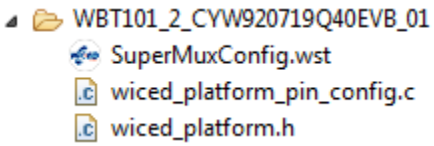
Chapter 2: Using the WICED SDK to Connect Inputs and Outputs to MCU Peripherals

Time 2 Hours

At the end of this chapter you should be able to write firmware for the MCU peripherals (GPIOs, PWMs, UART, I2C, and ADC) and to interface with the shield including the PSoC, LEDs, Buttons, Thermistor, Humidity Sensor, Ambient Light Sensor, Potentiometer, and OLED display. In addition, you will understand the role of the critical files related to the kit hardware platform.

2.1 The WICED Board Support Package (Platform)

The WICED SDK has files that make it easier to work with the peripherals on a given kit. In our case, we are using a baseboard kit along with an analog front-end (AFE) shield which contains a PSoC chip. To make it easier to interface with the shield, a set of platform files has been created. Since this is not installed by default in the SDK we need to copy the platform folder into the SDK Workspace. The folder for this kit/shield combination is named "WBT101_2_<KitName>" where <KitName> is the name of the baseboard kit being used and it is provided with the class materials in the "WW101 Files" folder. Copy the entire "WBT101_2_<KitName>" folder for the baseboard you are using from the class materials into the "platforms" directory in the SDK Workspace. For example, if you are using the CYW920719Q40EVB_01 kit, the contents of WBT101_2_CYW920719Q40EVB_01 is:



Two key files here are wiced_platform_pin_config.c, and wiced_platform.h. The wiced_platform.h file contains #define and type definitions used to set up and access the various kit and shield peripherals. For example, the shield contains two LEDs and two mechanical buttons. These are identified in wiced_platform.h using the names WICED_GPIO_PIN_LED_1, WICED_GPIO_PIN_LED_2, WICED_GPIO_PIN_BUTTON_1, and WICED_GPIO_PIN_BUTTON_2.

```
/*! pins for buttons and LEDs on the shield */
#define WICED_GPIO_PIN_LED_1     WICED_P26
#define WICED_GPIO_PIN_LED_2     WICED_P28
#define WICED_GPIO_PIN_BUTTON_1  WICED_P00
#define WICED_GPIO_PIN_BUTTON_2  WICED_P01
```

The names used are re-mapped from the base board so that instead of the LEDs and buttons on the base board, we will use the corresponding resources from the shield. Note that the baseboard only contains one button while the shield contains two so there is no corresponding button 2 on the baseboard. The names for the two LEDs and the first button will stay the same – only the platform that we target will be different. This means if you have a project that uses LEDs and button 1, you can use the same project C file and make file to run it using either just the baseboard or the baseboard plus shield by just changing the platform name in the Make Target. The LEDs on the baseboard and the shield are on the same pins (although LED1 and LED2 are swapped) but they are of opposite polarity. If you don't want the LEDs on the base board blinking opposite to the ones on the shield, just use the DIP switches on the base board to turn them off.

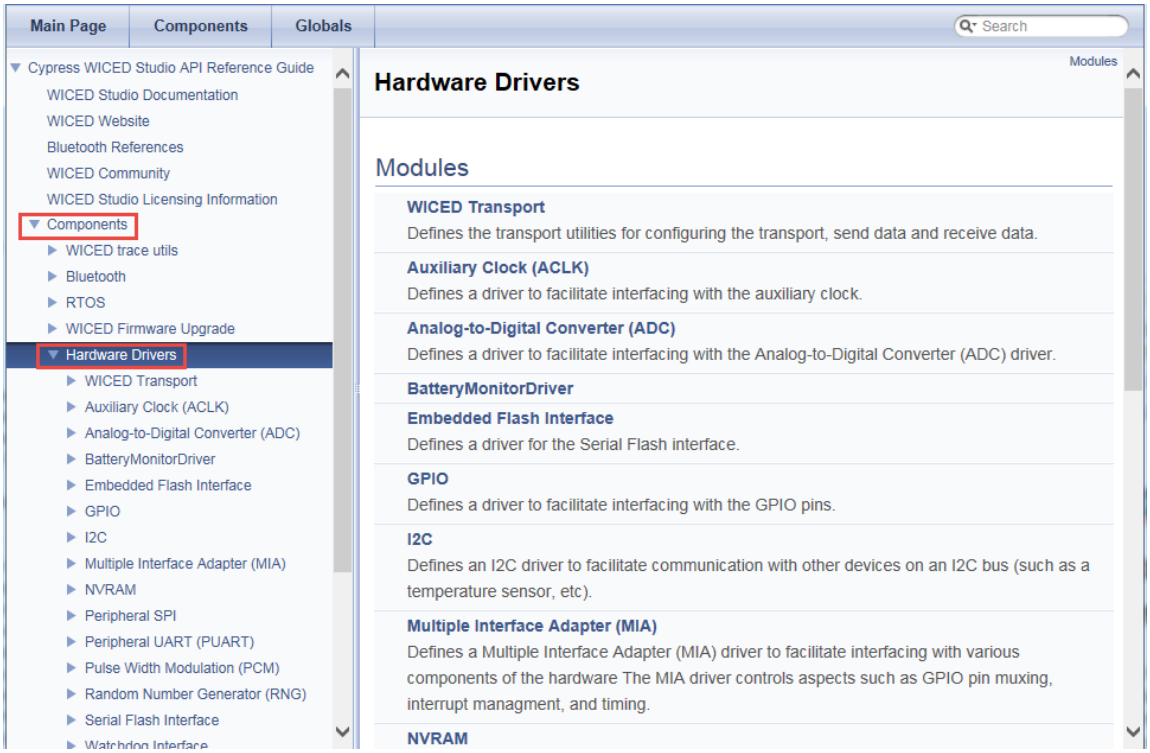
The wiced_platform_pin_config.c file contains constant arrays that are used to configure the peripherals and pins and to initialize them to the correct state. For example, the LED pins are initialized as outputs and the button pins are initialized as inputs with a resistive pullup.

The third file in the platform folder called SuperMuxConfig.wst is a configuration file used by the SuperMux pin configuration tool. That tool will be discussed later.

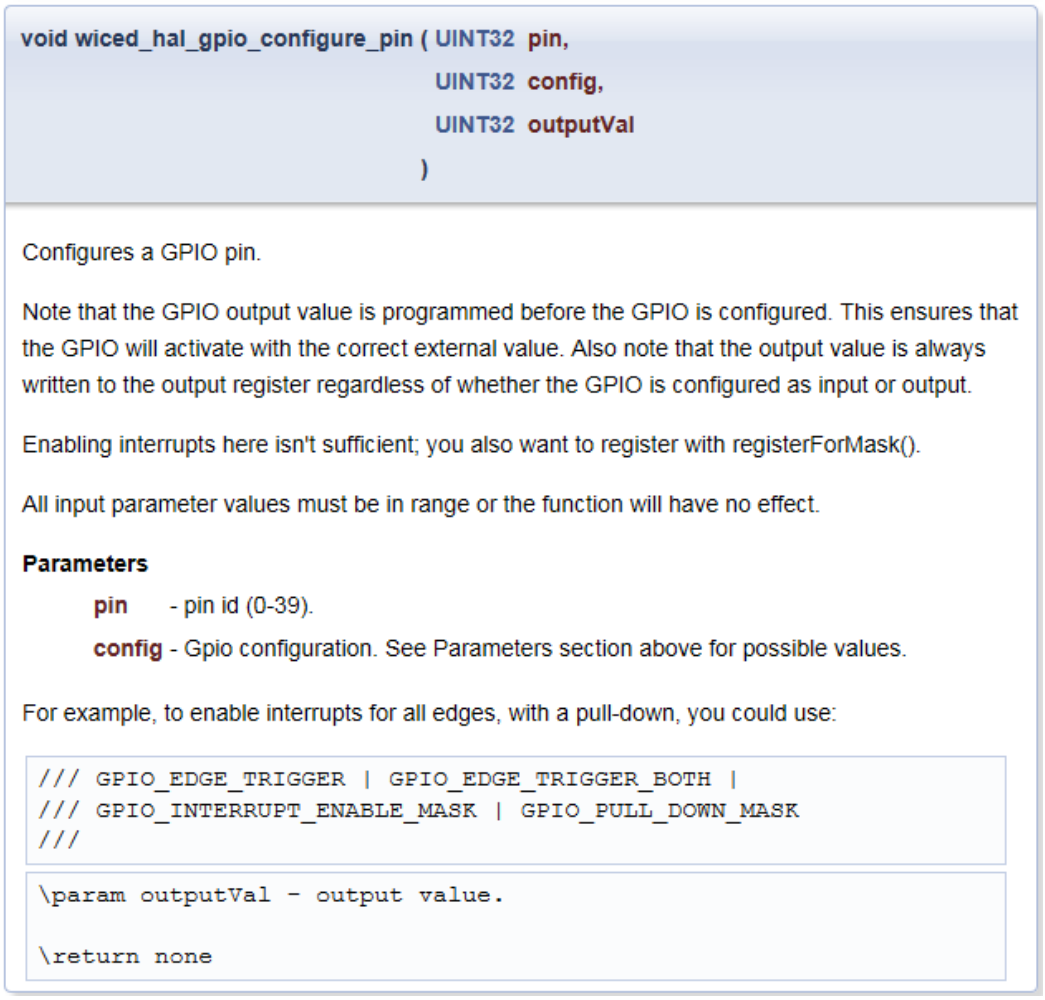
If you develop your own hardware, it is best to add a new folder to the SDK Workspace platform folder with the appropriate files for your hardware. It is usually easiest to copy an existing platform and modify it as necessary for any different hardware connections.

2.2 Documentation

CPU peripheral documentation can be found in the SDK Workspace doc folder. The file API.html contains the documentation of the APIs that we will be using. Open this file by right-clicking on it and selecting *Open With > System Editor* and then expand "Components" and "Hardware Drivers" to see the list of supported components. We will be using GPIO, Pulse Width Modulation (PWM), Peripheral UART (PUART), I2C, and Analog-to-Digital Converter (ADC).



Click on GPIO to see the list of GPIO APIs and then click on the *wiced_hal_gpio_configure_pin* function for a description.



void wiced_hal_gpio_configure_pin (UINT32 pin,
UINT32 config,
UINT32 outputVal
)

Configures a GPIO pin.

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 want to register with registerForMask().

All input parameter values must be in range or the function will have no effect.

Parameters

pin - pin id (0-39).

config - Gpio configuration. See Parameters section above for possible values.

For example, to enable interrupts for all edges, with a pull-down, you could use:

```
/// 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 information on the configuration value that is required. To find that information, once you are in WICED Studio you can highlight the function in the C code, right click, and select "Open Declaration". 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:

```
84      /// Interrupt Enable
85      /// GPIO configuration bit 3, interrupt enable/disable defines
86      GPIO_INTERRUPT_ENABLE_MASK    = 0x0008,
87      GPIO_INTERRUPT_ENABLE         = 0x0008,
88      GPIO_INTERRUPT_DISABLE        = 0x0000,
89
90
91      /// Interrupt Config
92      /// GPIO configuration bit 0:3, Summary of Interrupt enabling type
93      GPIO_EN_INT_MASK              = GPIO_EDGE_TRIGGER_MASK | GPIO_TRIGGER_POLARITY_MASK | GPIO_DUAL_EDGE_TRIGGER_MA
94      GPIO_EN_INT_LEVEL_HIGH        = GPIO_INTERRUPT_ENABLE | GPIO_LEVEL_TRIGGER,
95      GPIO_EN_INT_LEVEL_LOW         = GPIO_INTERRUPT_ENABLE | GPIO_LEVEL_TRIGGER | GPIO_TRIGGER_NEG,
96      GPIO_EN_INT_RISING_EDGE       = GPIO_INTERRUPT_ENABLE | GPIO_EDGE_TRIGGER,
97      GPIO_EN_INT_FALLING_EDGE      = GPIO_INTERRUPT_ENABLE | GPIO_EDGE_TRIGGER | GPIO_TRIGGER_NEG,
98      GPIO_EN_INT_BOTH_EDGE         = GPIO_INTERRUPT_ENABLE | GPIO_EDGE_TRIGGER | GPIO_EDGE_TRIGGER_BOTH,
99
100
101     /// GPIO Output Buffer Control and Output Value Multiplexing Control
102     /// GPIO configuration bit 4:5, and 14 output enable control and
103     /// muxing control
104     GPIO_INPUT_ENABLE             = 0x0000,
105     GPIO_OUTPUT_DISABLE           = 0x0000,
106     GPIO_OUTPUT_ENABLE            = 0x4000,
107     GPIO_KS_OUTPUT_ENABLE         = 0x0001, //Keyscan Output enable
108     GPIO_OUTPUT_FN_SEL_MASK       = 0x0000,
109     GPIO_OUTPUT_FN_SEL_SHIFT      = 0,
110
111
112     /// Global Input Disable
113     /// GPIO configuration bit 6, "Global_input_disable" Disable bit
114     /// This bit when set to "1" , P0 input_disable signal will control
115     /// ALL GPIOs. Default value (after power up or a reset event) is "0".
116     GPIO_GLOBAL_INPUT_ENABLE      = 0x0000,
117     GPIO_GLOBAL_INPUT_DISABLE     = 0x0040,
118
119
120     /// Pull-up/Pull-down
121     /// GPIO configuration bit 9 and bit 10, pull-up and pull-down enable
122     /// Default value is [0,0]--means no pull resistor.
123     GPIO_PULL_UP_DOWN_NONE        = 0x0000,   //[0,0]
124     GPIO_PULL_UP                  = 0x0400,   //[1,0]
125     GPIO_PULL_DOWN                = 0x0200,   //[0,1]
126     GPIO_INPUT_DISABLE            = 0x0600,   //[1,1] // input disables the GPIO
127
```

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

Note that right-clicking and selecting "Open Declaration" on function names and data-types inside WICED Studio is often very useful in finding information on how to use functions and what values are allowed for parameters.

2.3 Creating a new WICED Studio project

2.3.1 Directory Structure

A WICED Studio project can be located anywhere within the apps folder of the SDK Workspace. For convenience, it is often easier to create a folder for all your projects. You can also copy an existing example project to a new name or folder rather than starting from scratch. The key parts of a project are:

A folder with the name of the project.

A makefile called makefile.mk inside the project folder.

A C source file (usually called <project>.c) inside the project folder.

IMPORTANT: Do NOT use "File -> New" to create a new project unless you are using the SuperMux Tool which will be described later.

2.3.2 makefile

The makefile contains the list of all source files (including <project>.c). It may also define macros to provide access to libraries, and other C flags, etc.

2.3.3 C file

There will be various #include lines required at the top of main.c depending on the resources used in your project. These files can be found in the SDK under the platform or include folders. A few examples are shown below. The first 4 are usually required in any project.

```
#include "wiced.h"                  // Basic formats like stdint, wiced_result_t, WICED_FALSE, WICED_TRUE
#include "wiced_platform.h"         // Platform file for the kit
#include "sparcommon.h"             // Common application definitions
#include "wiced_bt_stack.h"         // Bluetooth Stack
#include "wiced_bt_dev.h"           // Bluetooth Management
#include "wiced_bt_ble.h"           // BLE
#include "wiced_bt_gatt.h"          // BLE GATT database
#include "wiced_bt_uuid.h"          // BLE standard UUIDs
#include "wiced_rtos.h"             // RTOS
#include "wiced_bt_app_common.h"    // Miscellaneous helper functions including wiced_bt_app_init
#include "wiced_transport.h"        // HCI UART drivers
#include "wiced_bt_trace.h"         // Trace message utilities
#include "wiced_timer.h"            // Built-in timer drivers
#include "wiced_hal_i2c.h"          // I2C drivers
#include "wiced_hal_adc.h"          // ADC drivers
#include "wiced_hal_pwm.h"          // PWM drivers
#include "wiced_hal_puart.h"        // PUART drivers
#include "wiced_rtos.h"             // RTOS functions
#include "wiced_hal_nvram.h"        // NVRAM drivers
#include "wiced_hal_wdog.h"         // Watchdog
```

The main entry of the application is a function called APPLICATION_START. That function typically does a minimal amount of initialization then it starts the Bluetooth stack and registers a stack callback function by calling *wiced_bt_stack_init()*. The Bluetooth stack callback function then typically controls the rest of the application based on Bluetooth events. Most application initialization is done once the Bluetooth stack has been enabled. That event is called BTM_ENABLED_EVT in the callback function. The full list of events from the Bluetooth stack can be found in the file include/20719/wiced_bt_dev.h file.

A minimal C file for an application will look something like this:

```
#include "wiced.h"
#include "wiced_platform.h"
#include "sparcommon.h"
#include "wiced_bt_dev.h"

/*****************************    Function Prototypes   ******************/
wiced_result_t bt_cback( wiced_bt_management_evt_t event, wiced_bt_management_evt_data_t *p_event_data);

/*****************************    Functions   *****************************/
/* Main application. This just starts the BT stack and provides the callback function.
 *  The actual application initialization will happen when stack reports that BT device is ready. */
APPLICATION_START( )
{
   /* Add initialization required before starting the BT stack here */
   wiced_bt_stack_init( bt_cback, NULL, NULL ); /* Register BT stack callback */
}

/* Callback function for Bluetooth events */
wiced_result_t bt_cback( wiced_bt_management_evt_t event, wiced_bt_management_evt_data_t *p_event_data)
{
   wiced_result_t result = WICED_SUCCESS;

   switch( event )
   {
      /* BlueTooth stack enabled */
      case BTM_ENABLED_EVT:
         /* Initialize and start your application here once the BT stack is running */
         break;
      default:
         break;
   }
   return result;
}
```

2.3.4 Make Target

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

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

- <folder1> is the name of the folder below the apps folder.

- <folder2>, <folder3>, etc., are the rest of the path down to the project name. There can be as many or as few additional folder names as you want. Use a period to separate the folder names.
- <project> is the name of the project. The folder and makefile must have the same name.
- <platform> is the name of the hardware platform (i.e. kit). There must be an entry in the platforms directory that matches the name provided here.

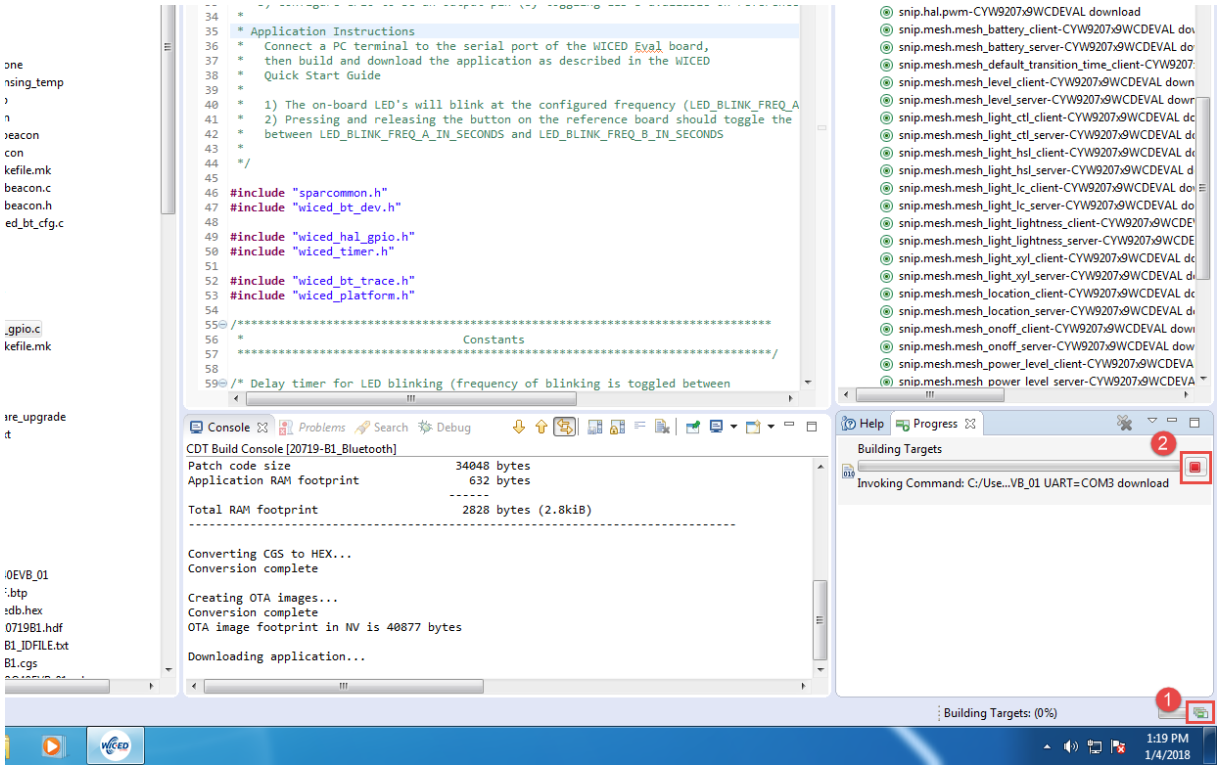
For example, if we create a folder called "wbt101" for our class projects and a subfolder called "ch02" for the chapter 2 projects, and call the first project "ex02_blinkled", the build target for our board (assuming we are using the shield with the CYW920719Q40EVB_01 as the baseboard) would be:

*wbt101.ch02.ex02_blinkled-WBT101_2_* CYW920719Q40EVB_01 *download*

The make targets that are already defined can be seen in the "Make Target" window along the right side of WICED Studio. Expand "20719-B1_Bluetooth" to see the existing make targets.

To create a new make target, you can right click on an existing make target that is similar to what you want to create and select *New…* This will give you a copy of the make target with "*Copy of* " at the beginning of the name. Delete "*Copy of* " (don't forget to remove the space!) and change the name as necessary for your new make target.

Once you have a make target, you can build the project and program the kit by just double clicking on it. ***IMPORTANT: Do NOT use "Project -> Build Project". It will NOT work.*** You can see the build progress in the *Console* window. If you need to kill a build that is in progress, you can click on the lower right corner of the IDE to open the *Progress* window and then click on the red box next to the build as shown below.



If the build fails with the following message, make sure your kit is plugged into a USB port on your laptop!

```
Detecting device...
+-----------------------------------------------------------------------------------------------+
| No CYW207x9 device detected.                                                                  |
| 1. Verify the CYW207x9 WICED eval board is connected _AND_ powered                            |
| 2. Verify all switches are set to the default positions                                       |
|    - see "Connect the WICED Evaluation Board" in the Quick Start Guide or Kit Guide           |
|      for defaults                                                                             |
| 3. Press the reset button on the WICED eval board and retry                                   |
|                                                                                               |
| See 20719-B1_Bluetooth/README.txt for more info.                                              |
| If this problem persists, the board EEPROM may need to be reset to factory defaults.          |
| Please see Recovery instructions in the Quick Start Guide or Kit Guide.                       |
+-----------------------------------------------------------------------------------------------+
```

If the build still fails with the same message, look in the device manager to make sure the drivers for the kit were properly installed. The board should show up as two devices under Ports (COM & LPT):

WICED HCI UART (COMxx)
WICED Peripheral UART (COMxx)

If you see anything listed in "Other devices" such as USB Serial Port, right click on each device, select "Update Driver Software", "Browse my computer for driver software", and then browse to the SDK installation folder (e.g. C:\Users\<username>\Documents\WICED-Studio-6.1). Make sure the box to Include subfolders is checked and click next. The driver should then install automatically.

Alternately, you can also install the drivers from WICED Studio. To use that method, in the project explorer go to "Drivers/Windows/uart", right click on the file "DPInst_x64.exe" (for 64-bit machines) or "DPInst.ext" (for 32-bit machines), and choose "Open With -> System Editor".

Common Build Errors

If anything went wrong during the build, carefully check the following items:

1. The make file has the correct name for the C source code file.
2. The make target has the correct names, paths, and spelling.
3. The folder hierarchy of the project is accurately represented in the make target.

Scroll through the Console window and look for error messages:

1. **No rule to make target** usually means you have a spelling error in the C source file name in the make file or a path error in the make target.
2. **Platform makefile not found** usually means that you have an error in the platform name in the make target or the platform files are not properly installed.
3. **Download failed** usually means that your kit is not connected, the device drivers are not installed, or the kit needs to be in recovery mode (reset the kit while holding the recover button to enter recovery mode). Recovery mode is sometimes required for the tool to acquire the kit for programming.

2.4 Pin Configuration (SuperMux Tool)

2.4.1 Pin Configuration File

The 20719 device contains multiple drivers on many of the pins that are multiplexed together. That is, many of the pins can be configured for one of several different functions such as GPIO, SPI, etc.

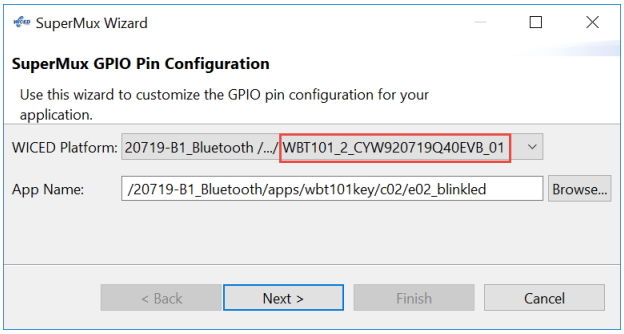
As discussed earlier the default pin mapping for the kit is in the wiced_platform_pin_config.c file, but this mapping can be over-ridden for a given project by placing a file called <project_name>_pin_config.c in the project folder, where <project_name> is the name of the project. Note that you don't need to change the pin mapping unless you want to change the kit's default pin behavior for a specific application requirement – usually the default pin mapping for the kit will do what you want.

2.4.2 SuperMux Configuration

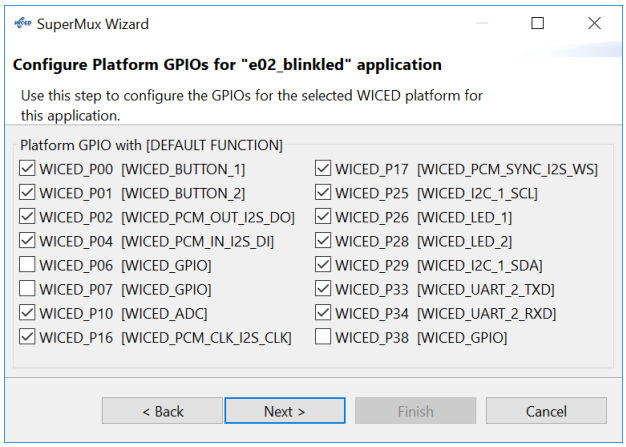
To simplify the creation of a custom pin configuration file for a project, there is a utility called the SuperMux Configuration tool. To run the tool, select the project folder for which you want to create a custom configuration file and then chose "File -> New -> WICED SuperMux GPIO Pin Configuration".

From the *WICED Platform* drop-down list, select the appropriate platform. Note that each platform has its own selection including the platform that represents the combination of the shield and base board. The *App Name* field displays the name of the application that was selected when the tool was launched.

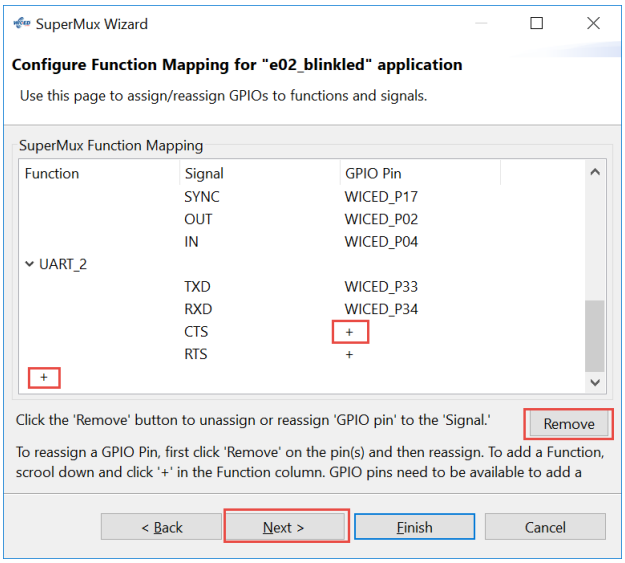
To select a different application, click *Browse* and select the appropriate application. Click *Next* to continue.



The SuperMux Wizard reads the platform configuration template from the selected WICED Platform folder (*20719-B1_Bluetooth\platforms\< name>\SuperMuxConfig.wst*). The Configure Platform GPIOs window lists the available GPIOs for the selected platform. The device GPIOs used in the platform by default are selected and the default function for each pin is displayed within brackets. You can select or clear GPIOs to specify which ones you want to use for your application. Clearing the checkbox corresponding to a GPIO pin will remove the pin's availability in the next step. After selecting the GPIOs you want to use, click *Next*.



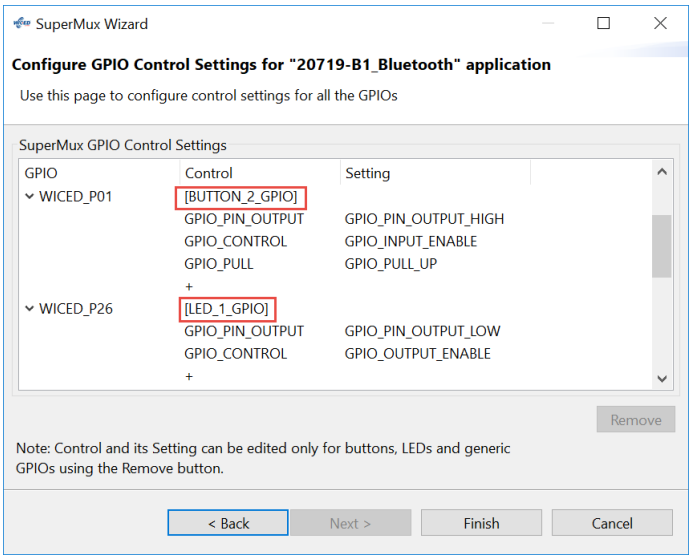
The next window is the *Function Mapping* window, which allows you to select the functions required and to map each function's signals to the desired GPIOs.



A few notes on using the *Function Mapping* window:

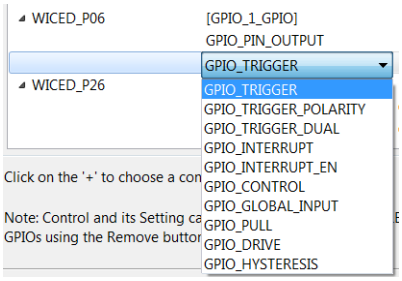
1. You can only assign a GPIO to a signal if it is not already assigned to a different signal.
2. To remove an existing GPIO signal assignment, select the pin and click the *Remove* button. Removing a function will also remove its GPIO pin assignments.
3. Click the "+" in the GPIO Pin column to assign a GPIO to a signal.
4. To remove a function, select the function and click the *Remove* button.
5. Click the "+" in the *Function* column (scroll to the bottom to see this) to add a new function.
6. You can only add a new function if there are unused pins.
7. Some functions such as SPI, I2C and PCM need to have pins configured for every available signal. Other functions such as UART have some required signals (e.g. TXD and RXD) and other signals that are optional (e.g. CTS and RTS). If a required signal is not assigned to a pin, the *Next* button will not be enabled.
8. When you have completed function to pin mapping, click *Next*.

The final SuperMux Configuration window is the *GPIO Control Settings* window. This window allows you to select the configuration options for GPIO pins. For example, you can select a pin to be an input or output, you can configure resistive pull up or pull down, and you can set the initial drive state for output pins. Pins with the function set as LED or BUTTON in the previous step will default to the appropriate selections for those functions but they can be overridden if desired.

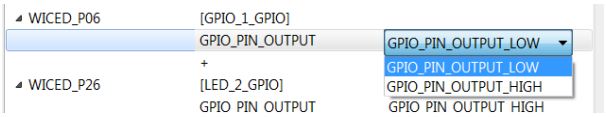


A few notes on using the *GPIO Control Settings* window:

1. Each GPIO pin is controlled by a control register. The Control column represents the bit fields of the control register. The Setting column represents the value of the specified bit field in the control register.
2. Some control fields are required and some are optional. The Finish button will not be active unless all required control fields are set.
3. Click the + sign from the Control column to add control fields. You can add controls for output pins, interrupts, drive strength, and so on.



4. Click the + sign from the Setting column to select the value for a given control bit field. For example, if you add a GPIO_PIN_OUTPUT control field, you can select a setting for the pin's initial state to be GPIO_PIN_OUTPUT_LOW or GPIO_PIN_OUTPUT_HIGH.



5. Click *Remove* to remove undesired settings. After a setting is removed, you can click the **+** sign from the Setting column to select a new setting, or click *Remove* again to remove the control field for that GPIO.

6. To change an existing setting, you must first remove the existing setting by selecting it and clicking *Remove*.

Once you have made all the desired selections, click *Finish*. The tool will then create an application specific pin configuration file called <app>_pin_config.c and a SuperMux Configuration file called <app>_pin_config.wsm in the application directory. It will also update the makefile.mk to include the new pin configuration file in the application.

You can re-run the SuperMux Configuration tool by double clicking on the <app>_pin_config.wsm file inside the folder for your project from the Project Explorer windwo. Note, you must first make sure the file is not open in an editor window. When you re-run the tool, it will create backup files (.bak) of any file that it modifies in your project folder.

2.5 Peripherals

2.5.1 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 for your platform such as WICED_GPIO_PIN_LED_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 most platforms, there is a helper function called *wiced_platform_register_button_callback()* which can be used to configure the button pin as an interrupt and register the callback function instead of calling them separately. For the helper function you need to use the button number rather than the button pin name. As an example, the two ways of configuring an interrupt for button 1 are shown below. Note how the separate functions use *WICED_GPIO_PIN_BUTTON_1* while the helper function uses *WICED_PLATFORM_BUTTON_1*.

Method 1:

```
wiced_hal_gpio_register_pin_for_interrupt( WICED_GPIO_PIN_BUTTON_1,
        gpio_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 );
```

Method 2:

```
wiced_platform_register_button_callback( WICED_PLATFORM_BUTTON_1,
        gpio_interrupt_callback, NULL, GPIO_EN_INT_FALLING_EDGE);
```

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(0)*. For example:

```
void gpio_interrupt_callback(void *data, uint8_t port_pin)
{
   /* Clear the gpio interrupt */
   wiced_hal_gpio_clear_pin_interrupt_status( WICED_PLATFORM_BUTTON_1 );

   /* 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 GPIO this is done automatically before the callback is executed and so it is not strictly necessary.

2.5.2 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 (a.k.a ACLK1) which is configurable.

You must include the PWM header file to use the PWMs:

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

To initialize a PWM block you need to specify the PWM to be used and the pin to be connected to it, and then call the start function. For example:

```
wiced_hal_pwm_configure_pin( WICED_GPIO_PIN_LED_1 ,PWM0 );
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. For example, if you 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_configure_pin (WICED_GPIO_PIN_LED_1, PWM0 );
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.

2.5.3 Debug Printing

The kit has two separate UART interfaces –the HCI UART (Host controller interface UART) and the PUART (peripheral UART) and. The HCI UART interface is used for programming the kit and often is used for 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.

There are 3 things required to allow debug print messages:

1. Place the following in the makefile.mk:

   C_FLAGS += -DWICED_BT_TRACE_ENABLE

2. Include the following header in the 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 in the WICED_BT_TRACE function. For example:

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

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

2.5.4 PUART

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 project:

```
#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\r");
/* 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("\r");
```

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 */
}
```

2.5.5 I2C

There is an I2C master on the device called WICED_I2C_1 which is routed by default to the Arduino header dedicated I2C pins. These pins connect to the OLED display and the PSoC on the shield.

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);
```

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.

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

```
#define I2C_ADDRESS (0x42)
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 (0x42)
uint8_t TxData[2] = {0x55, 0xAA};
uint8_t RxData[10];
wiced_hal_i2c_combined_read( TxData, sizeof(TxData), RxData, sizeof(RxData), I2C_ADDRESS );
```

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 byte called "control" followed by a 32-bit float called "temperature", you could set up a buffer like this:

```
struct {
        uint8_t control;
        float temperature;
} __attribute__((packed)) buffer;
```

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

2.5.6 OLED Display

TBD XXXXXXXXXXXXXXXXXXXX

2.5.7 ADC

The device contains a 16-bit signed ADC (-32768 to +32767).

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. When you read a sample, you must specify which channel to read from. 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 the ambient light sensor which is connected to GPIO WICED_P10, you would do the following:

```
#define ADC_CHANNEL     (ADC_INPUT_P10)

wiced_hal_adc_init();
raw_val = wiced_hal_adc_read_raw_sample( ADC_CHANNEL );
voltage_val = wiced_hal_adc_read_voltage( ADC_CHANNEL );
```

2.6 WICED_RESULT_T

Throughout the WICED 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 in WICED Studio, 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 ),   \
```

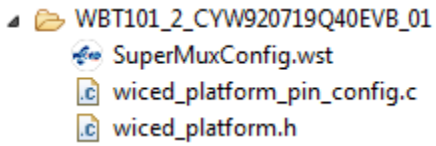
WICED_BT_*:

```
#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*/
```

2.7 Exercises

Exercise - 2.1 (PLATFORM) Install WW101_2_<KitName> into the platforms directory

1. Use what you learned in the fundamentals to install the files for the appropriate kit/shield combination into your SDK Workspace.
    a. Remember, the platforms folder is in the class material "WBT101_Files" folder.
2. Once you have installed the platform files, right click on the platform folder from inside WICED Studio and choose "Refresh". Once you do this, you should see the platform folder (e.g. WW101_2_CYW920719Q40EVB_01) and files. If you do not see them, ask for help – don't go forward until the platform is properly installed.



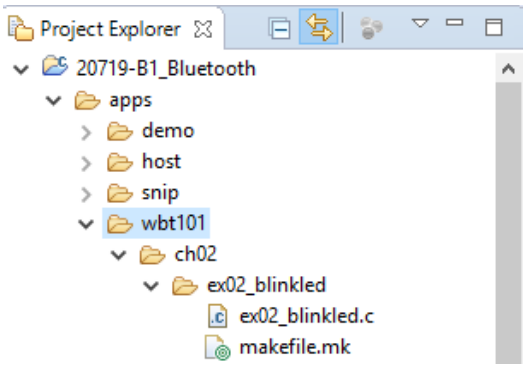
Questions to answer:

Which chip GPIOs are used for the I2C SCL and SDA pins?

Are the button pins pulled up or down? Where is that specified?

Exercise - 2.2 (GPIO) Blink an LED

1. Create a folder inside the SDK Workspace *20719-B1_Bluetooth/apps* folder called "wbt101" and a sub-folder called "ch02".
2. Copy the folder from the class files at WBT101_Files/templates/ch02/ex02_blinkled into the ch02 folder for your workspace. (You can just drag/drop from windows explorer into the WICED Studio project explorer.) When you finish, it should look like this:



3. Examine ex02_blinkled.c and makefile.mk to make sure you understand what they do.

4. All WICED BLE applications are multi-threaded (the BLE 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 but, in these exercises, we have shown you how to create a thread and associate it with a function for the code you will write.
5. Add code to 02_blinkled.c in the led_control thread function as indicated in the comments to do the following:
    a. Read the state of WICED_GPIO_PIN_LED_1
    b. Drive the state of WICED_GPIO_PIN_LED_1 to the opposite value.
6. Create a make target for your new project.
    a. Hint: If you right click on an existing make target and select "New" the target name will start out as "Copy of " followed by the existing target name. This makes it easy to setup a new target from an existing one that is similar. Make sure you remove "Copy of " from the beginning of the new target's name (including the space after "of ").
7. Program your project to the board.
    a. Hint: Be sure to save the files before building or else you will be building the old project. You can set "Window > Preferences > General > Workspace > Save automatically before build" if you want WICED Studio to save any changed files automatically before every build (this may be set by default).

Questions to answer:

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

What is the purpose of the function bt_cback? When does the BTM_ENABLED_EVT case occur?

What controls the rate of the LED blinking?

Exercise - 2.3 (GPIO) Add Debug Printing to the LED Blink Project

1. Copy your project from ex02_blinkled to ex03_blinkled_print. Rename the C file, modify the makefile as needed and create a make target.
    a. Hint: This can either be done from Window's Explorer, or it can be done from inside WICED Studio by using right-click, copy, paste, and rename.
2. Add WICED_BT_TRACE calls to display "LED LOW" and "LED HIGH" at the appropriate times.
    a. Hint: Remember to add the required include for the wiced_bt_trace.h header file.
    b. Hint: Remember to set the debug UART to WICED_ROUTE_DEBUG_TO_PUART.

c. Hint: Remember to use \n\r to create a new line so that information is printed on a new line each time the LED changes.
d. Hint: Don't forget to add the C flag to the makefile:

```
C_FLAGS += -DWICED_BT_TRACE_ENABLE
```

3. Program your project to the board.
4. Open a terminal window with a baud rate of 115200 and observe the messages being printed.
    a. Hint: if you don't have terminal emulator software installed, you can use putty.exe which is included in the class files under "Software_tools". To configure putty:
        i. 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.
        ii. Go to the session tab, select the Serial button, and click on "Open".

Exercise - 2.4 (GPIO) Read the State of a Mechanical Button

1. Copy the ex03_blinkled_print project to ex04_button, rename the C file, update the makefile, and create a make target.
2. In the C file:
    a. Change the thread sleep time to 100ms.
    b. In the thread function, check the state of the kit's button input (use WICED_GPIO_PIN_BUTTON_1). Turn on WICED_GPIO_PIN_LED_1 if the button is pressed and turn it off if the button is not pressed.
3. Program your project to the board.

Exercise - 2.5 (GPIO) Use an Interrupt to Toggle the State of an LED

1. Copy the ex04_button project to ex05_interrupt, rename the C file, update the makefile, and create a make target.
2. Remove the code from the previous example that blinks the LED – the calls to wiced_rtos_create_thread(), wiced_rtos_init_thread() and delete or comment out the thread function.
3. In the C file, set up a falling edge interrupt for the GPIO connected to the button and register the callback function.
4. Create the interrupt callback function so that it toggles the state of the LED each time the button is pressed.
5. Program your project to the board.

Exercise - 2.6 (I2C WRITE) Toggle 4 I2C Controlled LEDs

1. Copy ex02_blink to ex06_i2cwrite. Rename the C file, update the makefile, and create a make target.
2. Update the code so that instead of blinking one LED every 250ms, it will toggle between the four LEDs next to the CapSense buttons which are controlled by the PSoC on the shield board. The PSoC AFE shield contains an I2C slave with the following properties:
    a. Connected to dedicated I2C Arduino pins

b. 7-bit address = 0x42
c. Speed up to 400 kHz
d. EZI2C register access
    i. The first byte written is the register offset.
    ii. All reads start at the previous write offset.
e. The register map is as follows:

Offset	Description	Details
0x00–0x03	DAC value	This value is used to set the DAC output voltage
0x04	LED Values	4 least significant bits control CSLED3-CSLED0
0x05	LED Control	Set bit 1 in this register to allow the LED Values register to control the LEDs instead of the CapSense buttons
0x06	Button Status	Captures status of the CapSense buttons, Proximity sensor, and Mechanical buttons The bits are: Unused, MB1, MB0, Prox, CS3, CS2, CS1, CS0
0x07–0x0A	Temperature	Floating point temperature measurement from the thermistor
0x0B–0x0E	Humidity	Floating point humidity measurement
0x0F–0x12	Ambient Light	Floating point ambient light measurement
0x13–0x16	Potentiometer	Floating point potentiometer voltage measurement

f. Hint: Don't forget to add the include the header file for the I2C functions: wiced_hal_i2c.h.
g. Hint: To control the LEDs using I2C, you must first write 0x01 to the LED Control Register (at offset 0x05). This only needs to be done once during initialization.
h. Hint: To turn on a given LED, set that LEDs bit in the LED Values Register (at offset 0x04). For example, writing 0x01 will turn on LED0 while 0x04 will turn on LED2.

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

Exercise - 2.7 (I2C READ) Read PSoC CapSense Button Values using I2C

1. Copy ex06_i2cwrite to ex07_i2cread_buttons. Rename the C file, update the makefile, and create a make target.
2. Update the code so that every 100ms the button values are read from the I2C slave. Mask out just the CapSense buttons (bits 3-0) and print the value to the terminal using WICED_BT_TRACE if they have changed since the last time they were read.
    a. Hint: Don't forget to enable WICED_BT_TRACE in the makefile, add the include for the wiced_bt_trace.h header file, and redirect the debug UART to the PUART. See the debug printing exercise if you need additional help.
    b. Hint: Remember to set the offset to 0x06 to read the button register. You can do this just once during initialization and it will stay set for all future reads.
3. Program your project to the board and test it.

Exercise - 2.8 (Advanced) (I2C READ) Read PSoC Sensor Values using I2C

1. Copy ex07_i2cread_buttons to ex08_i2cread_sensors. Rename the C file, update the makefile, and create a make target.

2. Update the code so that every 100ms the temperature, humidity, ambient light, and potentiometer values are read from the PSoC. Print the values to the terminal using WICED_BT_TRACE.
    a. Hint: Remember to set the offset to 0x07 to read the temperature. You can do this just once and it will stay set for all future reads. With an offset of 0x07 you can read 16 bytes to get the temperature, humidity, ambient light, and potentiometer values (4 bytes each). You may want to use a structure containing four float variables to hold the data.
    b. Hint: The WICED_BT_TRACE function does not support floating point values. Therefore, use the following macros to convert floating point numbers into an integer part and a one decimal place fraction part.

```
/* Convert float into integer-dot-integer values */
#define FABS(f)          ((f<0.0)?-f:f)
#define INTEGER(f)       ((int)f)
#define FRACTION(f)      ((int)((FABS(f)-INTEGER(FABS(f)))*10))
…
float x;
…
WICED_BT_TRACE( "x= %4d.%1d\r\n", INTEGER(x), FRACTION(x) );
```

3. Some of the Arduino analog pins are shared with other functions on the board. Specifically, A0 is shared with the thermistor on the baseboard, A2 is shared with PUART CTS and A3 is shared with PUART RTS. Remove the short from J14 to disconnect the thermistor and remove the shorting blocks from J13 pins 1-2 and pins 3-4 to disconnect the PUART functions.
4. Program the project to the board and test it.

Exercise - 2.9 (Advanced) (PWM) LED brightness

1. Copy the ex02_blinkled project to ex09_pwm, rename the C file, update the makefile, and create a make target.
2. In the C file, configure a PWM to drive WICED_GPIO_PIN_LED_1 with an initial period of 100 and a duty cycle of 50%.
    a. Hint: 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).
3. Update the duty cycle in the thread function so that the LED gradually cycles through intensity values from 0 to 100%.
    a. Hint: Change the delay in the thread function to 10ms so that the brightness changes relatively quickly.
4. Program the project to the board and test it.

Exercise - 2.10 (Advanced) (PWM) LED toggling at specific frequency and duty cycle

In this exercise, we will use a PWM with a period of 1 second and a duty cycle of 20% so that the LED will blink at a 1 Hz rate but will only by on for 200ms each second.

1. Copy the ex09_pwm project to ex10_pwm_blink, rename the C file, update the makefile, and create a make target.

2. In the C file, initialize the aclk with a frequency of 1 kHz.
3. Change the PWM configuration to use PMU_CLK as the source and change the duty cycle to 20%
4. As you did in ex05, remove or comment out the calls to wiced_rtos_create_thread(), wiced_rtos_init_thread() and all of the thread function because that code will interfere with your PWM
5. Program the project to the board and test it.

Exercise - 2.11 (Advanced) (ADC) Measure Ambient Light Sensor

In this exercise we will measure the output of the ambient light sensor circuit which is connected to Arduino header A0. The input is connected after a trans-impedance-amplifier (TIA) in the PSoC. The result is that you will get a higher voltage for lower light levels and vice-versa.

Note: a thermistor on the baseboard is also connected to A0. Remove jumper J14 to disconnect the thermistor from A0 for this project.

1. Copy ex03_blinkled_print to ex11_adc, rename the C file, update the makefile, and create a make target.
2. In the C file, initialize the ADC when the Bluetooth stack is enabled.
3. In the thread function, read the count and voltage from the ADC. Print both values to the UART.
    a. Hint: look at the back of the base board to determine which ADC channel to use for A0.
4. Program the project to the board. Open a terminal window with a baud rate of 115200. Use the flashlight on your cellphone to shine a light on the sensor and then cover it with your hand to see the range of values reported.

Exercise - 2.12 (Advanced) (UART) Send a value using the standard UART functions

1. Copy the ex05_interrupt project to ex12_uartsend, rename the C file, update the makefile, and create a make target.
2. Modify the C file to initialize the UART with Tx enabled, baud rate of 115200, and no flow control. 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.
3. Program your project to 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 - 2.13 (Advanced) (UART) Get a value using the standard UART functions

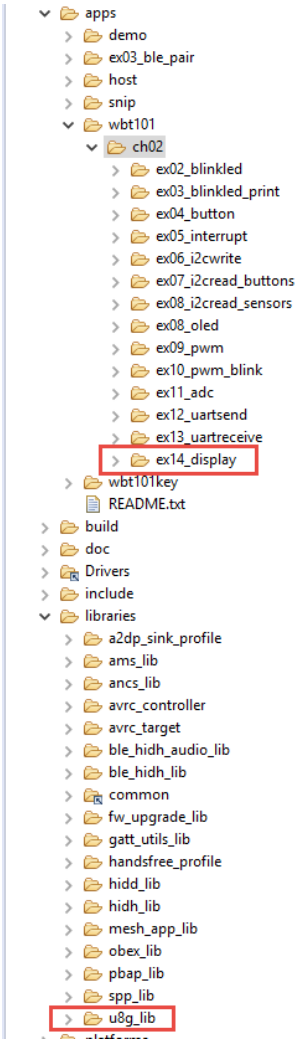
1. Copy ex11_uartsend to ex13_uartreceive, rename the C file, update the makefile, and create a make target.
2. Update the code to initialize the UART with Rx enabled, baud rate of 115200, no flow control, and an interrupt generated on every byte received.
    a. Hint: you can remove the code for the 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 an LED. If the byte is a 0, turn off an LED. Ignore any other characters.
4. Program your project 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 - 2.14 (Advanced) (I2C OLED) Display Data on the OLED Display

In this exercise you will use a library to display text and graphics on the OLED display on the shield.

1. Copy the folder from the class files at *WBT101_Files/templates/u8g_lib* into the *libraries* folder inside of WICED studio.
2. Copy the folder from the class files at *WBT101_Files/projects/wbt101key/ch02/ex14_display* into the folder *apps/wbt101/ch02* inside of WICED studio
3. When you are done with the two steps above it should look like this:



4. Create a make target for ex10_dispaly. Program the board and observe the OLED display.
5. Examine the project's makefile.mk to understand how to include the library and the desired fonts into your project.
6. Examine the makefile.mk inside the u8g_lib folder to learn how the library's source code is included.
7. Examine the C file to understand how to display text and graphics using the library,

2.8 Related Example "Apps"

App Name	Function
snip.hal.gpio	Demonstrates reading an input connected to a button and toggling an output driving LED.
snip.hal.puart	Demonstrates using the PUART to send and receive characters.
snip.hal.pwm	Demonstrates using the PWM to drive an LED.
snip.hal.adc	Demonstrates using the ADC to measure an analog voltage.
Snip.hal.i2c	Demonstrates using the I2C master with the motion sensor on the base board.

2.9 Known Errata + Enhancements + Comments

When you update to a new version of WICED, your settings, projects, and make targets don't get transferred over. This must all be done manually.