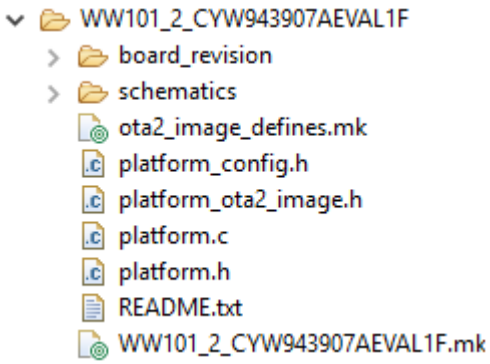
Chapter 2: Using the WICED SDK to Connect Inputs and Outputs to MCU Peripherals

Time 2 Hours

At the end of this chapter you will have WICED Studio installed and working on your computer and will understand how to program an existing project into a kit. You should be able to write firmware for the MCU peripherals (GPIOs, PWMs, UART, and I2C) 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 (platform.h and platform.c).

2.1 The 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 (AFE) shield which contains a PSoC 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 "WW101_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 "WW101_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 CYW943907AEVAL1F kit, the contents of WW101_2_CYW943907AEVAL1F is:



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_LED1, WICED_LED2, WICED_BUTTON1, and WICED_BUTTON2.

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. The names will stay the same – only the platform that we target will be different. This means if you have a project that uses LEDs and buttons, you can use the same project C file and make file to run it using just the baseboard or the baseboard plus shield by just changing the platform name in the Make Target!

```
382 /* LEDs and buttons on the shield */
383 #define WICED_LED1       ( WICED_GPIO_12 )
384 #define WICED_LED2       ( WICED_GPIO_6  )
385 #define WICED_BUTTON1    ( WICED_GPIO_10 )
386 #define WICED_BUTTON2    ( WICED_GPIO_8  )
```

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 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
};
```

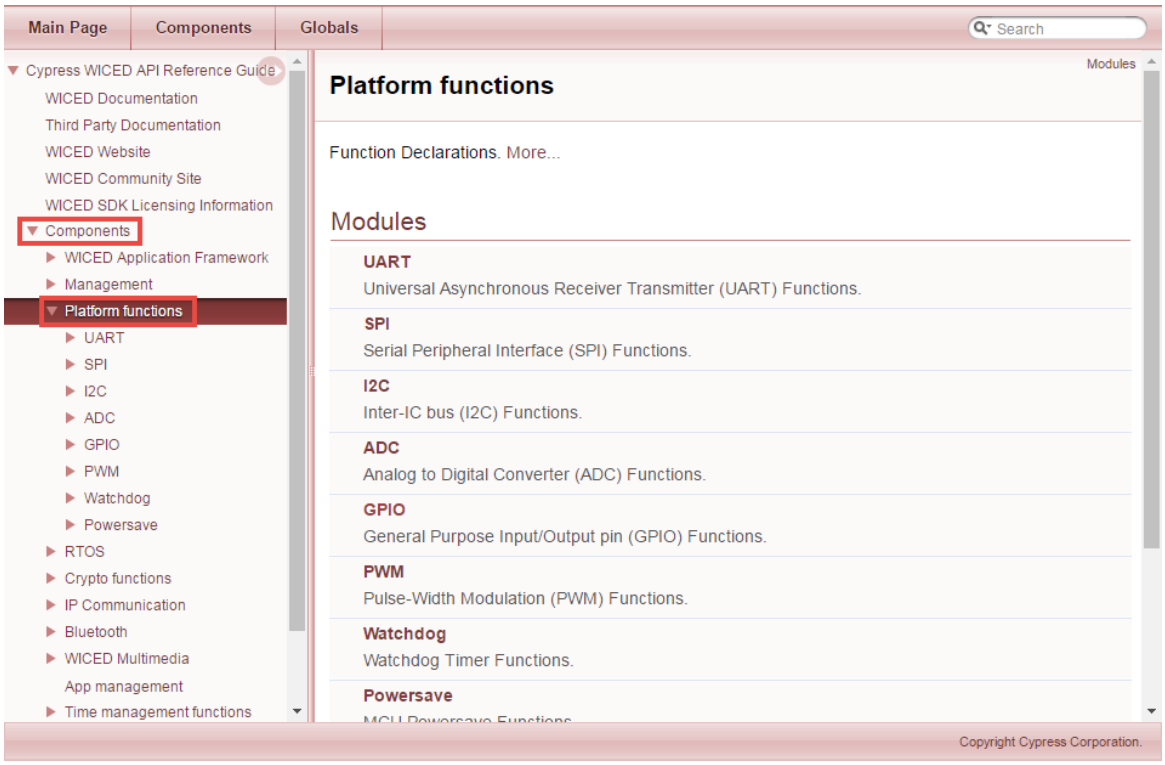
Note that the PWM names must be used in the PWM API function calls. That is, you must use *WICED_PWM_1* to use PWM 1. You cannot use *PIN_GPIO_10* in the PWM API function calls.

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.

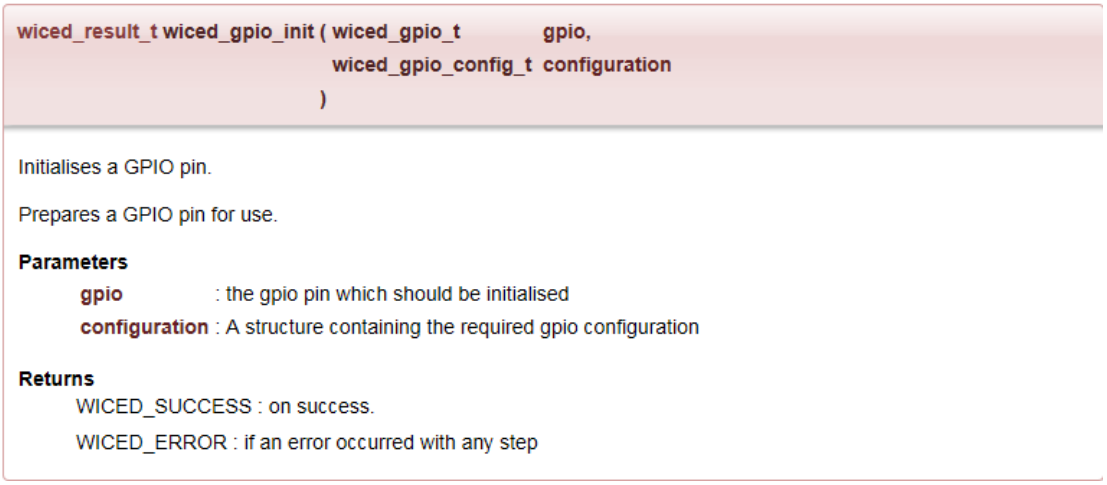
If you are working with a new kit, you may also need to add that platform to the list of valid platforms in the file located in the SDK at *apps/waf/tiny_bootloader/tiny_bootlader.mk*. This file is used by the make process.

2.2 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 by right-clicking on it and selecting *Open With > System Editor* and then expand "Components" and "Platform Functions" to see the list of supported components. 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.



The description tells you what the function does, but does not give information on the configuration structure that is required. 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;
```

2.3 Creating a new WICED Studio project

2.3.1 Directory Structure

A WICED Studio project (i.e. application) 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 makefile called <project>.mk inside the project folder.

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

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

IMPORTANT: Do NOT use "File -> New" to create a new project.

2.3.2 makefile

The makefile contains an 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. In addition, **the application name in the make file must not have any spaces at the end of the string**. If it does, the project will not build.

2.3.3 C file

You must have **#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 the initialization required to get the other WICED APIs to work properly and calls the functions that initialize the peripherals for the kit.

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 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 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.

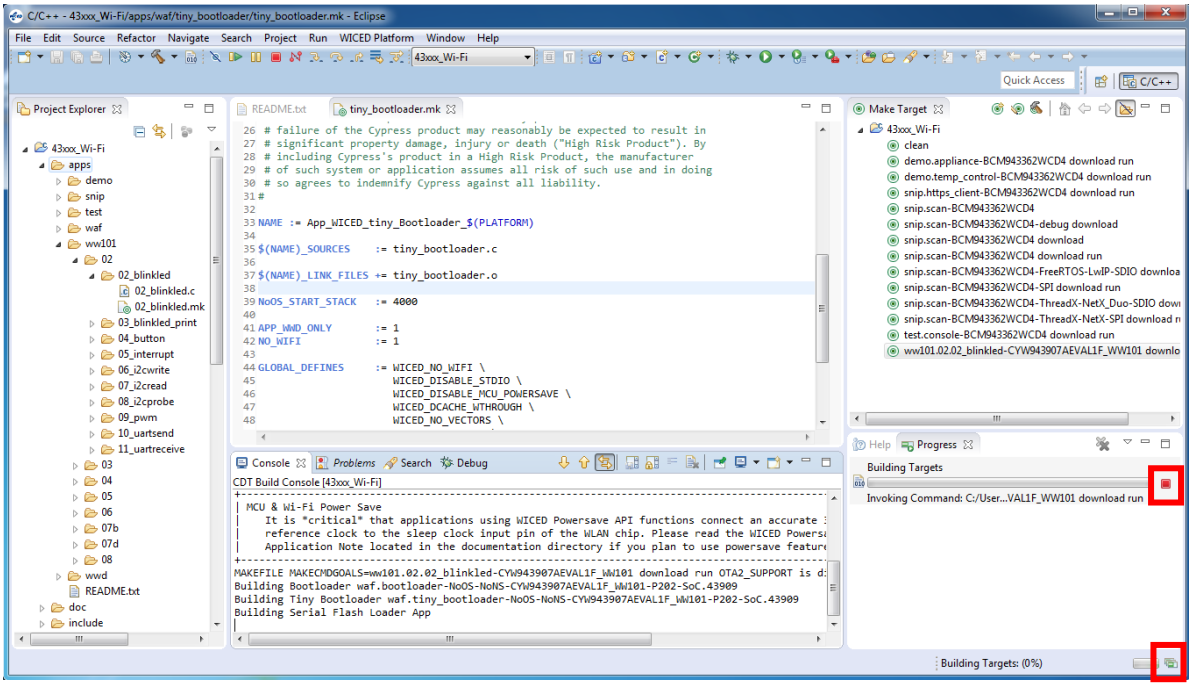
For example, if we create a folder called "ww101" for our class projects and a subfolder called "02" for the chapter 2 projects, and call the first project "02_blinkled", the build target for our board (assuming we are using the CYW943907AEVAl1F as the baseboard) would be:

ww101.02.02_blinkled-WW101_2_CYW943907AEVAL1F download run

The make targets that are already defined can be seen in the "Make Target" window along the right side of WICED 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. ***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 because it cannot find the target board, look in the device manager to make sure the drivers for the kit were properly installed. The board should show up as two devices:

Ports (COM & LPT) -> WICED USB Serial Port

Universal Serial Bus controllers -> WICED USB Serial Port B

If you see anything listed in "Other devices" such as BCM9WCD1EVAL1 and 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-5.0). Make sure the box to Include subfolders is checked and click next. The driver should then install automatically.

Alternately, you also install the drivers from WICED Studio. To use that method, in the project explorer go to "tools/drivers/BCM9WCD1EVAL1", right click on the file "dpinst_x64.exe" (for 64bit machines), and choose "Open With -> System Editor".

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

Common Build Errors

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

1. The project folder name and make file name are EXACTLY the same.
2. The make file has a unique application NAME and there are NO spaces at the end of the name.
3. The make file has the correct name for the C source code file.
4. The make target has the correct names, paths, and spelling.
5. 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 mismatch between the project folder name and the make file name or that you have a spelling error in the C source file name in the make file.
2. **Empty variable name** usually means you have a space after the application name in the make file.
3. **Unknown component…** usually means that your make target has an error.
4. **Recipe for target download_dct failed** usually means that your kit is not connected or the device drivers are not installed.
5. An error that says **undefined reference to `application_start'** usually means there is something wrong with the line in the make file that says ***$(NAME)_SOURCES := <project>.c***. For example, ***SOURCE*** instead of ***SOURCES*** or a missing **=** after the colon will fail.
6. An error related to waiting for an **unfinished process** usually means you will need to open the Window's Task Manager and kill processes with the following names:

   arm-none-eabi-gdb.exe

openocd-all-brcm-libfdti.exe

2.4 Peripherals

2.4.1 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()*.

2.4.2 PWM

The PWM has an API function to choose which PWM to use, 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 for the CYW43907 may result in an unexpected frequency.

2.4.3 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. There are different message types that can be enabled or disabled in the file "include/wiced_defaults.h". We will use one called "WPRINT_APP_INFO" which is meant for printing application information and is enabled by default. This is a macro that uses standard *printf()* formatting. 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 double set of parentheses is <u>required</u> due to the way the macro is defined.

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

There are other message macros that are not enabled by default in wiced_defaults.h such as WPRINT_APP_DEBUG and WPRINT_APP_ERROR. These macros are used in some of the examples so you can enable them in cases where those messages are useful.

2.4.4 UART

In addition to the USB-UART debug print functions, the device can also send/receive 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.

```
199 /**
200  * UART configuration
201  */
202 typedef struct
203 {
204     uint32_t                    baud_rate;
205     platform_uart_data_width_t  data_width;
206     platform_uart_parity_t      parity;
207     platform_uart_stop_bits_t   stop_bits;
208     platform_uart_flow_control_t flow_control;
209 } 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 */
```

Once you have a ring buffer setup, you can read from the UART using the function *wiced_uart_receive_bytes()*. It will read the specified number of bytes from the ring buffer for the given UART interface and will put them into a buffer that you provide. The function will return WICED_SUCCESS if a byte was received so you can tell if a byte was available in the buffer or not.

2.4.5 I2C

The device contains two I2C masters called WICED_I2C_1 and WICED_I2C_2. The OLED display and the PSoC on the shield connect to WICED_I2C_2. **The WICED_I2C_2 interface does not support clock stretching. Therefore, we will use it in standard speed mod`e (100 kHz) in our exercises to prevent any possible clock stretching from the peripherals that we are using.**

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 2 to connect to a slave at address 0x08 with a speed of 100 kHz (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 wiced_i2c_read and a dedicated write function called wiced_i2c_write. There is also a function called wiced_i2c_transfer which can do a read, a write, or both. We will focus on the separate functions here,

but feel free to look at the transfer function in the documentation and experiment with it. Some kits may only support one of the two methods.

wiced_i2c_read and wiced_i2c_write

The read/write functions (wiced_i2c_read and wiced_i2c_write) require a pointer to the device structure that was set up previously, a set of flags, a pointer to the buffer to read/write, and the number of bytes to read/write.

The possible flags are:

WICED_I2C_START_FLAG
WICED_I2C_STOP_FLAG
WICED_I2C_REPEATED_START_FLAG

Typically, for a complete read or write transaction, use the Start and Stop flags with an OR:

WICED_ I2C_START_FLAG | WICED_I2C_STOP_FLAG

For the buffer, 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".

wiced_i2c_transfer

If you decide to use the wiced_i2c_transfer function instead of the separate read/write functions, you first 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. 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.

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.

Once the message is set up, use *wiced_i2c_transfer()* to send or receive the message.

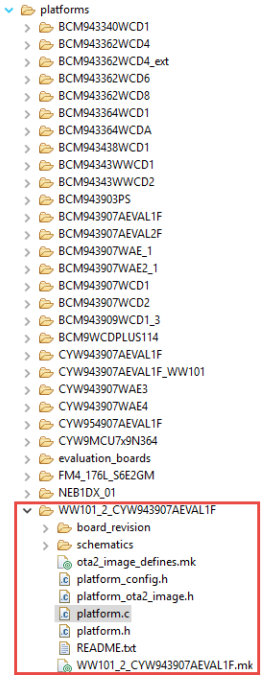
wiced_i2c_probe_device

You can also use *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. You must still initialize the device with *wiced_i2c_init()* before using *wiced_i2c_probe_device()*.

2.5 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 platform files folder is in the class material "WICED101 Files" folder.
    b. In addition to copying the platform files, open the file *apps/waf/tiny_bootloader/tiny_bootloader.mk* and add your platform name 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 platform folder (e.g. WW101_2_CYW943907AEVAL1F) and files. If you do not see them, ask for help – don't go forward until the platform is properly installed.



Questions to answer:

The table at the top of platform.h says that WICED_LED1 connects to WICED_GPIO_12, Arduino header D5, and WICED_PWM_3. Explain how this mapping was determined. You will need to refer to platform.h, platform.c and the schematic for the base board.

Exercise - 2.2 (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_LED1, OUTPUT_PUSH_PULL); */

    while ( 1 )
    {
            /* Add Code to Blink WICED_LED1 here */
    }
}
```

02_blinkled.mk:

```
NAME := Apps_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_LED1 low
    b. Wait 250ms
    c. Drive WICED_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_LED1 get assigned to the correct pin for this kit?

In what file and on what line is the pin connected to WICED_LED1 set as an output?

Exercise - 2.3 (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.
    a. Hint: Remember that you need to use double parenthesis to get this to work.
    b. Hint: Remember to use \n to create a new line so that information is printed each time the LED changes.
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".
    b. Hint: For MacOS, you may want to use the "screen" command as a terminal window.
        i. Look for a USB serial device in /dev/tty.*
        ii. Use the command:

            screen /dev/tty.<your_device> 115200

Exercise - 2.4 (GPIO) Read the State of a Mechanical 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_BUTTON1). Turn on WICED_LED1 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 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.
        vi. Hint: For the argument to pass to the interrupt from the *wiced_gpio_irq_en*able function call, use NULL.
3. Create the interrupt service routine (ISR) so that it toggles the state of the LED each time the button is pressed.
    a. Hint: The argument list to the ISR must be (void* arg). For example, your ISR should look something like this:

```
button_isr(void* arg) {
        <your code here>
}
```

    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.

Exercise - 2.6 (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 PSoC on the shield board. The PSoC AFE 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 (100 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 0 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.

Exercise - 2.7 (I2C READ) Read PSoC 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).

Exercise - 2.8 (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 2 – one for the PSoC, and one for the OLED.

Exercise - 2.9 (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_LED1 on the shield board instead of using the GPIO functions.
    a. Hint: The LED is connected to WICED_GPIO_12 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_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.

Exercise - 2.10 (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.

Exercise - 2.11 (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.

2.6 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.

2.7 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.