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

Objective

At the end of this chapter you should be able to write firmware for the MCU peripherals (GPIOs, PWMs, UART, and I2C) and to interface with the shield including the analog coprocessor, 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 (platform.h and platform.c).

Time: 2 Hours

Fundamentals

Setting up a new WICED board support package

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 shield which contains a PSoC analog co-processor chip. In order to make it easier to interface with the shield, a set of platform files has been created. Since this is not installed by default in the SDK we need to copy the platform folder into the SDK Workspace. The folder for this kit/shield combination is named “CYW943907AEVAL1F_WW101” and is provided with the class materials. Copy the entire folder from the class materials into the “platforms” directory in the SDK Workspace. The contents of CYW943907AEVAL1F_WW101 is:

- CYW943907AEVAL1F_WW101
  - board_revision
  - schematics
  - CYW943907AEVAL1F_WW101.mk
  - ota2_image_defines.mk
  - platform_config.h
  - platform_ota2_image.h
  - platform.c
  - platform.h
  - README.txt

Two key files here are platform.c and platform.h. The 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 platform.h using the names WICED_SH_LED0, WICED_SH_LED1, WICED_SH_MB0, and WICED_SH_MB1.

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

On this kit, there are also 2 mechanical buttons and 2 LEDs on the base board. These are also defined in the platform.h file:

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

The platform.c file contains several constant arrays and structures that are used to configure the peripherals. This file also contains the functions used to initialize and control the peripherals. For example, the LED pins are initialized as outputs and the button pins are initialized as inputs with a resistive pullup.

In platform.h you will also find a list of all of the valid peripherals. For example, there are 6 PWMs:

```
typedef enum
{
    WICED_PWM_1,
    WICED_PWM_2,
    WICED_PWM_3,
    WICED_PWM_4,
    WICED_PWM_5,
    WICED_PWM_6,
    WICED_PWM_MAX, /* Denotes the total number of PWM port aliases. Not a valid PWM alias */
} wiced_pwm_t;
```

The pins used for each PWM can be found in platform.c:

```
/* PWM peripherals. Used by WICED/platform/MCU/wiced_platform_common.c */
const platform_pwm_t platform_pwm_peripherals[] =
{
    [WICED_PWM_1]  = {PIN_GPIO_10,  PIN_FUNCTION_PWM0, },   /* or PIN_GPIO_0, PIN_GPIO_8,  PIN_GPIO_12, PIN_GPIO_14, PIN_GPIO_16, PIN_PWM_0   */
    [WICED_PWM_2]  = {PIN_GPIO_11,  PIN_FUNCTION_PWM1, },   /* or PIN_GPIO_1, PIN_GPIO_7,  PIN_GPIO_9,  PIN_GPIO_13, PIN_GPIO_15, PIN_PWM_1   */
    [WICED_PWM_3]  = {PIN_GPIO_16,  PIN_FUNCTION_PWM2, },   /* or PIN_GPIO_8, PIN_GPIO_0,  PIN_GPIO_10, PIN_GPIO_12, PIN_GPIO_14, PIN_PWM_2   */
    [WICED_PWM_4]  = {PIN_GPIO_15,  PIN_FUNCTION_PWM3, },   /* or PIN_GPIO_1, PIN_GPIO_7,  PIN_GPIO_9,  PIN_GPIO_11, PIN_GPIO_13, PIN_PWM_3   */
    [WICED_PWM_5]  = {PIN_PWM_4,    PIN_FUNCTION_PWM4, },   /* or PIN_GPIO_0, PIN_GPIO_8,  PIN_GPIO_10, PIN_GPIO_12, PIN_GPIO_14, PIN_GPIO_16 */
    [WICED_PWM_6]  = {PIN_GPIO_7,   PIN_FUNCTION_PWM5, },   /* or PIN_GPIO_1, PIN_GPIO_9,  PIN_GPIO_11, PIN_GPIO_13, PIN_GPIO_15, PIN_PWM_5   */
};
```

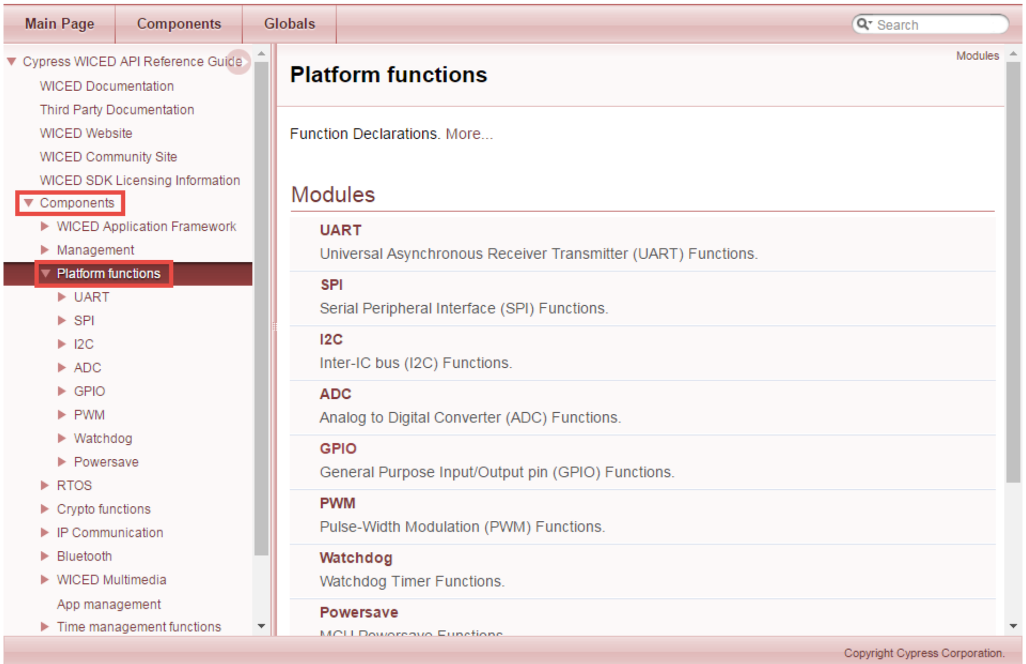
Note that the PWM names must be used in the PWM API function calls. That is, you must use *WICED_PWM_1* to use PWM 1. You cannot use *PIN_GPIO_10* in the PWM API function calls.

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.

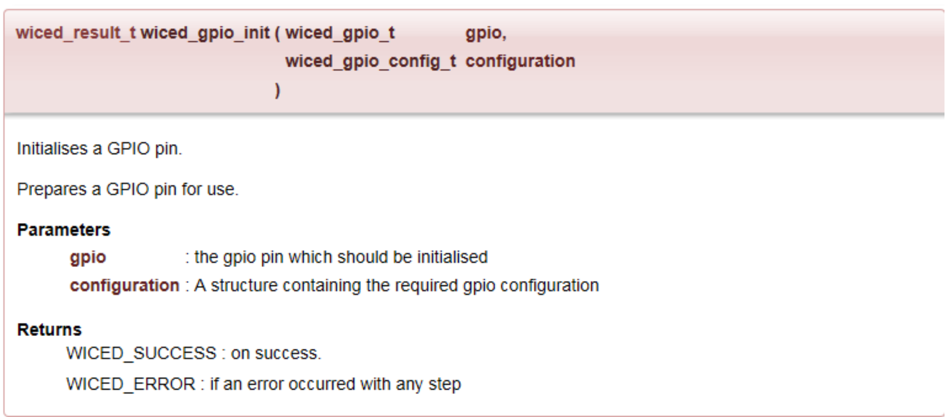
If you are working with a new kit, you may also need to add that platform to the list of valid platforms in the file located in the SDK at *apps/waf/tiny_bootloader/tiny_bootlader.mk*. This file is used by the make process.

Documentation

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 from a browser and expand "Components" and "Platform Functions" to see the list of supported components (you can open this file from inside WICED Studio too). We will be using GPIO, PWM, UART, and I2C.



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



```
wiced_result_t wiced_gpio_init ( wiced_gpio_t          gpio,
                                 wiced_gpio_config_t   configuration
                               )
```

Initialises a GPIO pin.

Prepares a GPIO pin for use.

Parameters

- **gpio** : the gpio pin which should be initialised
- **configuration** : A structure containing the required gpio configuration

Returns

WICED_SUCCESS : on success.

WICED_ERROR : if an error occurred with any step

The description tells you what the function does, but does not give information on the configuration structure. To find that information, once you are in WICED Studio you can highlight the parameter in the C code, right click, and select "Open Declaration" (you will try this later in the exercises). If you don't already have a valid parameter provided, you can also get there by using "Open Declaration" on the function name, then the parameter type, and then the type name. This will show you the datatype with an explanation of the allowed choices:

```
/**
 * Pin configuration
 */
typedef enum
{
    INPUT_HIGH_IMPEDANCE,       /* Input - must always be driven, either actively c
    INPUT_PULL_UP,              /* Input with an internal pull-up resistor - use wi
    INPUT_PULL_DOWN,            /* Input with an internal pull-down resistor - use
    OUTPUT_PUSH_PULL,           /* Output actively driven high and actively driven
    OUTPUT_OPEN_DRAIN_NO_PULL,  /* Output actively driven low but is high-impedance
    OUTPUT_OPEN_DRAIN_PULL_UP,  /* Output actively driven low and is pulled high wi
} platform_pin_config_t;
```

Creating a new WICED Studio project

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 copy an existing example project to a new name rather than starting from scratch. The key parts of a project are:

A folder with the name of the project.

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

A makefile called <project>.mk inside the project folder.

IMPORTANT: The <project> name must be the same for the folder name, C file name, and makefile.

The makefile contains the application name (any unique string), and the list of all source files (including <project>.c. It may also contain a list of valid and/or invalid platforms for the given project, makefile macros to provide access to libraries, and other resources such as images, web pages, etc. **The application name in the make file MUST BE A UNIQUE STRING.** If any two projects in the entire workspace have the same application name then the build may not work. In some cases the build may go into an infinite loop because the make target builds the files from the wrong project and then can't find the correct object files forcing it to continually request the (wrong) files to be re-built. Therefore, it is recommended that the complete project name including the folder path hierarchy be used in the application name.

Make Target

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

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

- <folder1> is the name of the folder below the apps folder.
- <folder2>, <folder3>, etc., are the rest of the path down to the project name. 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, main C file, and makefile must all 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.

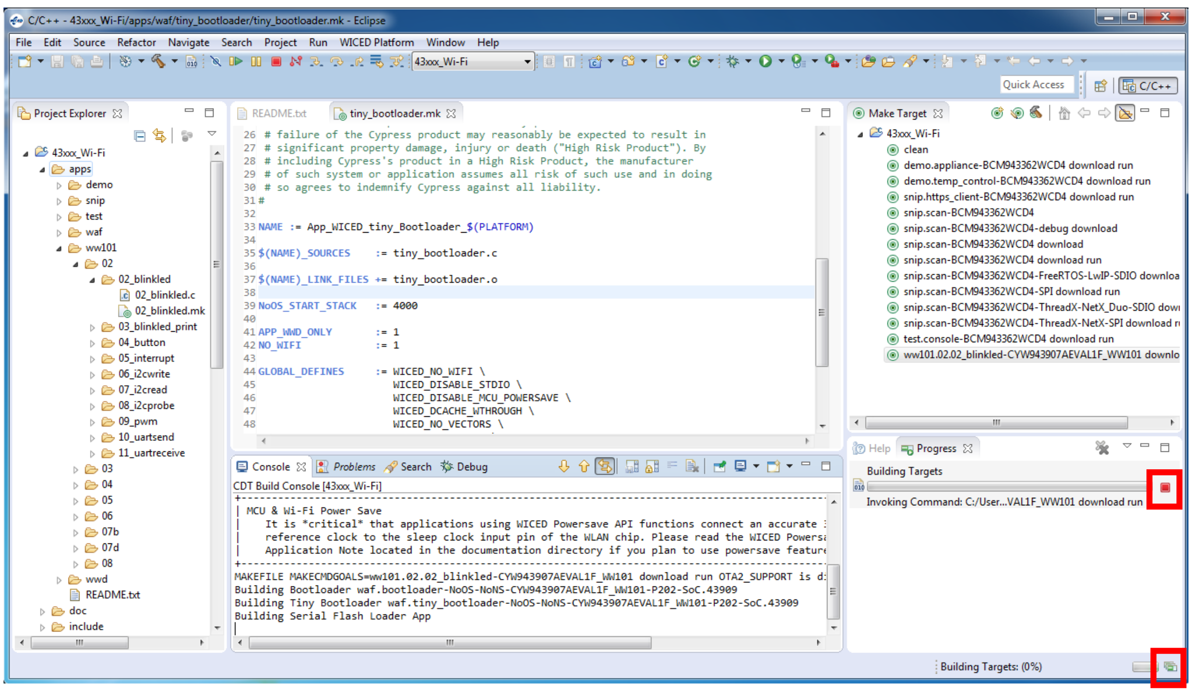
For example, if we create a folder called "ww101" for our class projects and a subfolder called "02" for the chapter 2 projects, and call the first project "02_blinkled", the build target for our board would be:

ww101.02.02_blinkled-CYW943907AEVAL1F_WW101 download run

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

To create a new make target 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. 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.



For some devices, the module contains two chips – the microcontroller and the Wi-Fi/Bluetooth radio. The make target option "download" just downloads the firmware to the microcontroller but does not affect the radio firmware. In most cases, that is all that is needed since the default radio firmware doesn't change. However, in some cases, you may need to modify the radio chip's firmware. In that case, you can download the radio firmware once by adding "download_apps" to the make target. That is, you would have:

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

C file

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

Peripherals

GPIO

As explained previously, GPIOs must be initialized before they are used. The IOs on the kit that are connected to specific peripherals such as LEDs and buttons are often automatically initialized for you as part of the platform files.

Once initialized, input pins can be read using *wiced_gpio_input_get()* and outputs can be driven using *wiced_gpio_output_high()* and *wiced_gpio_output_low()*. The parameter for these functions is the WICED pin name such as WICED_GPIO_1 or a peripheral name for your platform such as WICED_LED1.

GPIO interrupts are controlled using *wiced_gpio_input_irq_enable()* and *wiced_gpio_input_irq_disable()*.

PWM

The PWM has an API function to choose the pin, set the frequency (in Hz) and the duty cycle (in percent). This function is used for initialization and to change the frequency or duty cycle once the PWM is running. It also has functions to start and stop the output. See the API documentation for details.

In addition to initializing the PWM you must also call the start function for the parameters to take effect and for the PWM to generate an output. You should call the start function every time you update parameters using the init function.

If you are using a PWM on a pin that was initialized as a GPIO such as the LEDs on the shield, you must first call the pin deinit function before the PWM will be able to output a signal on that pin.

Entering a value for the frequency lower than that ~600Hz may result in an unexpected frequency.

Debug Printing

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

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

Note that the extra set of parenthesis is required due to the way the macro is defined.

Note that the \n is required to print a new line. The terminal may not actually print anything until its buffer is full or until a new line occurs.

UART

In addition to the USB-UART debug print functions, the device can also send standard UART data over the Arduino UART pins (D0 and D1) using STDIO_UART as defined in the “platform.h” file. These pins are also connected to the on-board USB-UART Bridge so the same terminal window used for the debug messages will work for standard UART communication too. On the CYW943907AEVAL1F kit, there is a second UART (called WICED_UART_2) connected to Arduino pins D8 and D9.

There are API functions for UART initialization, transmit, and receive. See the API documentation for details on these functions.

If you are using the STDIO_UART defined in the platform, then you don’t need to call the initialization function and you do not need to set up a ring buffer as described below since those functions are already called from *platform_stdio_init()* which is in turn called from “platform.c”. These are needed only if you are using a different UART interface or different UART settings. The STDIO_UART is by default set up for 115200 baud, 8 bit width, no parity, no flow control.

If you want to disable the STDIO_UART functionality or use that interface with different settings, add the following to the make file for the project:

```
GLOBAL_DEFINES := WICED_DISABLE_STDIO
```

Once you do this, you will no longer see the standard boot time information displayed on the terminal.

The UART initialization function requires a configuration structure of type *wiced_uart_config_t* with the following elements. This is defined in “platform_peripheral.h”. As mentioned above, you can find this structure by highlighting, right clicking, and selecting “Open Declaration” from inside WICED Studio on the function name, parameter type, and type name.

```
/**
 * UART configuration
 */
typedef struct
{
    uint32_t                    baud_rate;
    platform_uart_data_width_t  data_width;
    platform_uart_parity_t      parity;
    platform_uart_stop_bits_t   stop_bits;
    platform_uart_flow_control_t flow_control;
} platform_uart_config_t;
```

You can also use “Open Declaration” on each of the types inside the structure to find valid choices. For example, for the data width, the possible choices are:

```
/**
 * UART data width
 */
typedef enum
{
    DATA_WIDTH_5BIT,
    DATA_WIDTH_6BIT,
    DATA_WIDTH_7BIT,
    DATA_WIDTH_8BIT,
    DATA_WIDTH_9BIT
} platform_uart_data_width_t;
```

If you are using the UART to receive, you must provide a buffer of type *wiced_ring_buffer_t*. This buffer must be initialized using the *ring_buffer_init()* function which requires a pointer to the ring buffer, a pointer to an array to hold the data, and the size of the buffer. For example, the following could be used to create a 10 byte ring buffer called rx_buffer:

```
#define RX_BUFFER_SIZE (10)

wiced_ring_buffer_t rx_buffer;

uint8_t rx_data[RX_BUFFER_SIZE];

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

I2C

The device contains two I2C masters called WICED_I2C_1 and WICED_I2C_2. The OLED display and the PSoC analog co-processor on the shield connect to WICED_I2C_2.

As with other peripherals, you need to initialize the block using the initialization function. However, in this case, the parameter you pass it is not the name of the block, but a structure of the type *wiced_i2c_device_t*. That structure contains information about the I2C slave that you are going to communicate with. For example, the following could be used to initialize I2C block 1 to connect to a slave at address 0x08 with a speed of 100kHz (standard speed).

```
const wiced_i2c_device_t i2cDevice = {
        .port = WICED_I2C_2,
        .address = I2C_ADDRESS,
        .address_width = I2C_ADDRESS_WIDTH_7BIT,
        .speed_mode = I2C_STANDARD_SPEED_MODE
};
```

There are two ways to read/write data from the slave. There is a dedicated read function called and a dedicated write function called. There is also a function called wiced_i2c_transfer which can do a read, a write, or both. We will focus on wiced_i2c_trasnfer here, but feel free to look at the separate read/write functions in the documentation and experiment with them.

Before you can use the wiced_i2c_transfer function, you need to set up a message structure of type *wiced_i2c_message_t*. There are three functions that can be used for that purpose: *wiced_i2c_init_tx_message()*, *wiced_i2c_init_rx_message()*, *wiced_i2c_init_combined_message()*. See the API documentation for details on these functions. Note that the "retries" parameter must be set to a non-zero value (e.g. 1). A value of 0 means don't even try to send the message once.

<u>For the CYW943907AEVAL1F kit, I2C does not support DMA. Therefore, the "disable_dma" parameter in message initialization call must be set to WICED_TRUE. Otherwise, the I2C transfer will fail.</u>

Once the structure is setup, use the function *wiced_i2c_transfer()* to send or receive the message.

You can also use the *wiced_i2c_probe_device()* to check to see if there is an I2C slave at the given address. The function will return WICED_TRUE if a device is found and WICED_FALSE if a device is not found. Note that you must still initialize the device with *wiced_i2c_init()* before using *wiced_i2c_probe_device()*.

You may also 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 rx_buffer like this:

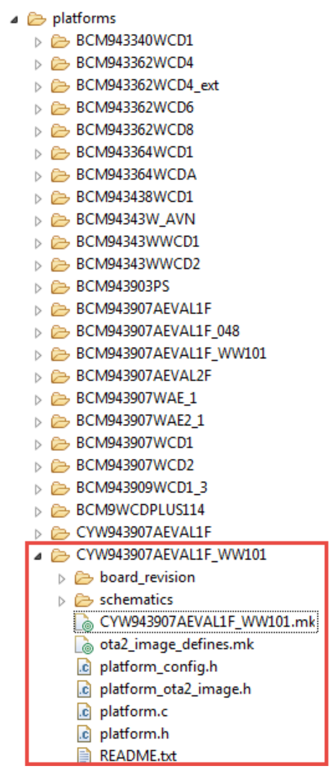
```
struct {
	uint8_t control;
	float temperature;
} __attribute__((packed)) rx_buffer;
```

Note that there are two underscores before and after the word “attribuite” and there are two sets of parenthesis around the word “packed”.

Exercises

01 (PLATFORM) Install CYW943907AEVAL1F_WW101 into the platforms directory

1. Use what you learned in the fundamentals to install the files for the CYW943907AEVAL1F_WW101 kit into your SDK Workspace.
    a. In addition to copying the platform files, open the file *apps/waf/tiny_bootloader/tiny_bootloader.mk* and add *CYW943907** to the list of valid platforms.
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 CYW943907AEVAL1F_WW101 folder and files. If you do not see them, ask for help – don't go forward until the platform is properly installed.



<u>Questions to answer</u>:

The table at the top of platform.h says that WICED_SH_LED0 connects to WICED_GPIO_7, Arduino header D10, and WICED_PWM_2. Explain how this mapping was determined. You will need to refer to platform.h, platform.c and the schematic for the base board.

02 (GPIO) Blink an LED

1. Create a folder inside the SDK Workspace *43xxx_Wi-Fi/apps* folder called "ww101" and a sub-folder called "02".
2. Inside the "02" folder, create a project folder called "02_blinkled".
3. Inside the "02_blinkled" folder, create files called "02_blinkled.c" and "02_blinkled.mk".
4. Copy the text as shown below into the .c and .mk files.
    a. Hint: you can copy/paste from the electronic copy of the manual to make this step easier.

   02_blinkled.c:

```
/* Blink LED1 on the shield with a frequency of 2 Hz */
#include "wiced.h"

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

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

    while ( 1 )
    {
             /* Add Code to Blink WICED_SH_LED1 here */
    }
}
```

   02_blinkled.mk:

```
NAME := App_WW101_02_02_blinkled

$(NAME)_SOURCES := 02_blinkled.c
```

5. Add code to 02_blinkled.c in the infinite loop as indicated to do the following:
    a. Drive WICED_SH_LED1 low
    b. Wait 250ms
    c. Drive WICED_SH_LED1 high
    d. Wait 250ms
        i. Hint: See the API documentation for the GPIO functions use to drive the LED high and low.
        ii. Hint: Use the *wiced_rtos_delay_milliseconds()* function for the delay.
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.

Questions to answer:

Why can’t you read the value of the LED using the *wiced_gpio_input_get()* function instead of using a variable to remember the state?

In what file and on what line does the WICED_SH_LED1 get assigned to the correct pin for this kit?

In what file and on what line is the pin connected to LWICED_SH_LED1 set as an output?

03 (GPIO) Add Debug Printing to the LED Blink Project

1. Copy your project from 02_blinkled to 03_blinkled_print. 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 WPRINT_APP_INFO calls to display “LED OFF” and “LED ON” at the appropriate times.
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”.

04 (GPIO) Read the State of a Button

1. Copy the 02_blinkled project to 04_button, update the makefile, and create a make target.
2. In the C file, check the state of the kit’s button input (use WICED_SH_MB1). Turn on WICED_SH_LED1 if the button is pressed and turn it off if the button is not pressed.
3. Program your project to the board.

05 (GPIO) Use an Interrupt to Toggle the State of an LED

1. Copy the 04_button project to 05_interrupt, update the makefile, and create a make target.
2. In the C file, set up a falling edge interrupt for the GPIO connected to the button.
   a. Hint: See the documentation for *wiced_gpio_input_irq_enable().*
   b. Hint: In your C code do the following:

        i. Type *wiced_gpio_input_irq_enable()* in your code.
        ii. Highlight *wiced_gpio_input_irq_enable()*, right click on it, and select "Open Declaration". This will show the required parameters for the function.
        iii. Highlight *wiced_gpio_irq_trigger_t*, right click on it, and select "Open Declaration".
        iv. Highlight *platform_gpio_irq_trigger_t*, right click on it, and select "Open Declaration".
        v. Identify the correct value to use for a falling edge interrupt.
    c. Hint: For the argument to pass to the interrupt handler, use NULL.
    d. Hint: For the interrupt handler function declaration, use (void* arg) for the argument list.
3. Create the interrupt service routine (ISR) so that it toggles the state of the LED each time the button is pressed.
    a. Hint: You can use a static boolean variable type in the ISR to remember the LED state:
        i. *static wiced_bool_t led1 = WICED_FALSE;*
4. Program your project to the board.

06 (I2C WRITE) Toggle I2C Controlled LEDs

1. Copy 05_interrupt to 06_i2cwrite. Update the makefile and create a make target.
2. Update the code so that when the button is pressed, it will toggle between the four LEDs next to the CapSense buttons which are controlled by the analog co-processor on the shield board. The analog co-processor shield contains an I2C slave with the following properties:
    a. Connected to Arduino pins D14 and D15 (WICED_I2C_2)
    b. 7-bit address = 0x42
    c. Standard Speed (100kHz)
    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: To control the LEDs using I2C, you must first write 0x01 to the LED Control Register (at offset 0x05).

g. 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.
h. Hint: In the ISR, just set a flag to force an I2C update. Do the I2C processing in the main application loop only when the flag is set. Make sure the flag variable is defined as a volatile global variable.
i. Hint: Make sure the number of re-tries is set to 1. A value of 0 means don't try to send the message at all!

07 (I2C READ) Read Analog Co-Processor Sensor Values over I2C

1. Copy 06_i2cwrite to 07_i2cread. Update the makefile and create a make target.
2. Update the code so that every time the button is pressed the temperature, humidity, ambient light, and Potentiometer data are read from the I2C slave. Print the values to the terminal using WPRINT_APP_INFO.
    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).

08 (Advanced) (I2C PROBE) Probe for I2C devices

1. Copy 06_i2cwrite to 08_i2cprobe. Update the makefile and create a make target.
2. Update the code so that every time the button is pressed a scan is done of every possible I2C address. Print the address of any devices found to the terminal (in hex) using WPRINT_APP_INFO.
    a. Hint: The I2C address is 7 bits. 0x00 is a special "All Call" address, and all values above 0x7C are reserved for future purposes, so the only valid addresses are 0x01 – 0x7B.
3. What addresses are found on the shield?
    a. Hint: There should be 3 – one for the PSoC analog co-processor, one for the OLED, and one for an external ADC on the baseboard.

09 (Advanced) (PWM) LED brightness

1. Copy the 02_blinkled project to 09_pwm, update the makefile, and create a make target.
2. In the C file, configure a PWM to drive WICED_SH_LED1 on the shield board instead of using the GPIO functions.
    a. Hint: The LED is connected to WICED_GPIO_17 so you need to find out which PWM is connected to that pin (look in the platform files).
    b. Hint: You must call wiced_gpio_deinit on WICED_SH_LED1 so that the PWM can drive the pin rather than the GPIO driver.
3. Configure the PWM and change the duty cycle in the main loop so that the LED gradually changes intensity.
    a. Hint: Don't forget to call the wiced_pwm_start function after you call the wiced_pwm_init function every time you change the PWM configuration.
    b. Hint: use a delay so that the intensity goes from 0% to 100% in one second.

10 (Advanced) (UART) Write a value using the standard UART functions

1. Copy the 05_interrupt project to 10_uartsend. Modify the makefile and create a make target.
2. Modify the C file so that the number of times the button has been pressed is sent out over the UART interface whenever the button is pressed. For simplicity, just count from 0 to 9 and then wrap back to 0 so that you only have to send a single character each time.
    a. Hint: Disable the STDIO_UART in the make file by adding the line:
        i. GLOBAL_DEFINES := WICED_DISABLE_STDIO
    b. Hint: Setup a UART configuration structure for a baud rate of 9600, data with of 8, no parity, 1 stop bit, and no flow control and initialize the UART.
    c. Hint: Set a flag variable inside the ISR and then do the UART send function in the main application loop. Make sure the flag variable is defined as a volatile global variable.
    d. Hint: use NULL for the read buffer since we will only be transmitting values.
3. Program your project to the board.
4. Open a terminal window with a baud rate of 9600.
    a. Hint: The kit will show up in the device manager under "Ports (COM & LPT)" as *"WICED USB Serial Port"*.
5. Press the button and observe the value displayed in the terminal.

11 (Advanced) (UART) Read a value using the standard UART functions

1. Copy 10_uartsend to 11_uartreceive. Update the makefile and create a make target.
2. Update the code so that it looks for characters from the UART. If it receives a 1, turn on an LED. If it receives a 0, turn off an LED. Ignore any other characters.
    a. Hint: you will need to setup a ring buffer to receive the UART characters.
    b. Hint: remove the code for the button press and its interrupt.
3. Program your project to the board.
4. Open a terminal window with a baud rate of 9600.
    a. Hint: The kit will show up in the device manager under "Ports (COM & LPT)" as *"WICED USB Serial Port"*.
5. Press the 1 and 0 keys on the keyboard and observe the LED turn on/off.

Related Example "Apps"

App Name	Function
snip.gpio	Demonstrates reading an input connected to a button and toggling an output driving LED.
snip.uart	Demonstrates using the generic WICED UART to send and receive characters.
snip.stdio	Demonstrates using the UART with STDIO operations.

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.