BeagleBone Tutorial: GPIO, Interrupt, Analog and PWM

Table of Contents

Introduction

In this tutorial, users will:

1. Be introduced to MRAA.
2. Write C code to enable access GPIO pins.
3. Write C code to handle a SIGINT signal.
4. Write C code to handle GPIO interrupts.
5. Write C code to sample analog data.
6. Write C code to generate PWM signals.

Prerequisite Tutorials

Users should ensure they are familiar with the documents listed below before proceeding.

1. BeagleBone Tutorial – Introduction
2. BeagleBone Tutorial – Introduction to Linux
3. BeagleBone Tutorial – Introduction to Vim

List of Required Materials and Equipment

1. 1x IoT Developer Prototyping Kit including
    a. a BeagleBone Green Wireless
    b. a Grove Base Cape
    c. Grove sensors
2. 1x USB 2.0 A-Male to Micro B Cable (micro USB cable)
3. 1x Personal Computer

Introduction to the MRAA Library

MRAA is a C library that provides an Application Programming Interface (API) to establish a method of communication between compute modules (such as the BeagleBone) and other peripheral devices (such as sensors).

The BeagleBone and the MRAA library support a number of protocols including: Universal Asynchronous Receiver/Transmitter (UART), Inter-integrated Circuit (I2C), Pulse Width Modulation (PWM) and General Purpose Input/Output (GPIO) programming. Follow the below instructions to display version information about MRAA to standard output.

Note: SPI interface is not supported as of 7/5/2017.

1. Access the shell on the BeagleBone.

2. Issue the commands shown below.

   $ mkdir ~/tutorial5_examples
   $ cd ~/tutorial5_examples
   $ vi check_mraa_version.c

   Enter the C code from Figure 1 into the Vim editor.

```c
#include <stdio.h>
#include <mraa.h>

int main()
{
        printf("MRAA version: %s\n", mraa_get_version());
        return 0;
}
```

Figure 1: C code source file check_mraa_version.c

3. Compile the source code file into an executable binary file by issuing the following command.

   $ gcc -lmraa -o check_mraa_version check_mraa_version.c

   The **-lmraa** flag is required in order to access the functions provided by MRAA.

4. Execute the binary file by issuing the command below.

   $ ./check_mraa_version

GPIO

The BeagleBone has pins dedicated to each to the protocols mentioned in the introduction **(PWM, UART, I2C, etc)**. However, there are cases where a sensor or peripheral device will not use a standard protocol such as I2C or SPI. In order to interface with these devices, the BeagleBone provides access to **General Purpose Input/Output** (**GPIO**) pins.

The main use for a GPIO pin is to wake a processor up from sleep mode. Other applications include changing the logical levels of a device, such as an LED or a buzzer. In summary, GPIO pins can be used for either writing data to, or reading data from a device.

Demo – Buzzer Control

In the demonstration below, you will change the logic level of a GPIO pin to write digital value to a buzzer to turn it on and off. A developer can utilize the **mraa_gpio_write** function to change the logic level of a GPIO pin.

To control a buzzer, follow the steps outlined below.

1. Shut the BeagleBone down. In order to avoid any damages, hardware must be configured when the device is powered off.

2. Remove the USB cable from the BeagleBone to remove power.

3. Insert the Grove Cape into the BeagleBone. Using a Grove cable, connect a buzzer to GPIO_51 pin on the Grove Cape. Please be attentive to the hardware configuration as shown in the picture below. If the hardware is not configured correctly, the system will not perform as desired or may even get permanently damaged.

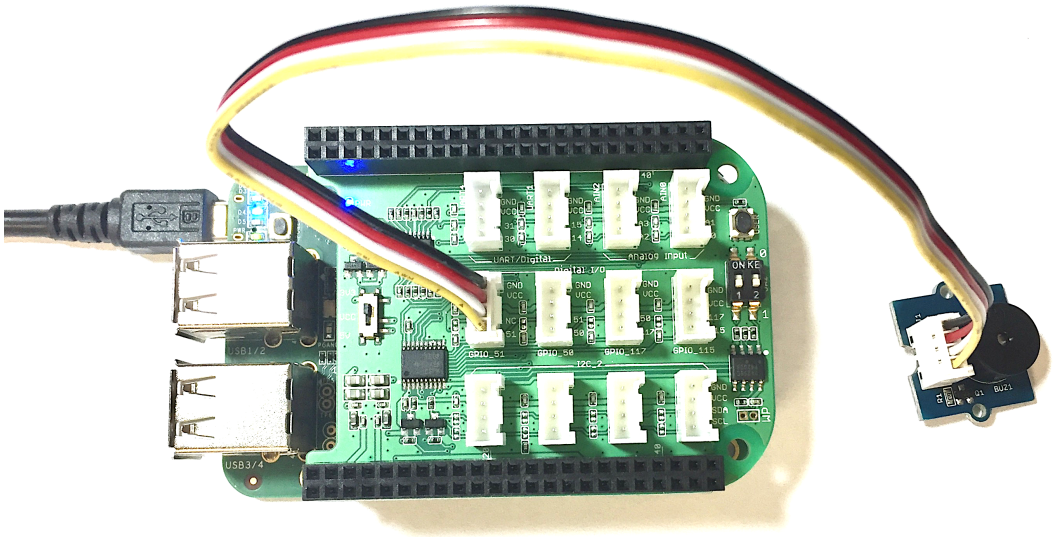

Figure 2: Buzzer control hardware configuration

4. Connect the USB cable to the BeagleBone and to your personal computer.
5. Access the shell on your BeagleBone, and issue the command shown below.

   $ cd ~/tutorial5_examples

6. Create the source code file by issuing the command shown below.

   $ vi buzzer.c

   Enter the C code from Figure 3 into the Vim editor.

```c
#include <signal.h>
#include <mraa/gpio.h>

// flag checked in while loop in main function.
// it is initialized to be 1, or logical true.
// it will be set to 0, or logical false on receipt of SIGINT
sig_atomic_t volatile run_flag = 1;

// this function will set the run_flag to logical false.
void do_when_interrupted(int sig)
{
        // if-statement will verify if the signal is the SIGINT signal before proceeding.
        if (sig == SIGINT)
                run_flag = 0;
}

int main()
{
        // declare buzzer as a mraa_gpio_context variable (GPIO).
        mraa_gpio_context buzzer;
        // initialize MRAA pin 62 (gpio_51) for buzzer.
        buzzer = mraa_gpio_init(62);
        // configure buzzer GPIO interface to be an output pin.
        mraa_gpio_dir(buzzer, MRAA_GPIO_OUT);

        // when SIGINT signal is received, call do_when_interrupted.
        signal(SIGINT, do_when_interrupted);

        // execute if run_flag is logical true (1).
        // break while loop if run_flag is logical false (0).
        while (run_flag) {
                // turn the buzzer on by setting the output to logical true.
                mraa_gpio_write(buzzer, 1);
                sleep(1);
                // turn the buzzer off by setting the output to logical false.
                mraa_gpio_write(buzzer, 0);
                sleep(1);
        }

        mraa_gpio_write(buzzer, 0);
        mraa_gpio_close(buzzer);

        return 0;
}
```

Figure 3: C code source file buzzer.c

7. Compile the code.

   $ gcc -lmraa -o buzzer buzzer.c

8. Execute the binary file.

   $ ./buzzer

   The buzzer will continuously sound every other second. If not, please check the code, hardware configuration, and reboot the system.

9. Issue the **Ctrl-C** keystroke to terminate.

You may have noticed that you used GPIO pin 51 and yet initialized pin 62 in your C code. 62 is the MRAA pin number for GPIO pin 51. The Grove Cape has four GPIO connectors and their MRAA pin mapping is shown in the table below.

GPIO	MRAA	GPIO	MRAA
GPIO_51	62	GPIO_117	71
GPIO_50	60	GPIO_115	73

For more information on MRAA pin mapping, please refer to https://www.seeed.cc/project_detail.html?id=1592.

GPIO Interrupt

In the previous section, the code contained a function named **signal**. This function is used to specify system behavior when a signal such as **SIGINT** is received. A user can generate the **SIGINT** signal by issuing the **Ctrl-C** keystroke while interacting with the Linux Operating System shell program. Consider the code from Figure 3. If the **SIGINT** signal is detected, the signal handler function **do_when_interrupted** will be called. Calling the **do_when_interrupted** function causes the variable **run_flag** to be set to **0**. Setting this variable to **0** will cause the function to exit as the condition in the **while(run_flag)** loop will be set to **logical false**.

Similar functionality can be implemented with a hardware component such as a button if a GPIO pin can be configured as an interrupt source. An interrupt handler function similar to the signal handler function **do_when_interrupted** can be implemented to handle generated.

Demo – GPIO Interrupt on Button Press

Follow the instruction below to use a button provided in the **IoT Developer Prototyping kit** to generate interrupts on a GPIO pin.

1. Shutdown the BeagleBone and remove it from power before any hardware configuration.

2. Connect a buzzer to GPIO_51 and a button to GPIO_50 as shown in Figure 4.

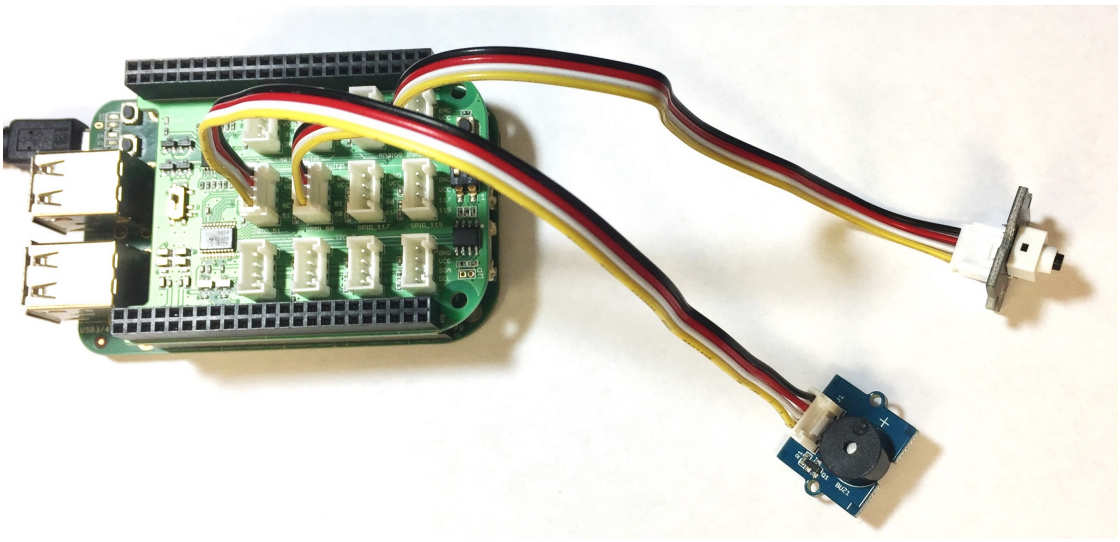

Figure 4: Hardware configuration to demonstrate generation of signals using a button press

3. Connect the USB cable and access the shell on the BeagleBone.

4. Navigate to ~/**tutorial5_examples** directory.

5. Create the source code file by issuing the command shown below.

   $ vi gpio_interrupt.c

6. Enter the C code from Figure 5 into the Vim editor.

```c
#include <signal.h>
#include <mraa/gpio.h>

// flag checked in while loop in main function.
// it is initialized to be 1, or logical true.
// it will be set to 0, or logical false on receipt of SIGINT
sig_atomic_t volatile run_flag = 1;

// this function will set the run_flag to logical false.
void do_when_interrupted()
{
        run_flag = 0;
}

int main()
{
        // declare buzzer and button as mraa_gpio_context variables (GPIO).
        mraa_gpio_context buzzer, button;
        // initialize MRAA pin 62 for buzzer and MRAA pin 60 for button.
        buzzer = mraa_gpio_init(62);
        button = mraa_gpio_init(60);

        // configure buzzer GPIO interface to be an output pin.
        mraa_gpio_dir(buzzer, MRAA_GPIO_OUT);
        // configure button GPIO interface to be an input pin.
        mraa_gpio_dir(button, MRAA_GPIO_IN);

        // when the button is pressed, call do_when_interrupted.
        mraa_gpio_isr(button, MRAA_GPIO_EDGE_RISING, &do_when_interrupted, NULL);

        // execute if run_flag is logical true (1).
        // break while loop if run_flag is logical false (0).
        while (run_flag) {
                // turn the buzzer on by setting the output to logical true.
                mraa_gpio_write(buzzer, 1);
                sleep(1);
                // turn the buzzer off by setting the output to logical false.
                mraa_gpio_write(buzzer, 0);
                sleep(1);
        }

        mraa_gpio_write(buzzer, 0);
        mraa_gpio_close(buzzer);
        mraa_gpio_close(button);

        return 0;
}
```

Figure 5: C code source file gpio_interrupt.c

7. **$ gcc -lmraa -o gpio_interrupt gpio_interrupt.c**

8. **$ ./gpio_interrupt**

9. Press the button. Notice that the **gpio_interrupt** process is terminated.

The interrupt on the GPIO is triggered by a signal edge. There are two kinds of signal edges: a rising edge and a falling edge. These edges are illustrated in the figure below.

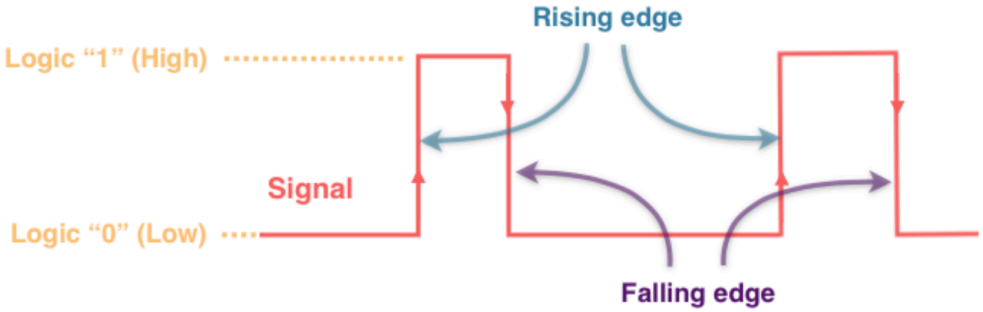

Figure 6: Signal edges

When the button is not pressed, the signal is low. As soon as the button is pressed, the signal "rises" to high. Such signal transition from low to high is called a "rising edge". The C code above has a GPIO interrupt setup for rising edges. The rising edge on the GPIO pin triggers the GPIO interrupt and the handler for the interrupt is called. The handler sets run_flag to 0; hence the program exits out of the while loop.

Follow the steps outlined below to modify the code such that the **SIGINT** signal is generated on a **falling edge** rather than a **rising edge**.

1. **$ vi gpio_interrupt.c**

2. The original mraa_gpio_isr function is called as follows.

   mraa_gpio_isr(button, **MRAA_GPIO_EDGE_RISING**,
   &do_when_interrupted, NULL);

   Modify the code such that the function is now called as shown below.

   mraa_gpio_isr(button, **MRAA_GPIO_EDGE_FALLING**,
   &do_when_interrupted, NULL);

3. **$ gcc -lmraa -o gpio_interrupt gpio_interrupt.c**

4. **$ ./gpio_interrupt**

5. Press and hold the button for a 5 seconds.

6. Release the button. Notice that the **gpio_interrupt** process terminates.

Analog Input

Analog vs Digital

To understand the difference between digital and analog signals, one must first understand the difference between discrete and continuous variables.

A continuous variable is a variable that can have **a value between ANY two other values**. In other words, a continuous variable can take on **infinitely many values**. For example, if the variable X is continuous, one value of X is 0.9 and the other is 1.0, X can have values of 0.95, 0.975, 0.9865, …, 1. Examples of continuous variables include: distance (object A can be 1m from object B, or 0.1m, or 0.01m, etc), and mass (object A can be 1kg, 1.1kg, 1.11kg, etc).

A discrete variable is a variable that is can **only** take **certain values**. In other words, a continuous variable can only take on a **finite set of values**. For example, if the variable Y is discrete, one value of Y is 0.9 and the other is 1.0, Y **cannot** be 0.95. It must either be 0.9, or 1. One example of a discrete variable is a letter, there is no definition for a letter halfway between A and B.

In the real world, most signals are continuous. However, these continuous values can be mapped to discrete values to make data easier to deal with. For example, if someone's height is 5'6.1325ft, it is often enough to say the person is 5'6ft tall. Similar conversions can be made to electronic components. For example, if a device outputs a current of 0.0124A, it could be appropriate to call it a 0.01A signal. This process is called **discretization** of a **continuous** signal.

Analog signals are signals that are continuous in the **y-axis** (**values**) they take. Digital signals are **discrete** in both the **x-axis** (the signals are discrete time signals) and the **y-axis** (the values at each time step can only be certain values). This is summarized in the table below.

Time (x-axis)	Value (y-axis)	Signal type
Continuous	Continuous	Continuous time analog
Continuous	Discrete	Piecewise-constant signals
Discrete	Continuous	Discrete time analog
Discrete	Discrete	Digital

The difference between analog and digital signals can also be examined in Figure 7.

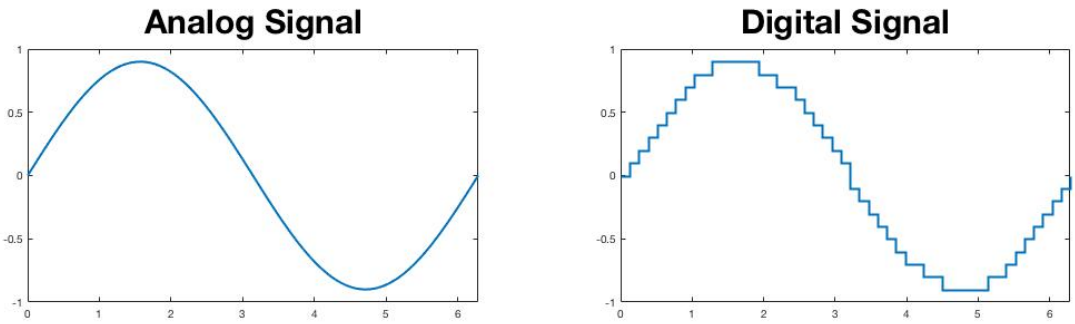

Figure 7: Representing sin(x) as a continuous time analog signal, and as a digital signal

Most signals in the real world are **analog signals**. However, computers operate using **binary**. Since binary is a discrete, **analog** data must be converted to **digital** before computers perform computations on the data. This is usually done with a hardware component called an **Analog to Digital Converter** (ADC). The BeagleBone's TI AM335x processor features a ADC.

Demo – Rotary Angle Sensor Data Acquisition

Follow the steps below to display analog data, read from the rotary sensor, to standard output.

1. Power down the BeagleBone, and remove all cables.

2. Connect a rotary angle sensor to AIN0, which is mapped to MRAA pin 1, as shown in the picture below. A rotary angle sensor is sometimes called a **potentiometer**.

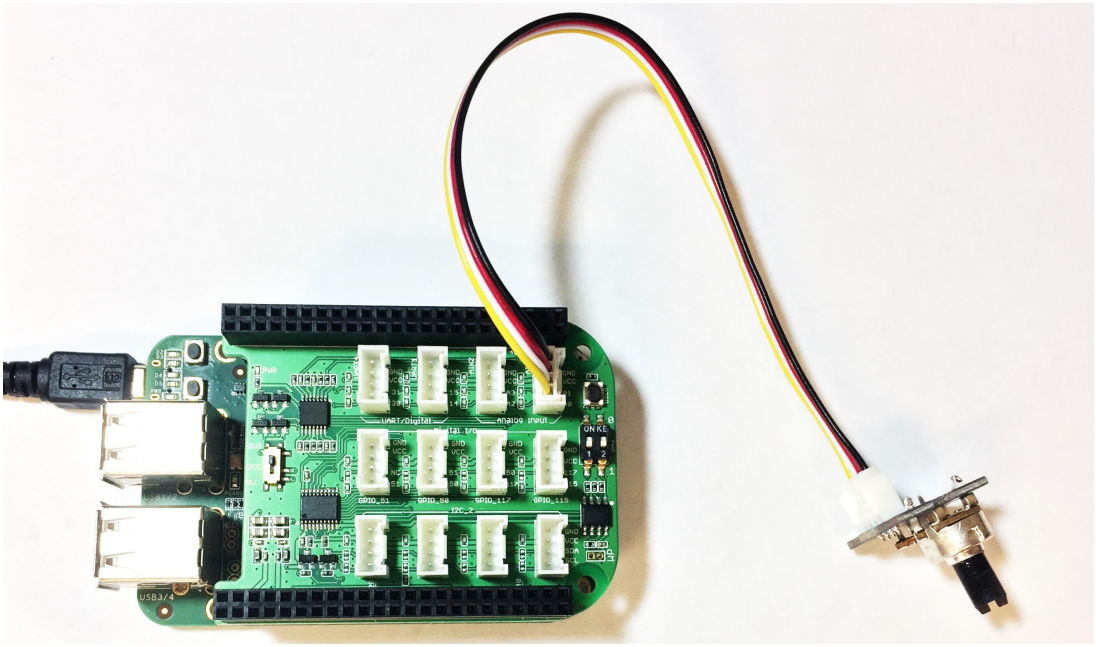

Figure 8: Hardware configuration to demonstrate how to read data from an analog sensor

Note: The Grove Cape has another analog connector, which is mapped to MRAA pin 3.

3. Connect the USB cable and access the shell on the BeagleBone.
4. Navigate to **~/tutorial5_examples** directory.
5. **$ vi rotary.c**
6. Enter the C code from Figure 9 into the Vim editor.

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <mraa/aio.h>

// flag checked in while loop in main function.
// it is initialized to be 1, or logical tru.
// it will be set to 0, or logical false on receipt of SIGINT
sig_atomic_t volatile run_flag = 1;

// this function will set the run_flag to logical false
void do_when_interrupted(int sig)
{
        if (sig == SIGINT)
                run_flag = 0;
}

int main()
{
        // variable to store the value returned from mraa_aio_read
        uint16_t value;
        // declare rotary as a mraa_aio_context variable (Analog Input/Output)
        mraa_aio_context rotary;
        rotary = mraa_aio_init(1);

        signal(SIGINT, do_when_interrupted);

        while (run_flag) {
                //mraa_aio_read will read the voltage provided by an analog device
                value = mraa_aio_read(rotary);
                printf("%d\n", value);
                usleep(100000);
        }
        mraa_aio_close(rotary);
        return 0;
}
```

Figure 9: C code source file rotary.c

7. **$ gcc -lmraa -o rotary rotary.c**
8. **$ ./rotary**
9. Slowly rotate the knob and observe the relative change in output.
10. Issue the **Ctrl-C** keystroke to terminate the **rotary** process.

PWM

Pulse-Width Modulation (PWM) is a **modulation scheme** that enables a **digital signal** to represent **analog data**. PWM has a variety of use cases including control of complex circuits, enabling custom brightness levels for Light Emitting Diodes (LED's) and controlling servos.

PWM is a clever scheme to enables digital circuits that can only output either logical "high" (e.g. +5V) or logical "low" (e.g. 0V) to produce an analog output somewhere between "high" and "low". The circuit produces this "in between" value by switching between "high" and "low" very quickly. For example, the BeagleBone is capable of producing a 400 MHz PWM signal. This means the BeagleBone is capable of generating a new pulse approximately every 2.5 nanoseconds. The signal can be averaged to deliver this "in between" value. This is typically achieved through some kind of low pass filtering circuitry.

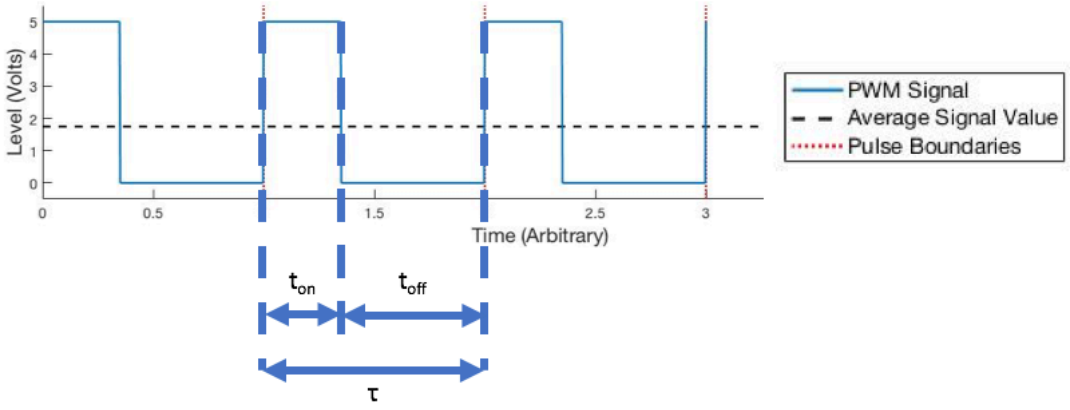

Figure 10: Key metrics of a PWM signal

From the waveform, there are three key values of interest:

1. t_{on} – the amount of time the circuit is outputting a logical level "high".
2. t_{off} – the amount of time the circuit is outputting a logical level "low".
3. τ – the period of the signal.

Any two of these values can be used to calculate the duty cycle of the PWM signal.

$$D = \frac{t_{on}}{\tau} = 1 - \frac{t_{off}}{\tau} = \frac{t_{on}}{t_{on} + t_{off}} \ [Unitless]$$

If the PWM signal **is a square wave as shown above**, the average value of the signal may be calculated from the below formula. It is important to note that the formula below **may not apply if the shape is different from** Figure 10.

$$\overline{y} = D \times (V_{max} - V_{min}) \ [V]$$

Figure 11 shows how adjusting the duty cycle changes the average value of the signal.

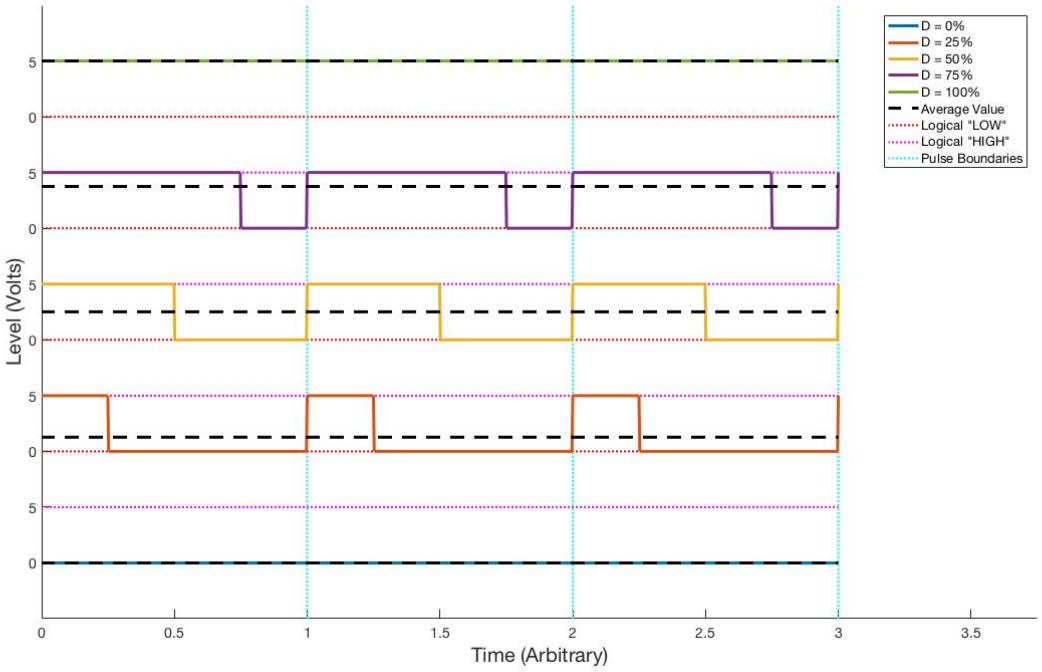

Figure 11: Visualization of how a PWM signal can be averaged to get an analog output from a digital circuit

The average value is the analog equivalent to the PWM square wave signal. Assume that the high voltage level is 5V and the low voltage level is 0V. The duty cycle of 100% indicates that the signal is constantly at 5V. If the duty cycle is 0%, the signal is at 0V. Things get interesting when the duty cycle is between 0% and 100%. Consider the duty cycle of 50%. The signal is 5V for half of the duration and 0V for the half. If the signal is properly filtered with a low pass filter, the resulting signal will be at 2.5V, which is the DC offset of the original PWM signal. Similarly, the duty cycle of 80% can be used to generate 4V signals.

Demo – Buzzer Volume Control

Follow the below steps in order generate a PWM signal to control the volume of a buzzer.

1. Power down the BeagleBone and remove all cables.
2. Connect a buzzer to GPIO_51 on the Grove Cape as we did in the buzzer control demo.

   Note: Only GPIO_51 and GPIO_50 on the Grove Cape can be used as PWM pins. The MRAA pin numbers are 62 and 60 respectively.
3. Connect the USB cable and access the shell on the BeagleBone.
4. Navigate to ~/**tutorial5_examples** directory.
5. **$ vi buzzer_volume.c.**

6. Enter the C code from Figure 12 into the Vim editor.

```c
#include <mraa/pwm.h>

int main()
{
    // declare pwm as a mraa_pwm_context variable (Pulse Width Modulation).
    mraa_pwm_context pwm;

    // initialize the PWM interface.
    pwm = mraa_pwm_init(62);

    // set the period of the PWM signal (microseconds).
    mraa_pwm_period_us(pwm,400);
    // enable PWM.
    mraa_pwm_enable(pwm, 1);

    // declare duty_cycle varialbe.
    int duty_cycle;

    // increase duty cycle over time.
    for(duty_cycle=0; duty_cycle < 30; duty_cycle++) {
            // generate appropriate PWM signal.
            mraa_pwm_write(pwm, (float)duty_cycle/100.0);
            usleep(100000);
    }

    // decrease duty cycle over time.
    for(duty_cycle=30; duty_cycle > 0; duty_cycle--) {
            // generate appropriate PWM signal.
            mraa_pwm_write(pwm, (float)duty_cycle/100.0);
            usleep(100000);
    }

    // diable PWM.
    mraa_pwm_enable(pwm, 0);
    mraa_pwm_close(pwm);

    return 0;
}
```

Figure 12: C code source file buzzer_volume.c

7. **$ gcc -lmraa -o buzzer_volume buzzer_volume.c**

8. **$ ./buzzer_volume**

   Observe that the buzzer volume increases and then decreases.