio_get_pin_interrupt_status(uint32_t pin) : uint3
● wiced_hal_gpio_get_pin_output(uint32_t pin) : uint32_t
● wiced_hal_gpio_init(void) : void
● wiced_hal_gpio_register_pin_for_interrupt(uint16_t pin, void
● wiced_hal_gpio_select_function(wiced_bt_gpio_numbers_t pi
● wiced_hal_gpio_set_pin_output(uint32_t pin, uint32_t val) : vo
● wiced_hal_unassign_gpio_function(uint32_t pin, uint32_t fun
# WICED_HAL_GPIO_PIN_UNUSED
```

3.4 PERIPHERALS

3.4-1 GPIO

As explained previously, GPIOs must be configured using the function *wiced_hal_gpio_configure_pin()*.
The IOs on the kit that are connected to specific peripherals such as LEDs and buttons are usually

configured for you as part of the platform files, so you don't need to configure them explicitly in your projects unless you want to change a setting (for example to enable an interrupt on a button pin). 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().* The parameter for these functions is the WICED pin name such as WICED_P01 or a peripheral name from your BSP such as WICED_GPIO_PIN_LED_1.

For example:

```
  btnState = wiced_hal_gpio_get_pin_input_status( WICED_GPIO_PIN_BUTTON_1 ); /* Get
pin state */
  wiced_hal_gpio_set_pin_output(WICED_GPIO_PIN_LED_2, 0); /* Set pin low */
  wiced_hal_gpio_set_pin_output(WICED_GPIO_PIN_LED_2,
       !wiced_hal_gpio_get_pin_output(WICED_GPIO_PIN_LED_2) );  /* Invert desired pin
state */
```

You will use code like this in exercise 1.

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( WICED_GPIO_PIN_BUTTON_1,
       my_interrupt_callback, NULL);

  wiced_hal_gpio_configure_pin( WICED_GPIO_PIN_BUTTON_1,
       ( GPIO_INPUT_ENABLE | GPIO_PULL_UP | GPIO_EN_INT_FALLING_EDGE),
       GPIO_PIN_OUTPUT_HIGH );
```

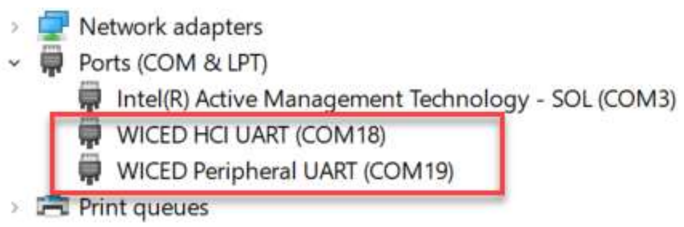
The interrupt callback function is passed user data (optional) and the pin number. The callback function should clear the interrupt using *wiced_hal_gpio_clear_pin_interrupt_status*. For example:

```
  void gpio_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.

3.4-2 DEBUG PRINTING

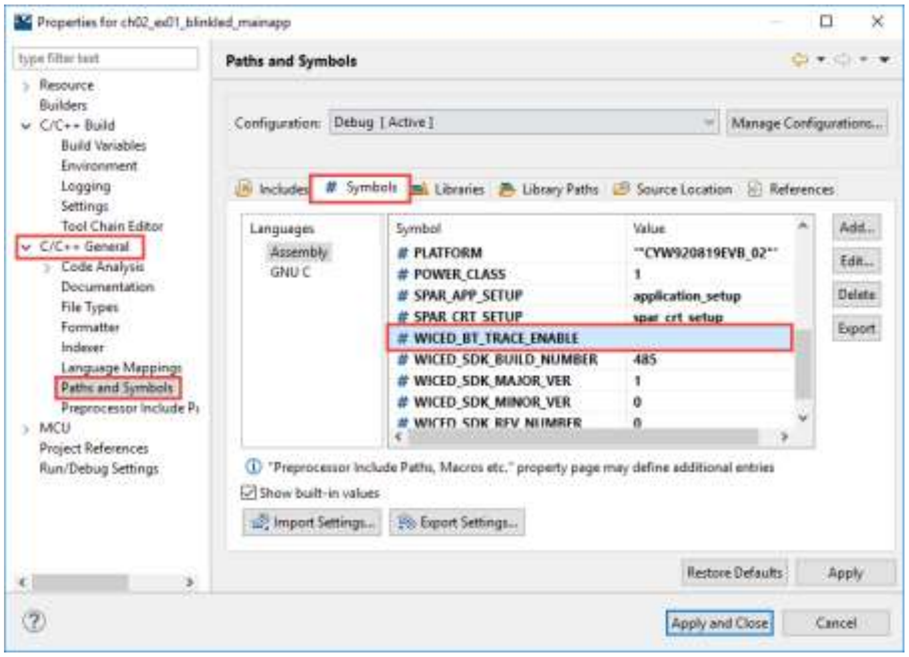
The kit has two separate UART interfaces –the HCI UART (Host Controller Interface UART) and the PUART (peripheral UART). The HCI UART interface is used for programming the kit and often is used by a host microcontroller to communicate with the BLE device. It will be discussed in more detail in a later chapter. 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.


Network adapters
Ports (COM & LPT)
Intel(R) Active Management Technology - SOL (COM3)
WICED HCI UART (COM18)
WICED Peripheral UART (COM19)
Print queues

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 Properties in the *Symbols* tab under *C/C++ General->Paths and Symbols*.

   Note: to get to the window, first select the project that you want to modify in the project explorer, then either click "Project Build Settings" in the Quick Panel or use the menu item *Project->Properties*.

   Configuration: Debug [Active]

Symbol	Value
# PLATFORM	""CYW920819EVB_02""
# POWER_CLASS	1
# SPAR_APP_SETUP	application_setup
# SPAR_CRT_SETUP	spar_crt_setup
# WICED_BT_TRACE_ENABLE	
# WICED_SDK_BUILD_NUMBER	485
# WICED_SDK_MAJOR_VER	1
# WICED_SDK_MINOR_VER	0
# WICED_SDK_REV_NUMBER	0

   "Preprocessor Include Paths, Macros etc." property page may define additional entries

   ☑ Show built-in values

   The provided starter templates all set this up automatically so editing the properties is not usually necessary.

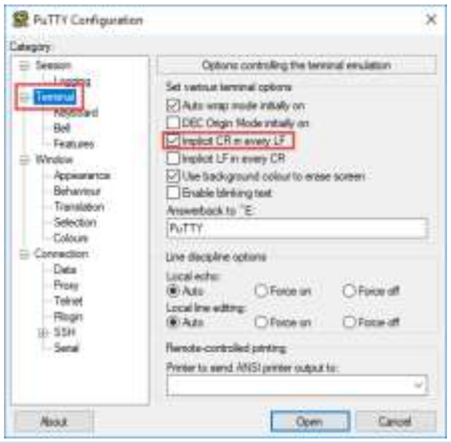
2. Include the following header in the top-level C file:
```
#include "wiced_bt_trace.h"
```
3. Indicate which interface you want to use by choosing one of the following:
```
wiced_set_debug_uart( WICED_ROUTE_DEBUG_NONE);
wiced_set_debug_uart( WICED_ROUTE_DEBUG_TO_PUART );
wiced_set_debug_uart( WICED_ROUTE_DEBUG_TO_HCI_UART );
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 sprintf-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.
Note that we typically put "\n" at the end of strings to be printed but not "\r" to save flash/ram.
Typically, you will want to setup 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 with if you want it to be on by default (you can also save the default speed to be 115200).



Options controlling the terminal emulation

Set various terminal options
[✓] Auto wrap mode initially on
[ ] DEC Origin Mode initially on
[✓] Implicit CR in every LF
[ ] Implicit LF in every CR
[✓] Use background colour to erase screen
[ ] Enable blinking text
Answerback to ^E: PuTTY

Line discipline options
Local echo: [✓ Auto] Force on Force off
Local line editing: [✓ Auto] Force on Force off

Remote-controlled printing
Printer to send ANSI printer output to:

3.4-3 PUART (PERIPHERAL UART)

In addition to the debug printing functions, the PUART can also 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 setup the flow control and baud rate. For example:

```
        wiced_hal_puart_init( );
        wiced_hal_puart_flow_off( );
        wiced_hal_puart_set_baudrate( 115200 );
```

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 */
    }
```

3.4-4 TIMERS

A timer allows you to schedule a function to run at a specified interval - e.g. send data every 10 seconds. First, you initialize the timer using wiced_init_timer, you give it a pointer to a timer structure, specify the function want run, provide 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 repeat:

```
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 of u*int32_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 u*int32_t arg* argument in its definition.

Once 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 that 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.

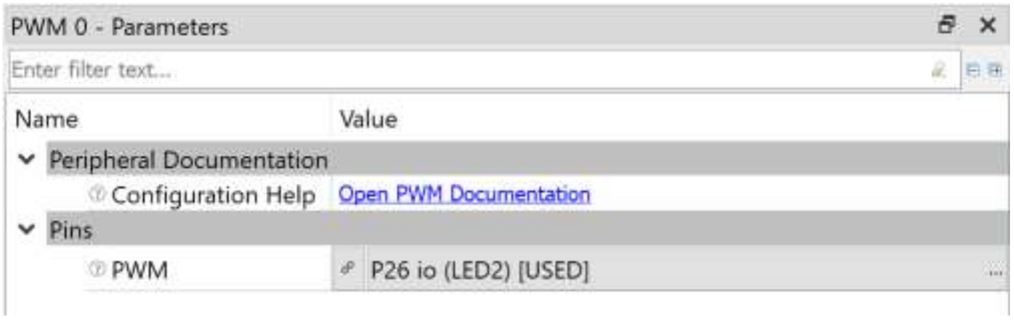
For example, to setup a timer that runs a function called myTimer every 100ms, you would do something like this:

```
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 */
}
```

3.4-5 PWM

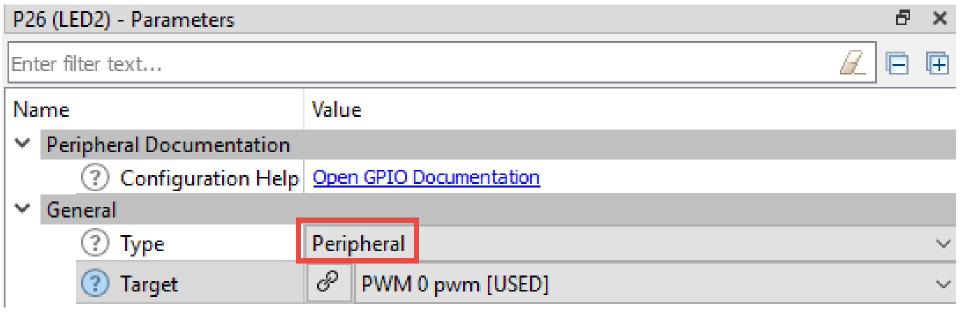
There are 6 PWM blocks (PWM0 – PWM5) on the device each of which can be routed to any GPIO pin. The PWMs are 16 bits (i.e. they count from 0 to 0xFFFF).

The PWMs can use either the LHL_CLK (which is 32 kHz) or PMU_CLK (aka ACLK1) which is configurable. You use the Device Configurator to enable the PWM and assign its output to one of the LED pins.


PWM 0 - Parameters

Name	Value
Peripheral Documentation	
Configuration Help	Open PWM Documentation
Pins	
PWM	P26 io (LED2) [USED]

You also need to change the pin configuration from LED to Peripheral:


P26 (LED2) - Parameters

Name	Value
Peripheral Documentation	
Configuration Help	Open GPIO Documentation
General	
Type	Peripheral
Target	PWM 0 pwm [USED]

You can jump back and forth from the PWM to its associated pin using the button that looks like 2 links in a chain.

Note: The configurator currently only does pin routing – in the future it will provide more functionality like selecting clock sources, setting period and compare, etc. If you don't want to use the configurator for pin routing, you can instead use a line of code during initialization like this:

```
    wiced_hal_gpio_select_function(WICED_GPIO_PIN_LED_2, WICED_PWM0);
```

The pin name used above is defined in wiced_platform.h and the peripheral name is defined in wiced_hal_gpio.h.

The solution projects and templates generally use code instead of the configurators because it makes the new application creation simpler.

Whether you use the configurator or manual pin routing from the code, you must include the PWM header file to use the PWM API functions:

```
    #include "wiced_hal_pwm.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 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.

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:

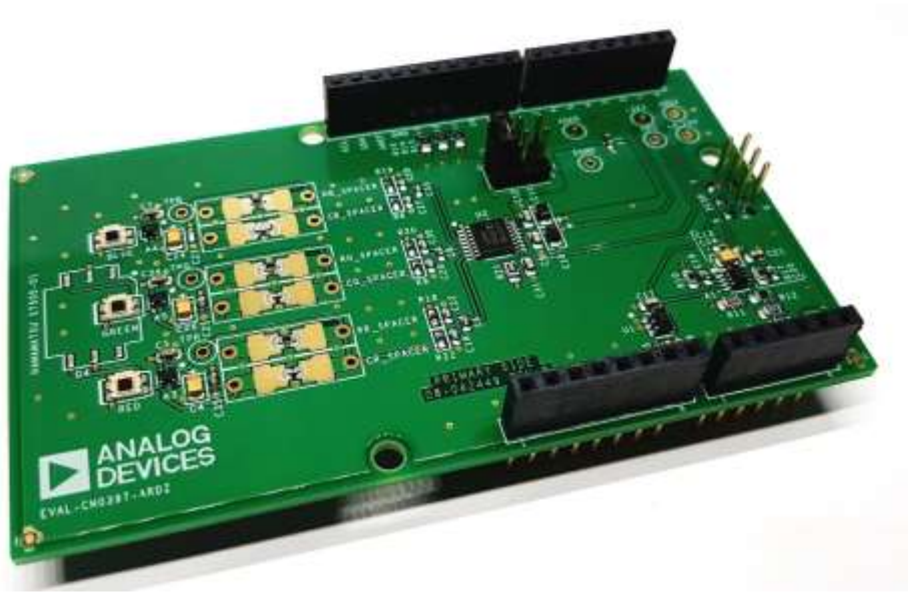
```
#define CLK_FREQ    (1000)

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

Note that there is only 1 PMU clock available so 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, change values while the PWM is running, get the init value, and get the toggle count. There is even a helper function called *wiced_hal_pwm_params()* which will calculate the parameters you need given the clock frequency, the desired output frequency, and desired duty cycle. See the documentation for details on each of these functions.

3.4-6 SPI (ADI CN0397 RGB LIGHT SENSOR)

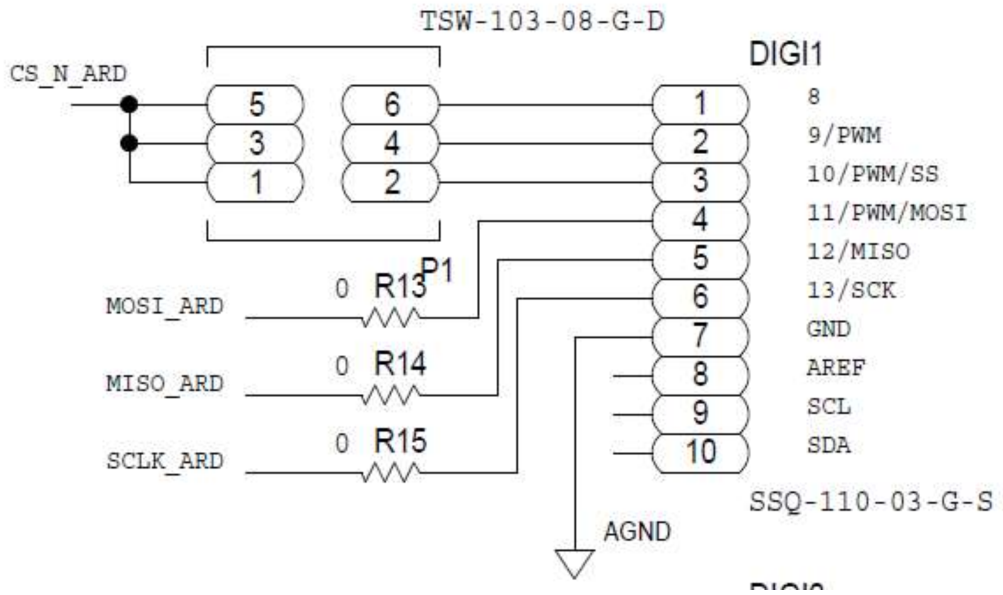
This section shows how to interface both hardware and software to the CN0397 RGB Light Sensor Arduino Shield.
CN0397 is an Arduino compatible shield that is optimized for smart agriculture utilizing wavelength specific photodiodes. Photosynthetic response of plants varies due to the wavelength and intensity of light received. Photodiodes used in this circuit has peak sensitivities over the wavelengths of interest, red and blue region, and over the green region which is mainly rejected by the leaves of the plant.



CN0397 Users Guide

The circuit on the shield uses **AD8500,** a low power, precision CMOS op amp with a low input bias current of a typical 1pA which is used in a transimpedance amplifier configuration to convert the current output of the photodiodes into voltage. It also features **AD7798** a 3-channel, low noise, low power 16-bit Delta Sigma ADC that converts the analog voltage into digital data in for the processing of data into light intensity. Gain resistor for each of the channel had been calculated to maximize the full scale of the ADC and additional filtering to remove unwanted signals.
We will be interfacing to the AD7798 3 channel ADC as a SPI Slave.
For the SPI hardware, we need to define 4 pins
CS – Slave Select or Chip Select
SCLK – SPI Clock
MOSI – Master data Out / Slave In
MISO – Master data In / Slave Out

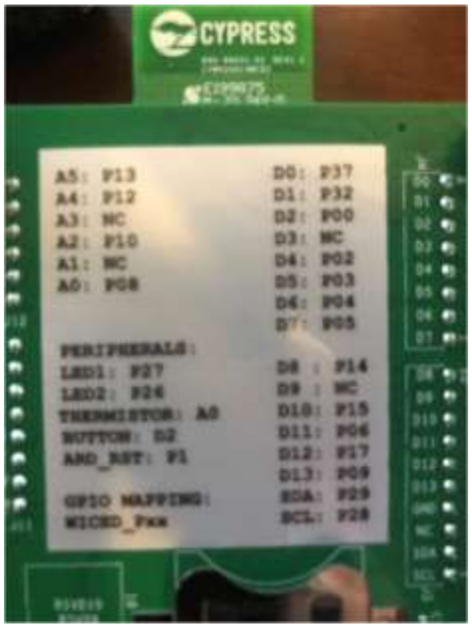
On the CN0397, they are defined and found on the schematic



Since this is the only shield, we will use SS on DIG1 -10

CS	DIG1 – 10 (make sure that the jumper on P1 on the CN0397 is across pins 1-2)
MOSI	DIG1 – 11
MISO	DIG1 – 12
SCLK	DIG1 – 13

On the bottom of the CYW920819EVB-02, there is a sticker with the Platform Names



CS	DIG1 -10	=	P15
MOSI	DIG1 – 11	=	P06
MISO	DIG1 – 12	=	P17
SCLK	DIG1 – 13	=	P09

So that is the hardware connection. Now for the SW. Create a new application using the CN0397 Template project. The file with application_start() is spi_w_sensor.c in the template application . This will need to be modified to add the SPI port

You must include the header file to use the SPI functions: #include "wiced_hal_pspi.h".   There is a demo project in the SDK 1.4 examples called datalogger which has a dual SPI master with the RTOS threads enables. We used this as a base project to create the SPI drivers.   From the HAL_Dual_SPI_Master_mainapp, we find these setting for the SPI which we will need for the CN0397 drivers.

```
/*SPI 1 defines for HAL_Dual_SPI_Master*/
#define CLK_1                                   WICED_P15
#define MISO_1                                  WICED_P14
#define MOSI_1                                  WICED_P13
#define CS_1                                    WICED_P12
```

We need to change the Pin definitions to match the Arduino shield

```
/*SPI 1 defines for Master to talk to CN0397 slave */
#define CLK_1                                   WICED_P09
#define MISO_1                                  WICED_P17
#define MOSI_1                                  WICED_P06
#define CS_1                                    WICED_P15
/* 1 MHz frequency*/
#define DEFAULT_FREQUENCY                       (1000000u)
/* SPI register configuration macro*/
#define GPIO_CFG(CS_1,CLK_1,MOSI_1,MISO_1)
((((UINT32)CS_1&0xff)<<24)|((UINT32)CLK_1&0xff)<<16)|(((UINT32)MOSI_1&0xff)<<8)|((UIN
T32)MISO_1)
```

In the initialize_app routine, the SPI1 peripheral gets initialized. (Remember: to use SPIs calls in a function you need to `#include "wiced_hal_pspi.h"`.)

```
wiced_hal_pspi_init(SPI1,
                    SPI_MASTER,
                    INPUT_PIN_PULL_UP,
                    GPIO_CFG(CS_1,CLK_1,MOSI_1,MISO_1),
                    DEFAULT_FREQUENCY,
                    SPI_LSB_FIRST,
                    SPI_SS_ACTIVE_LOW,
                    SPI_MODE_0,
                    CS_1);
```

From the Arduino drivers supplied by ADI initialize the SPI port to Mode 3 with the MSB First. These need to be changed.

```
wiced_hal_pspi_init(SPI1,
                    SPI_MASTER,
                    INPUT_PIN_PULL_UP,
                    GPIO_CFG(CS_1,CLK_1,MOSI_1,MISO_1),
                    DEFAULT_FREQUENCY,
                    SPI_MSB_FIRST,
                    SPI_SS_ACTIVE_LOW,
                    SPI_MODE_3,
                    CS_1);
```

The CN0397 is initialized in the sensor thread and then sets the state to Read the CN0397 data and display it in the debug window and then delay for 500ms.   For the spi_master_w_sensor.c to know where the CN0397 calls are, you need to include CN0397.h.

```
void spi_sensor_thread(uint32_t arg )
{
    wiced_result_t result;
    master_state curr_state = SENSOR_INIT;

    while(WICED_TRUE)
    {
        switch(curr_state)
        {
        case SENSOR_INIT:

            /* Init CN0397 and read ID */
             CN0397_Init();

             curr_state = READ_AD7798;

            break;

        case READ_AD7798:

             CN0397_SetAppData();
             CN0397_DisplayData();
             wiced_rtos_delay_milliseconds(500, ALLOW_THREAD_TO_SLEEP);

            break;

        default:

             break;
        }
    }
 }
```

The initializing, calibration and reading of the data is done in the drivers which were developed by ADI for this shield.   You will see those routines in CN0397.c and AD7798.c
We had to modify the spi_sensor_read() and spi_sensor_write() routines for the CYW920818EVB-02. The files are SPI_Comm.c and SPI_Comm.h.   First you need to activate the CS signal by setting it LOW. (Remember: to use GPIO in a function you need to `#include "wiced_hal_gpio.h"`). Then you perform the write(s) or write and read(s). Finally, you de-activate the CS signal by setting it HIGH. You can find the `wiced_hal_pspi_tx_data()` and `wiced_hal_pspi_rx_data` in the SDK 1.4 API documentation under Components → Hardware Drivers → PeripheralSPIDriver
Remember: to use SPIs calls in a function you need to `#include "wiced_hal_pspi.h"`.
Here is the spi_sensor_write()

```
void  spi_sensor_write(uint8_t byteCount, uint8_t *send_msg)
{
    /* Chip select is set to LOW to select the slave for SPI transactions*/
    wiced_hal_gpio_set_pin_output(CS_1, GPIO_PIN_OUTPUT_LOW);
    /* Writ data to slave */
    wiced_hal_pspi_tx_data(SPI1,
                           byteCount,
                           (uint8_t*)send_msg);

    /* Chip select is set to HIGH when SPI transaction is complete*/
    wiced_hal_gpio_set_pin_output(CS_1, GPIO_PIN_OUTPUT_HIGH);
    return;
}
```

3.4-7 NVRAM

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 BT_20819 SDK provides an abstraction called the "NVRAM" (it is really just an area of flash memory that is set aside) for this purpose. The BSP 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 20819 device), of which 500 are available for user data. To use the NVRAM, the application developer writes/reads to/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. By using the VSID the user does not need to concern themselves with the (moving) physical location of their data. The only cost of this implementation is that reads and writes to NVRAM take a variable amount of time. Note that the 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 project 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 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 that 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.

3.4-8 I2C

There is an I2C master on the device which is routed by default to the Arduino header dedicated I2C pins. It is also connected to an LSM9DS1 motion sensor on the CYW920819EVB-02 kit.

3.4-8A INITIALIZATION

You must include the I2C header file to use the I2C functions:

```
	#include "wiced_hal_i2c.h"
```

To initialize the I2C block you need to call the initialization function. If you want a speed other than the default of 100 kHz then you have to call the set_speed function after the block is initialized:

```
	wiced_hal_i2c_init();
	wiced_hal_i2c_set_speed(I2CM_SPEED_400KHZ);
```

3.4-8B READ AND WRITE FUNCTIONS

There are two ways to read/write data from/to the slave. There is a dedicated read function called *wiced_hal_i2c_read()* and a dedicated write function called *wiced_hal_i2c_write()*. There is also a function called wiced_hal_i2c_combined_read() which will do a write followed by a read with a repeated start between them. These functions are all blocking.

The separate read/write functions require a pointer to the buffer to read/write, the number of bytes to read/write, and the 7-bit slave address. The LSM9DS1 is at address 0xD4 (write) / 0xD5 (read) and so, to generate the 7-bit address, shift 0xD4 right by 1 bit (0x6A).

For example, to write 2 bytes followed by a read of 10 bytes:

```
	#define I2C_ADDRESS (0x6A)
	uint8_t TxData[2] = {0x55, 0xAA};
	uint8_t RxData[10];
	wiced_hal_i2c_write( TxData, sizeof(TxData), I2C_ADDRESS );
	wiced_hal_i2c_read( RxData, sizeof(RxData), I2C_ADDRESS );
```

If you need to write a value (e.g. a register offset value) followed by a read, you can use the *wiced_hal_i2c_combined_read()* function to do both in one function call. The function takes a pointer to the write data buffer, the number of bytes to write, a pointer to the read data buffer, the number of bytes to read, and finally, the 7-bit slave address.

For example, the same operation shown above could be:

```
	#define I2C_ADDRESS (0x6A)
	uint8_t TxData[2] = {0x55, 0xAA};
	uint8_t RxData[10];
	wiced_hal_i2c_combined_read( TxData, sizeof(TxData), RxData,
	sizeof(RxData), I2C_ADDRESS );
```

3.4-8C READ/WRITE BUFFER

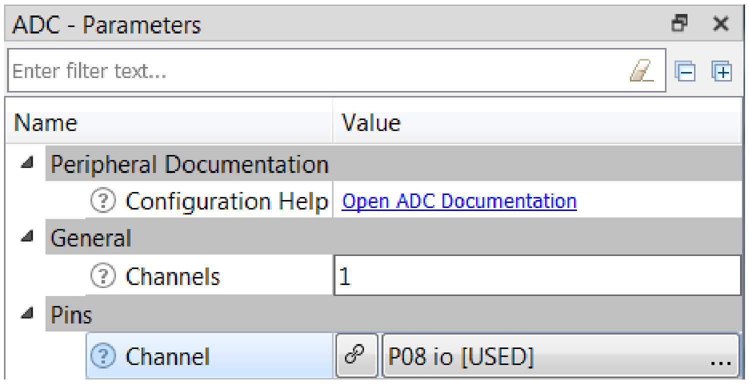
For the buffer containing the data that you want to read/write, you may want to setup a structure to map the I2C registers in the slave that you are addressing. In that case, if the structure elements are not all 32-bit quantities, you must use the packed attribute so that the non-32-bit quantities are not padded, which would lead to incorrect data. For example, if you have a structure with 3-axis 16-bit acceleration values, you could set up a buffer like this:

```
	struct {
		int16_t ax;
		int16_t ay;
		int16_t az;
	} __attribute__((packed)) accel_data;
```

There are two underscores before and after the word "attribute" and there are two sets of parentheses around the word "packed".

3.4-9 ADC

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 that this is not routing signals on the device, merely assigning virtual channel numbers to the pins). Note that the voltage channels, like Vddio, are configured automatically for you. The ADC is enabled and set to P08 (thermistor) by default.



Name	Value
Peripheral Documentation	
Configuration Help	Open ADC Documentation
General	
Channels	1
Pins	
Channel	P08 io [USED]

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 these 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 );
```

3.4-10 RTC (REAL TIME CLOCK)

The CYW20819 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 CYW20819, the firmware takes the clock input from the internal LPO for the RTC. The CYW20819 supports both 32-kHz and 128-kHz LPOs, but the internal defaults to 32-kHz.

The BT_20819A1 SDK 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.

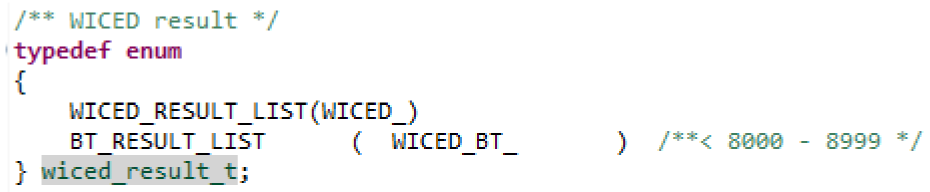
You must include "wiced_rtc.h" and the following code to initialize the RTC for use:

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

     rtc_setRTCTime( &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.

3.5 WICED_RESULT_T

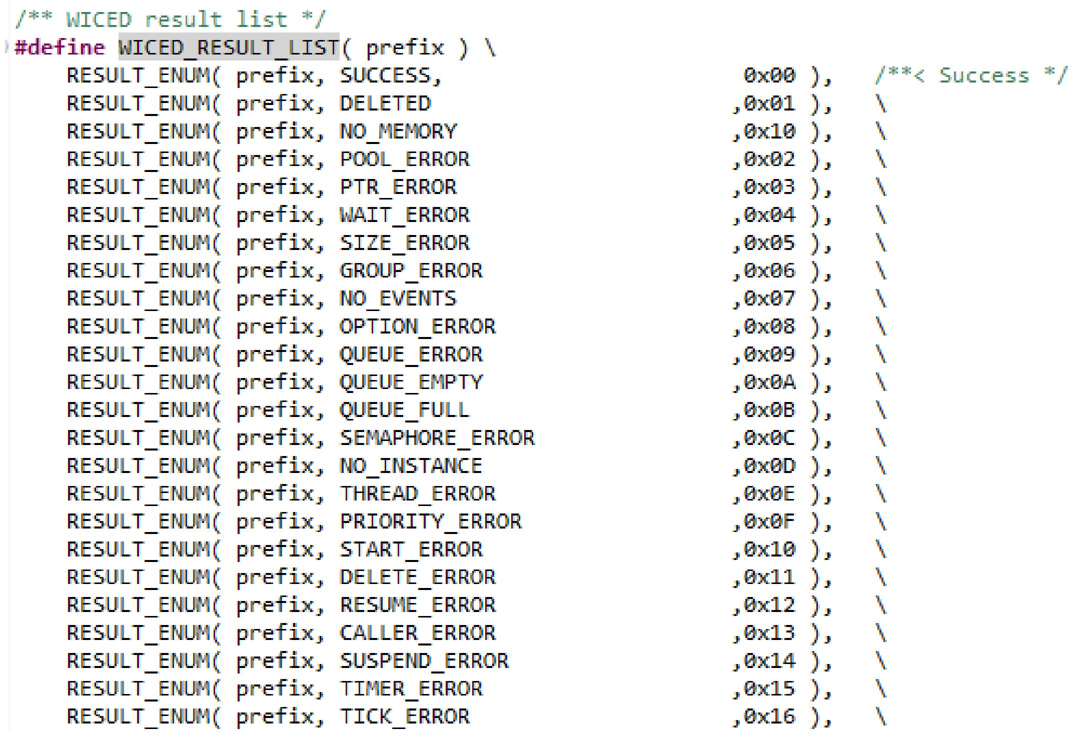
Throughout the SDK, a value from many of the functions is returned telling you what happened. The return value is of the type "wiced_result_t" which is a giant enumeration. If you right-click on wiced_result_t from a variable declaration, select "Open Declaration", and choose wiced_result.h you will see this:



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

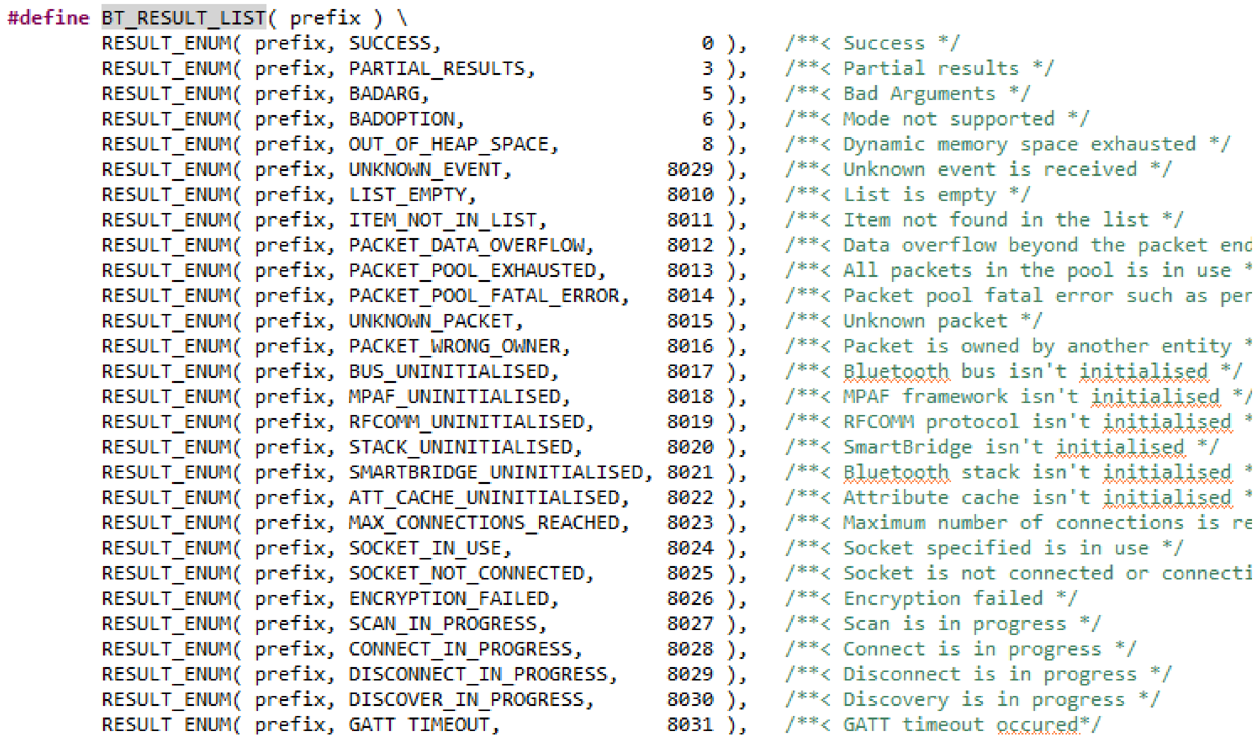
To see standard return codes (WICED_*), right click and choose Open Declaration on WICED_RESULT_LIST. For Bluetooth specific return codes (WICED_BT_*), right click and choose Open Declaration on BT_RESULT_LIST. The lists look like this:

WICED_* :



```
/** 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 ),   \
    RESULT_ENUM( prefix, DELETE_ERROR                 ,0x11 ),   \
    RESULT_ENUM( prefix, RESUME_ERROR                 ,0x12 ),   \
    RESULT_ENUM( prefix, CALLER_ERROR                 ,0x13 ),   \
    RESULT_ENUM( prefix, SUSPEND_ERROR                ,0x14 ),   \
    RESULT_ENUM( prefix, TIMER_ERROR                  ,0x15 ),   \
    RESULT_ENUM( prefix, TICK_ERROR                   ,0x16 ),   \
```

WICED_BT_*:



```
#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 */
        RESULT_ENUM( prefix, SCAN_IN_PROGRESS,        8027 ),   /**< Scan is in progress */
        RESULT_ENUM( prefix, CONNECT_IN_PROGRESS,     8028 ),   /**< Connect is in progress */
        RESULT_ENUM( prefix, DISCONNECT_IN_PROGRESS,  8029 ),   /**< Disconnect is in progress */
        RESULT_ENUM( prefix, DISCOVER_IN_PROGRESS,    8030 ),   /**< Discovery is in progress */
        RESULT_ENUM( prefix, GATT_TIMEOUT,            8031 ),   /**< GATT timeout occured*/
```