# 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, ADC, UART, and I2C) to interface with the shield including the LEDs, Switches, Thermistors, Analog Co-Processor (via I2C) and the KitProg UART. 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, the kit we are using is called “BCM94343W\_AVN”. Each board is supported by a folder in the SDK Workspace folder. The folder for our kit is not installed by default in the SDK so we need to copy the platform folder into the SDK Workspace. The folder for this board is named “BCM94343W\_AVN” 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 BCM94343W\_AVN 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 peripherals. For example, the kit board contains two LEDs and a button. These are identified in platform.h using the names WICED\_LED1, WICED\_LED2, and WICED\_BUTTON1.



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 pins LEDs are initialized as outputs and the button as an input with a resistive pullup for the button.

In platform.h you will also find a list of all of the valid peripherals. For example, the 4 ADC channels for this kit are:



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



Note that the ADC names must be used in the ADC API function calls. That is, you must use *WICED\_ADC\_1* to use ADC channel 1. You cannot use *WICED\_GPIO\_2* in the ADC 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.

### 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, ADC, 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. 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:



### 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.

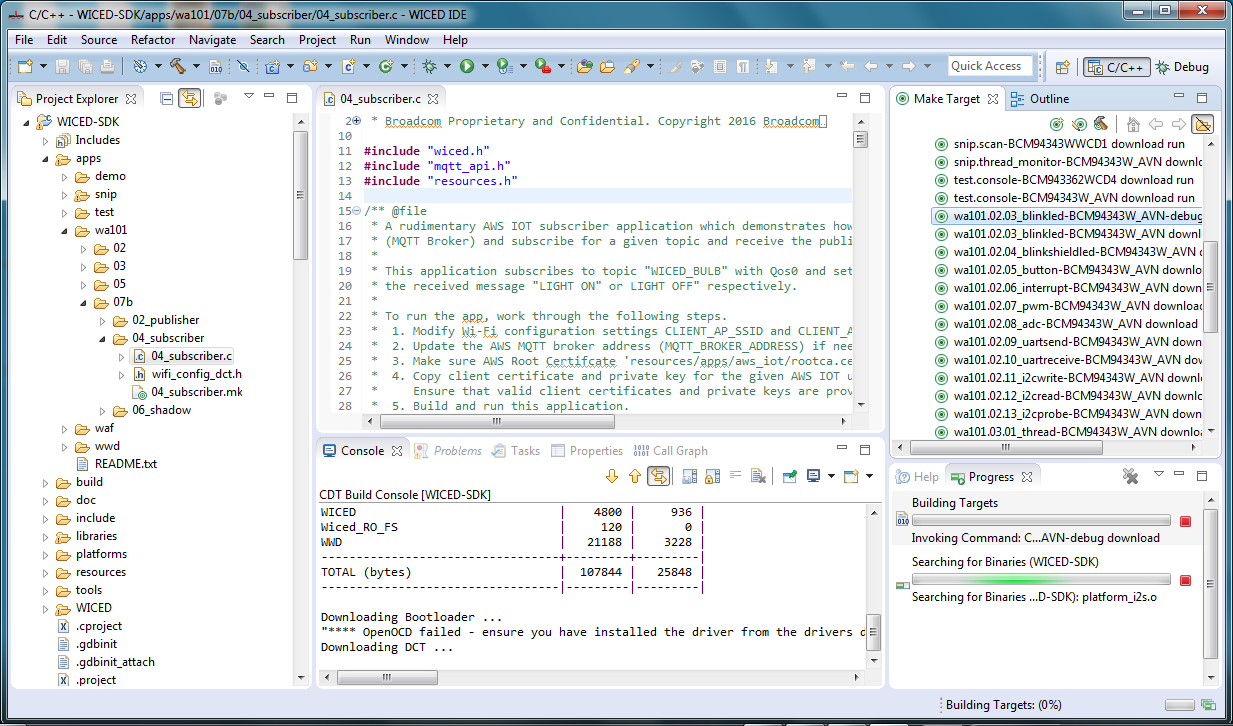
For example, if we create a folder called “ww101” for our class projects and a subfolder called “02” for the chapter 2 projects, and call the first project “03\_blinkled”, the build target for our board would be:

*wa101.02.01\_blinkled-BCM94343W\_AVN 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 the BCM94343W\_AVN kit, 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 receive a kit that has custom radio firmware which prevents it from working as expected. 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.

**Note that the initialization function will not start the PWM so you must use the start function after initializing it the first time.**

Entering a value for the frequency lower than that ~375Hz result in an unexpected frequency.

#### ADC

The ADC has an initialization API function to choose the channel (i.e. pin), and set the sampling cycle. It also has functions to take a single sample or to take a series of samples. The samples are 16 bit unsigned values. See the API documentation for details.

#### 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.**

#### 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 BCM94343W\_AVN kit, there is a second UART (called WICED\_UART\_2) connected to Arduino pins D8 and D9.

There are API functions for UART initialization, transmission, 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.



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:



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.

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).



Before sending data, 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.

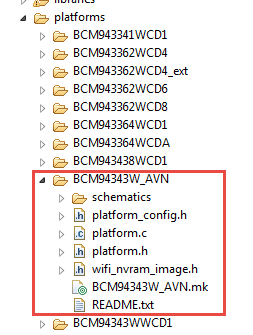
Once the structure is setup, use the function *wiced\_i2c\_transfer()* to send or receive the message.

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()*.

## Exercises

### 01 (PLATFORM) Install BCM94343W\_AVN into the platforms directory

1. Use what you learned in the fundamentals to install the files for the BCM94343W\_AVN kit into your SDK Workspace.
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 BCM94343\_AVN folder and files. If you do not see them, ask for help – don’t go forward until the platform is properly installed.



Questions to answer:

You will need to look in “platform.h”, “platform.c”, and the kit schematic to answer the following questions.

Which WICED GPIO pin is WICED\_PWM\_1 connected to?

Which module pin is WICED\_PWM\_1 connected to?

Which Arduino header pin is WICED\_PWM\_1 connected to?

### 02 (PROJECT) Setup a new project from a template

1. Create a folder inside the SDK Workspace *43xxx\_Wi-Fi/apps* folder called “ww101” and a sub-folder called “02”.
2. Copy the project called “02\_template” from *WW-101 Files/projects/WICED/ww101key/02* into *ww101/02*.
3. Rename the template project to “03\_blinkled”.
   1. Hint: Remember to change the names of the C file and makefile to match the project name.
4. Modify the makefile as necessary.
5. Create a make target for your new project.
   1. 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 “).

The remaining projects involve using either the kit or the kit with the shield attached. For those that require the shield, it is indicated in the project title. For the other projects, you can remove the shield board if you want to have easier access to the base board (i.e. button, ambient light sensor, LEDs).

The shield is designed to be powered from the base board. It is not necessary to connect a USB mini-B cable to the shield board unless you want to reprogram the PSoC 4 on the shield. In fact, the base board cannot be programmed if the shield is connected to a USB mini-B cable.

### 03 (GPIO) Blink an LED

1. Modify 03\_blinkled.c to blink LED 1 every 250ms.
   1. Hint: See the API documentation for the GPIO functions use to drive the LED high and low.
   2. Hint: Use the *wiced\_rtos\_delay\_milliseconds()* function for the delay.
2. Program your project to the board.
   1. 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 the LED set as an output?

### 04 (Advanced) (GPIO) Toggle a Pin that isn’t pre-initialized by the platform files (shield required)

1. Copy your project from 03\_blinkled to 04\_blinkshieldled. Modify the makefile as needed and create a make target.
   1. 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. Connect the analog co-processor shield to the kit.
   1. Hint: The USB connectors on the boards should both be on the same side.
   2. Hint: The USB connector to the analog co-processor shield is not needed for any of these exercises. The shield board is powered from the base board. The baseboard cannot be programmed if the USB connector to the analog co-processor shield is plugged in.
3. Modify the project to blink the Green LED on the shield. This is connected to Arduino pin A1 which connects to SDK pin WICED\_GPIO\_3.
   1. Hint: Don’t forget to initialize the IO as an output. See the API documentation for GPIOs for the API function required.
4. Program your project to the board.

### 05 (GPIO) Read the state of a button

1. The shield is not required for this exercise – you can remove it for now.
2. Copy the 03\_blinkled project to 05\_button, update the makefile, and create a make target.
3. In the C file, check the state of the kit’s button input (use WICED\_BUTTON1). Turn an LED on if the button is pressed and turn it off if the button is not pressed.
4. Program your project to the board.

### 06 (GPIO) Use interrupt to toggle the state of an LED

1. Copy the 05\_button project to 06\_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.
   1. Hint: See the documentation for *wiced\_gpio\_input\_irq\_enable().*
   2. Hint: In your C code do the following:
      1. Type *wiced\_gpio\_input\_irq\_enable()* in your code.
      2. Highlight *wiced\_gpio\_input\_irq\_enable()*, right click on it, and select “Open Declaration”. This will show the required parameters for the function.
      3. Highlight *wiced\_gpio\_irq\_trigger\_t*, right click on it, and select “Open Declaration”.
      4. Highlight *platform\_gpio\_irq\_trigger\_t*, right click on it, and select “Open Declaration”.
      5. Identify the correct value to use for a falling edge interrupt.
   3. Hint: For the argument to pass to the interrupt handler, use NULL.
   4. 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.
4. Hint: You can use a static boolean variable type in the ISR to remember the LED state:
   * 1. *static wiced\_bool\_t led1 = WICED\_FALSE;*
5. Program your project to the board.

### 07 (Advanced) (PWM) LED brightness (shield required)

1. Copy the 04\_blinkshieldled project to 07\_pwm, update the makefile, and create a make target.
2. In the C file, configure a PWM to drive the Green LED on the shield board instead of using the GPIO functions.
   1. Hint: The Green LED is connected to WICED\_GPIO\_3 so you need to find out which PWM is connected to that pin (look in the platform files).
   2. Hint: Don’t forget to start the PWM after configuring it.
3. Change the duty cycle of the PWM in the main loop so that the LED gradually changes intensity.
   1. Hint: use a delay so that the intensity goes from 0% to 100% in one second.
4. The shield is required for this project. Connect it to the kit if it isn’t already. Program your project to the board.

### 08 (ADC/Debug Print) Read ambient light sensor and print value on the PC

1. Create a new project called 08\_adc (or copy the 02\_template project). Modify the makefile and create a make target.
2. In the C file:
   1. Initialize the ADC to read the GPIO that has the ambient light sensor on it.
      1. Hint: look in the “platform.h” file for a #define for the light sensor.
   2. In the main loop, read a value from the ADC and print it to the screen using the WPRINT\_APP\_INFO macro.
   3. Wait a while (e.g. 250ms) before getting the next ADC sample.
3. Program your project to the board.
4. Open a terminal window with a baud rate of 115200 and view the ambient light sensor readings.
   1. Hint: The kit will show up in the device manager under “Ports (COM & LPT)” as *“WICED USB Serial Port”*.
   2. Hint: The light sensor is next to Arduino header pin D0. The value will be smaller for more light hitting the sensor. Use your cellphone flashlight as a light source to see a large change.
5. Reset the kit to see the startup messages that are displayed by the WICED firmware.

### 09 (Advanced) (UART) Write a value using the standard UART functions

1. Copy the 06\_interrupt project to 09\_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.
   1. Hint: Disable the STDIO\_UART in the make file by adding the line:
      1. GLOBAL\_DEFINES := WICED\_DISABLE\_STDIO
   2. 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.
   3. 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.
   4. 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.
   1. 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.

### 10 (Advanced) (UART) Read a value using the standard UART functions

1. Copy 09\_uartsend to 10\_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.
   1. Hint: you will need to setup a ring buffer to receive the UART characters.
   2. 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.
   1. 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.

### 11 (Advanced) (I2C WRITE) Change LED state on shield (shield required)

1. Copy 06\_interrupt to 11\_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 red and blue LEDs 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:
   1. Connected to Arduino pins D14 and D15 (WICED\_I2C\_1)
   2. 7-bit address = 0x08
   3. Standard Speed (100kHz)
   4. EZI2C register access
      1. The first byte written is the register offset.
      2. All reads start at the previous write offset.
   5. The register map is as follows:

|  |  |  |
| --- | --- | --- |
| Offset | Description | Details |
| 00 | Red LED | 0 = OFF, non-zero = ON |
| 01 | Blue LED | 0 = OFF, non-zero = ON |
| 02 | Temperature | LSB of temperature \* 100 |
| 03 | Temperature | MSB of temperature \* 100 |
| 04 | Humidity | LSB of humidity \* 10 |
| 05 | Humidity | LSB of humidity \* 10 |

* 1. Hint: To turn ON the Red LED and turn OFF the Blue LED, you need to send three bytes: 0x00 (offset), 0x01 (Red LED ON), 0x00 (Blue LED OFF).
  2. 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.
  3. 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!

### 12 (Advanced) (I2C READ) Read sensor values (shield required)

1. Copy 11\_i2cwrite to 12\_i2cread. Update the makefile and create a make target.
2. Update the code so that every time the button is pressed the temperature and humidity data are read from the I2C slave. Print the values to the terminal using WPRINT\_APP\_INFO.
   1. Hint: Remember to set the offset to 02 to read the temperature. You can do this just once and it will stay set for all future reads. With an offset of 02 you can read 4 bytes to get the temperature and humidity values.

### 13 (Advanced) (I2C PROBE) Probe for I2C devices (shield required)

1. Copy 11\_i2cwrite to 13\_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.
   1. 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?
   1. Hint: There should be 3 – one for the PSoC analog co-processor, and two for the on-board FRAM.

### 14 (Advanced) (PLATFORM) Make/modify platform files for the shield (shield required)

1. We have not created a complete custom board but we are using a PSoC Analog Co-processor shield connected to the base board. Let’s make a platform for this hardware configuration to make our projects more readable.
2. First, in the platforms folder copy the “BCM94343W\_AVN” folder to a new folder called “WA101”. You can do this right from inside WICED Studio.
3. In the new folder, rename the make file WA101.mk and delete the schematics directory.
4. Update the README.txt file.
5. In “platform.c” and “platform.h” add the following new items:
   1. SHIELD\_I2C -> This will point to WICED\_I2C\_1
   2. SHIELD\_LED\_GREEN -> This will point to WICED\_GPIO\_3
      1. Hint: Also update the initialization function so that WICED\_GPIO\_3 is initialized as a push-pull output and is driven low just like LED1 and LED2.
6. Since our baseboard and CPU have not changed, we don’t need to make any other changes in “platform.h”, “platform.c”, “platform\_config.h”, or “Wi-Fi\_nvram\_image.h”. If you were to make a complete new board, potentially with a different Wi-Fi device, those files would need to be updated as well.
7. Once you are done, go back and modify projects 04 and 11 to use the new definitions.
   1. Hint: you will need to change the make targets so that they use the new platform name.
   2. Hint: on project 04, you no longer need to initialize the GPIO for the Green LED.

## 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.