# 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 development kit

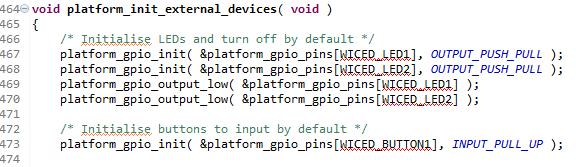
The WICED SDK has files that make it easier to work with the peripherals on a given kit. In our case, the kit we are using is called “BCM94343W\_AVN”. It is not installed by default in the SDK so we need to copy the platform folder into the SDK. The folder for this board is provided with the class materials. Copy it into <install path>/WICED-SDK-3.7.0-3/WICED-SDK/platforms/. The contents of BCM94343W\_AVN is:



Two key files here are platform.c and platform.h. These contain #define, type definitions, and constants that set up the various kit peripherals. For example, in platform.h you will see the two LEDs and button can be accessed using the names WICED\_LED1, WICED\_LED2, and WICED\_BUTTON1.



In addition, the pins are initialized properly in platform.c (outputs for the LEDs and an input with a resistive pullup for the button):



The platform.h and platform.c files also contain definitions for the various peripherals such as PWMs and ADCs. In platform.h you will 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:

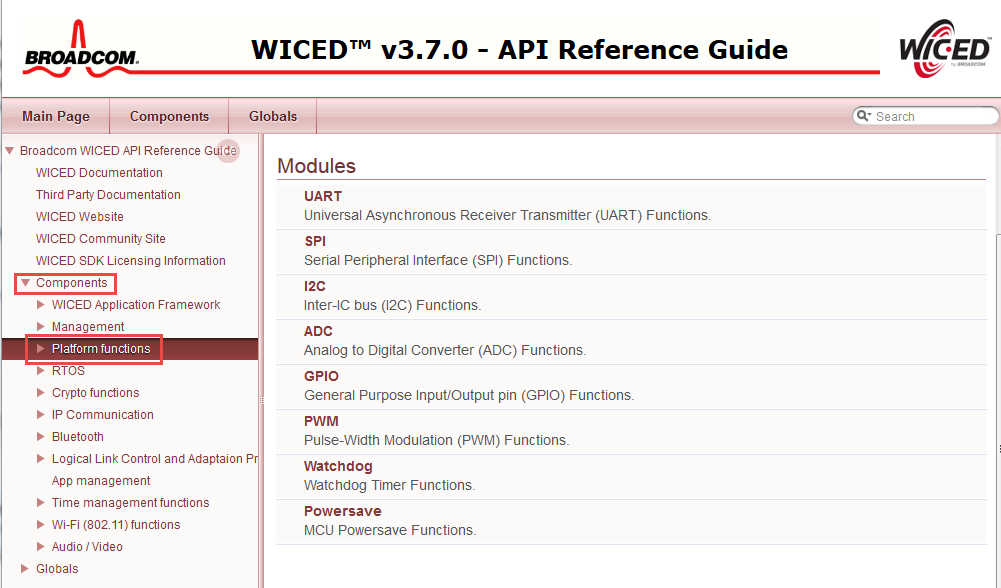


Note that the ADC names must be used in the ADC API function calls. That is, you must use *WICED\_ADC\_1* to use ADC channel 1. You cannot use *WICED\_GPIO\_2* in the ADC API function calls.

If you develop your own hardware, then it is best to add a new folder to the SDK 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 “doc” directory inside the SDK. 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 from inside the SDK 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, in the code you can highlight the parameter, right click, and select “Open Declaration”. 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 project

#### Directory Structure

A WICED 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 where <project> the name of the project

A make file called <project>.mk inside the project folder

The make file 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. It may also contain make file macros to provide access to libraries and other resources such as images, web pages, etc.

#### 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> and (optionally) <folder2> are the path to the project from the apps directory. Many examples provided in the SDK have only <folder1> but we will create an extra level to keep our class projects separated by chapter.

<project> is the name of the project

<platform> is the name of the kit.

For example, if we create a folder called “wa101” for our class projects and a subfolder called “01” for the chapter 1 projects, and call the first project 01\_blinkled, the build target for our board would be:

*wa101.01.01\_blinkled-BCM94343W\_AVN download 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*.

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

If you want to suppress the standard debug printing on STDIO\_UART, add the following line to the make file for the project:

GLOBAL\_DEFINES := WICED\_DISABLE\_STDIO

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

The 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 the SDK 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 structures. One note: 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 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 WICED SDK.

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 apps folder called wa101 and a sub-folder called 01.
2. Copy the project called “02\_template” from the wa101key/01 folder into wa101/01.
3. Rename the template project to “03\_blinkled”.
   1. Hint: Remember to change the names of the C file and make file to match the project name.
4. Modify the make file 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.

### 03 (GPIO) Blink an LED

1. Modify 03\_blinkled.c to blink LED 1 every 250msec.
   1. Hint: Use a Boolean variable to remember the state of the LED:
      1. *wiced\_bool\_t led1 = WICED\_FALSE;*
   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 the SDK 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 (GPIO) Toggle a Pin that isn’t pre-initialized by the Platform files

1. Copy your project from 03\_blinkled to 04\_blinkshieldled. Modify the make file 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 the WICED SDK 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.
4. Program your project to the board.

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

1. Copy the 03\_blinkled project to 05\_button, update the make file, and create a make target.
2. 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.
3. Program your project to the board.

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

1. Copy the 05\_button project to 06\_interrupt, update the make file, 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: For the argument to pass to the interrupt handler, use NULL.
   3. Hint: For the 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.
   1. Hint: if you use a variable to remember the LED state inside the ISR, make sure it is static so that it is not reset every time you enter the ISR.
4. Program your project to the board.

### 07 (PWM) LED brightness

1. Copy the 04\_blinkshieldled project to 07\_pwm, update the make file, 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: 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: set the duty cycle change and delay such that the intensity goes from 0% to 100% in one second.
4. 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 make file 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 function.
   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 (UART) Write a value using the standard UART functions

1. Copy the 06\_interrupt project to 09\_uartsend. Modify the make file 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: Use a baud rate of 115200, data with of 8, no parity, 1 stop bit, and no flow control.
   2. 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.
   3. 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 115200.
   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 (UART) Read a value using the standard UART functions

1. Copy 09\_uartsend to 10\_uartreceive. Update the make file 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: 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 115200.
   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 (I2C WRITE) Change LED state on shield

1. Copy 05\_interrupt to 11\_i2cwrite. Update the make file 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 (I2C READ) Read sensor values

1. Copy 11\_i2cwrite to 12\_i2cread. Update the make file 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 (I2C PROBE) Probe for I2C devices

1. Copy 11\_i2cwrite to 13\_i2cprobe. Update the make file 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 (PLATFORM) Make/modify platform files for the shield

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 the WICED SDK.
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/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 wifi\_nvram\_image.h. If you were to make a complete new board potentially with a different WiFi 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.