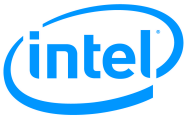
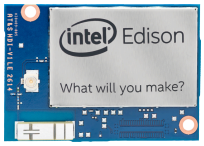
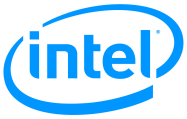
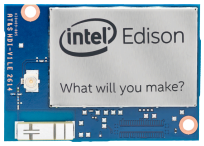


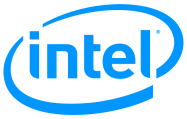


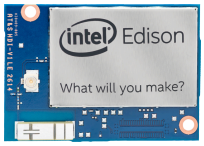

# Intel® Edison Tutorial: GPIO, Interrupts, Analog and I2C Interfaces






# Table of Contents






## Introduction

In this tutorial, users will:

1. Be introduced to libmraa.
2. Write C code to enable access GPIO pins.
3. Write C code to handle a SIGINT signal.
4. Write C code to generate the SIGINT signal when a user presses a button attached to the Grove Base Shield.
5. Write C code to sample data from a rotary analog sensor.
6. Write C code to control the color of the Grove-LCD RGB Backlight v4.0 display.

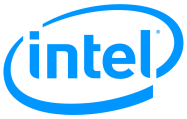
## Prerequisite Tutorials

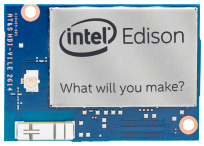
Users should ensure they are familiar with the documents listed below before proceeding.

1. Intel Edison Tutorial – Introduction, Linux Operating System Shell Access
2. Intel Edison Tutorial – Introduction to Linux
3. Intel Edison Tutorial – Introduction to Vim

## List of Required Materials and Equipment

1. 1x Intel Edison Kit
2. 2x USB 2.0 A-Male to Micro B Cable (micro USB cable)
3. 1x powered USB hub **OR** an external power supply
4. 1x Grove – Starter Kit for Arduino
5. 1x Personal Computer






## Introduction to the mraa Library

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

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

1. Access the shell on the Intel Edison. For more information, refer to the document labelled ***Intel Edison Tutorial – Introduction, Shell Access and SFTP***.
2. Ensure that the Intel Edison has the Vim editor installed by following the instructions in the document labelled ***Intel Edison Tutorial – Introduction to Vim***.
3. Issue the commands shown below.

   **$ mkdir ~/tutorial4_examples**
   **$ cd ~/tutorial4_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;
}
```

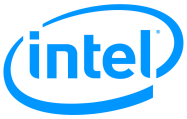
Figure 1: C code source file check_mraa_version.c

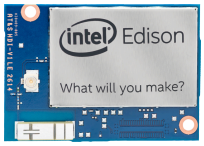
4. Compile the source code file into an executable binary file by issuing the following command.

   **$ gcc -lmraa -o check_mraa_version check_mraa_version.c**

   The **-lmraa** flag is required in order to access the functions provided by libmraa.
5. Execute the binary file by issuing the command below.

   **$ ./check_mraa_version**






## GPIO – Blink LED

The Intel Edison has pins dedicated to each to the protocols mentioned in the introduction **(PWM, SPI, 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 Intel Edison 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. In summary, GPIO pins can be used for either writing data to, or reading data from a device.

A developer can utilize the **mraa_gpio_write** function to change the logic level of a GPIO pin.

To make an LED blink, follow the steps outlined below.

1. Shut the Intel Edison down.
2. Remove all power and USB cables from the Intel Edison.
3. A Light Emitting Diode (**LED**) is an electrical component that emits light when a suitable voltage is applied across its terminals. Driving excessive current through an LED can cause it to break down. As such, the Grove – Starter Kit for Arduino contains an LED socket designed to prevent LED damage.

   Attach the LED to the LED socket. The LED has two legs. The shorter leg is the cathode (negative) and the longer leg is the anode (positive).

   Connect the longer leg to the socket pin with "+" sign. The orientation of the legs matter, so ensure that the hardware is configured correctly before proceeding.

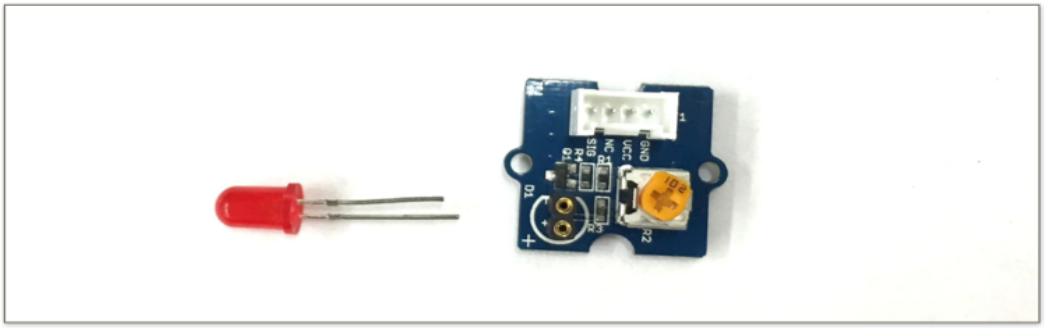



Figure 2: Light Emitting Diode (LED) and LED socket

4. Insert the Grove Base Shield into the Edison board for Arduino and connect other components as shown in the pictures below. Please be attentive to the below diagram. If the component is connected to an incorrect interface, the system will not perform as desired. For example, if you connect the LED to D4 **by mistake** instead of D3, the LED **will not emit light** if you do not correct for this difference in ports in your C-code.
5. Connect the USB and power cables to the Intel Edison.

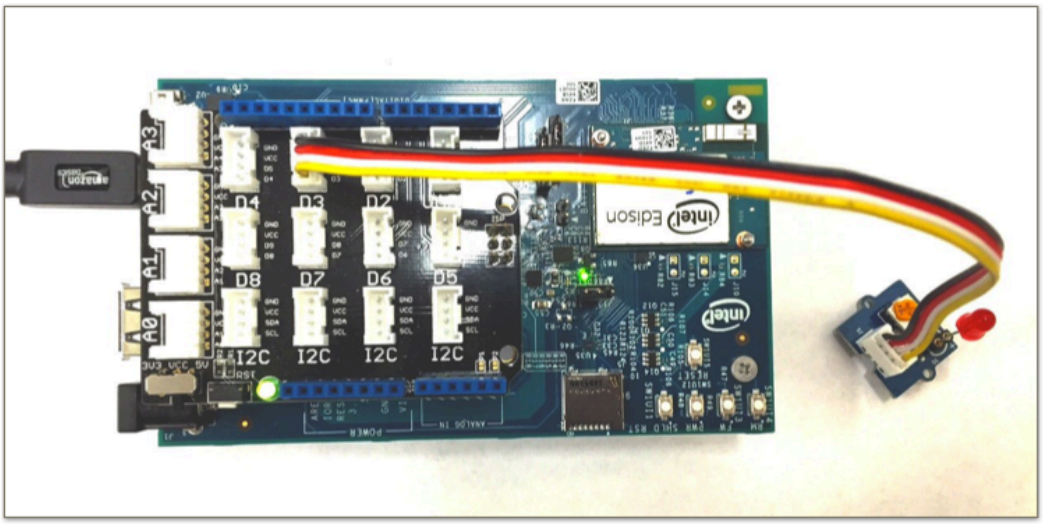



Figure 3: Correct hardware configuration to blink an LED

6. Ensure the Grove shield is operating at 5V by adjusting the switch.

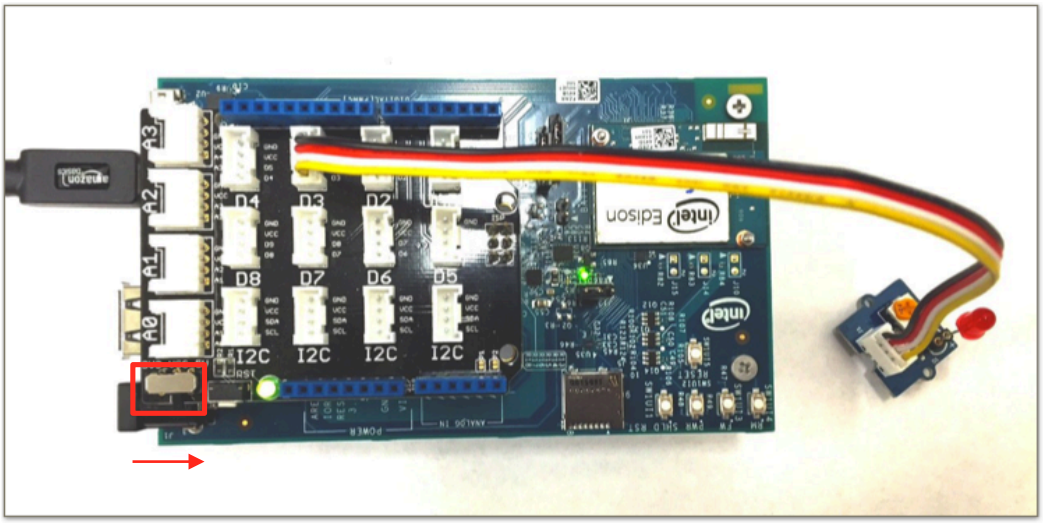



Figure 4: Ensuring that the Grove Base Shield is operating at 5V

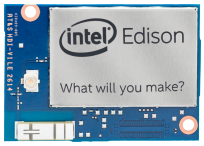
7. Access the shell on your Intel Edison, and issue the command shown below.

   **$ cd ~/tutorial4_examples**

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

   **$ vi blink.c**

   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(int sig)
{
    // if-statement will verify if the signal is the SIGINT signal before proceeding
    if (sig == SIGINT)
        run_flag = 0;
}

int main()
{
    // declare led as a mraa_gpio_context variable (General Purpose Input/Output)
    mraa_gpio_context led;
    // initialize interface D3 on the Grove Base Shield for led
    led = mraa_gpio_init(3);
    // configure led GPIO interface to be an output pin
    mraa_gpio_dir(led, MRAA_GPIO_OUT);

    // when the 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 LED on by setting the output to logical true
        mraa_gpio_write(led, 1);
        // use the sleep function so that humans can detect the change in LED state
        sleep(1);
        // turn the LED off by setting the output to logicla false
        mraa_gpio_write(led, 0);
        sleep(1);
    }

    mraa_gpio_close(led);
    return 0;
}
```

Figure 5: C code source file blink.c

9. Compile the code.

   **$ gcc -lmraa -o blink blink.c**

10. Execute the binary file.

    **$ ./blink**

11. Issue the **ctrl-C** keystroke to terminate the **blink** process.

For more information, refer to the below link.

http://iotdk.intel.com/docs/master/mraa/gpio_8h.html

## SIGINT on Button Press

In the previous section, GPIO – Blink LED, the code contained a function named **signal**. This function is used to specify system behavior when a signal such as SIGINT is received. Consider the code from Figure 5. If the **SIGINT** signal is detected, the 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**.

A user can generate the **SIGINT** signal by issuing the **ctrl-C** keystroke while interacting with the Linux Operating System shell program. In certain cases, the user may not have access to the shell. As such, this section will describe how to use a button provided in the **Grove – Starter kit for Arduino** to generate the **SIGINT** signal.

1. Configure the hardware such that it matches Figure 6.

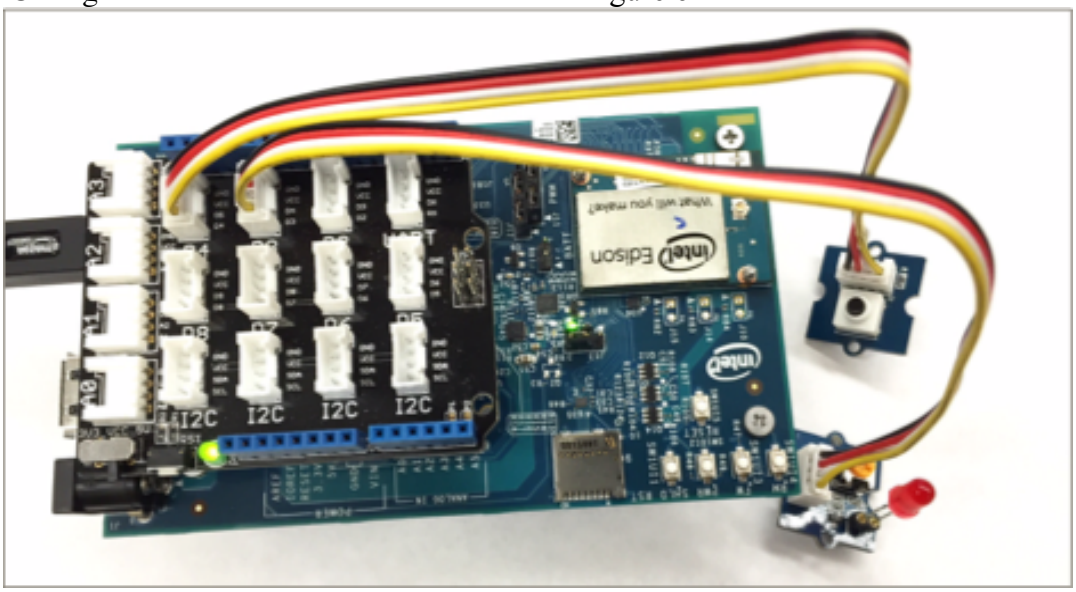



**Figure 6: Hardware configuration to demonstrate generation of signals using a button press**

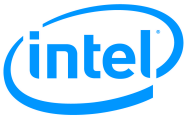
2. Ensure the Grove Base Shield is operating at 5V.
3. Access the shell on the Intel Edison and navigate to **~/tutorial4_examples**.
4. **$ vi gpio_interrupt.c**

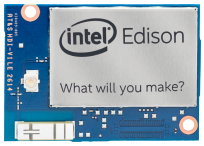
5. Enter the C code from Figure 7 into the Vim editor.

```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 button press
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 led and btn (button) as mraa_gpio_context variables
    mraa_gpio_context led, btn;
    // initialize interface D3 on the Grove Base Shield for led
    // initialize interface D4 on the Grove Base Shield for btn
    led = mraa_gpio_init(3);
    btn = mraa_gpio_init(4);

    // configure led GPIO interface to be an output pin
    mraa_gpio_dir(led, MRAA_GPIO_OUT);
    // configure btn GPIO interface to be an input pin
    mraa_gpio_dir(btn, MRAA_GPIO_IN);

    // when the the button is pressed, call do_when_interrupted
    mraa_gpio_isr(btn, 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 LED on by setting the output to logical true
        mraa_gpio_write(led, 1);
        // use the sleep function so that humans can detect the change in LED state
        sleep(1);
        // turn the LED off by setting the output to logicla false
        mraa_gpio_write(led, 0);
        sleep(1);
    }
    mraa_gpio_close(led);
    mraa_gpio_close(btn);

    return 0;
}
```

Figure 7: C code source file gpio_interrupt.c

6. **$ gcc -lmraa -o gpio_interrupt gpio_interrupt.c**
7. **$ ./gpio_interrupt**
8. Press the button. Notice that the **gpio_interrupt** process is terminated.
9. 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**.
10. **$ vi gpio_interrupt.c**






11. The original mraa_gpio_isr function is called as follows.

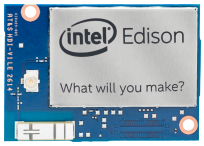
    mraa_gpio_isr(button, **MRAA_GPIO_EDGE_RISING**,
        &do_when_interrupted, NULL);

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

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

12. **$ gcc -lmraa -o gpio_interrupt gpio_interrupt.c**
13. **$ ./gpio_interrupt**
14. Press and hold the button for a 5 seconds.
15. Release the button. Notice that the **gpio_interrupt** process terminates.

For more information about GPIO interrupts in the mraa library, refer to the link shown below.
http://iotdk.intel.com/docs/master/mraa/gpio_8h.html




# 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).

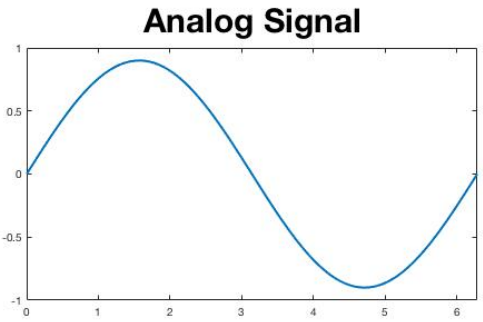
A discrete variable is a variable that is can **only** take **certain values**. In other words, a continuous variable can only take on a **finite set of values**. For example, if the variable Y is discrete, one value of Y is 0.9 and the other is 1.0, Y **cannot** be 0.95. It must either be 0.9, or 1. 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 8.



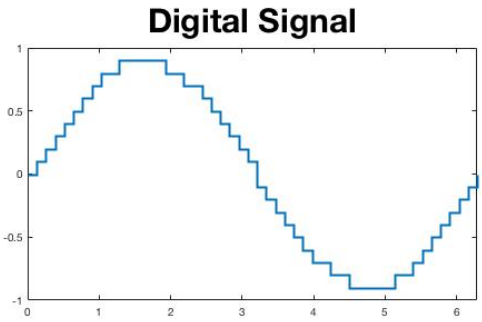




Figure 8: Representing sin(x) as a continuous time analog signal, and as a digital signal

Most data in the real world is continuous value. This means that most real-world signals are **analog signals**. However, computers operate using **binary**. Binary is a discrete, because it can only represent certain numbers. This means that computers must first translate data from being **analog** to **digital** before they perform computations on data. This is usually done with a hardware component called an **Analog to Digital Converter** (ADC).

The ADC chip (ADS7951) on the Arduino breakout board for the Intel Edison can be summarized as follows.

| **Property** | **Datasheet specification** | **Observed Behavior and Real World Implications** |
|---|---|---|
| Sample rate | 1-MHz | The time taken for the function **mraa_aio_read()** to return was determined experimentally to be 150 µs. This equates to about 6.67 kHz sampling frequency<br><br>Using Nyquist's theorem, the highest frequency signal that can be sampled is 3.33kHz<br><br>In reality, the highest frequency signal that can be sampled is around 100Hz |
| Analog Supply Range | 2.7V – 5.25V | The ADC chip is wired such that it can either be supplied with 3.3V or 5V<br><br>The Intel Edison cannot read a voltage level higher than 5V |
| Resolution | 8/10/12 bits | 8 bits can represent numbers from 0-255<br>10 bits can represent numbers from 0-1023<br>12 bits can represent numbers from 0-4095 |

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

1. Power down the Intel Edison, and remove all power and USB cables.
2. Assemble the Edison, a base shield, and a rotary angle sensor as shown in the picture below. A rotary angle sensor is sometimes called a **potentiometer**.

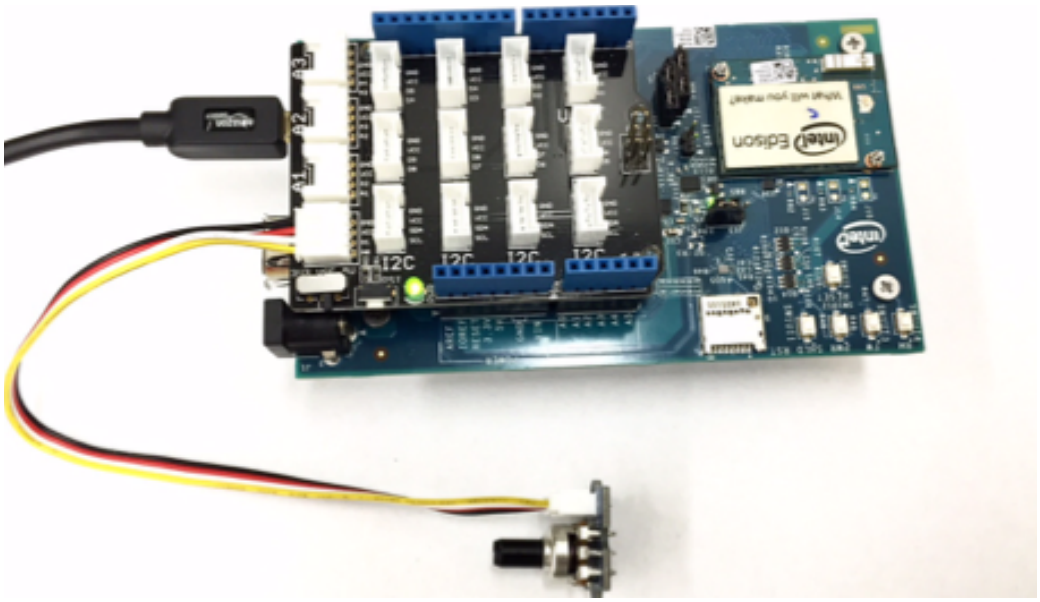



**Figure 9: Hardware configuration to demonstrate how to read data from an analog sensor**

3. Ensure the Grove Base Shield is operating at 5V.
4. Connect the USB and power cables to the Intel Edison.
5. Access the shell on the Intel Edison.
6. **$ vi rotary.c**
7. Enter the C code from Figure 10 into the Vim editor.

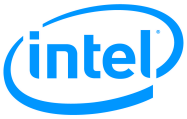
```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <mraa/aio.h>

// flag checked in while loop in main function
// it is initialized to be 1, or logical 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()
{
    // 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(0);

    signal(SIGINT, do_when_interrupted);

    while (run_flag) {
        // mraa_aio_read will read a the voltage provided by an analog device
        // the voltage read will be converted into a 10 bit integer
        // A 10 bit integer can represent 2^10 (1024) discrete values
        value = mraa_aio_read(rotary);
        printf("%d\n", value);
        sleep(1);
    }
    mraa_aio_close(rotary);

    return 0;
}
```

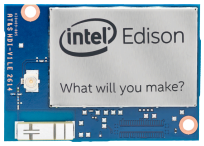
Figure 10: C code source file rotary.c

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

For more information about GPIO interrupts in the mraa library, please refer to the below link**:**

http://iotdk.intel.com/docs/master/mraa/aio_8h.html






## I2C (Inter-Integrated Circuit)

$I^2C$ stands for inter-integrated circuit. It is a **digital** and **serial** communication protocol that can have **multiple masters** and **multiple slaves**. This means that data is sent one byte at a time, rather than sending multiple bytes at the same time. Sending multiple bytes at the same time is called **parallel** communication, and is not very widely used anymore. Typical applications of **parallel** communication included communication to printers, and other laboratory equipment.

The $I^2C$ protocol is **low-speed** and is typically used for **short range** communication (few meters). This is because of the typical voltage levels, data transfer speeds and total bus capacitance of **400 pF**. A logical "low" is represented with **0V**, and a logical "high" is represented with **+5V, or +3.3V**. For longer range communication, consider using the **RS-232** protocol that specifies a logical "low" as **+3 to +15V** and a logical "high" as **-3V to -15V**.

Typical data transfer rates are **100kbit/s**. However, there are also a range of other communication speeds available: **10kbit/s** (low-speed), **400kbit/s** (Fast mode), **1Mbit/s** (Fast mode plus), and **3.4Mbit/s** (High Speed mode).

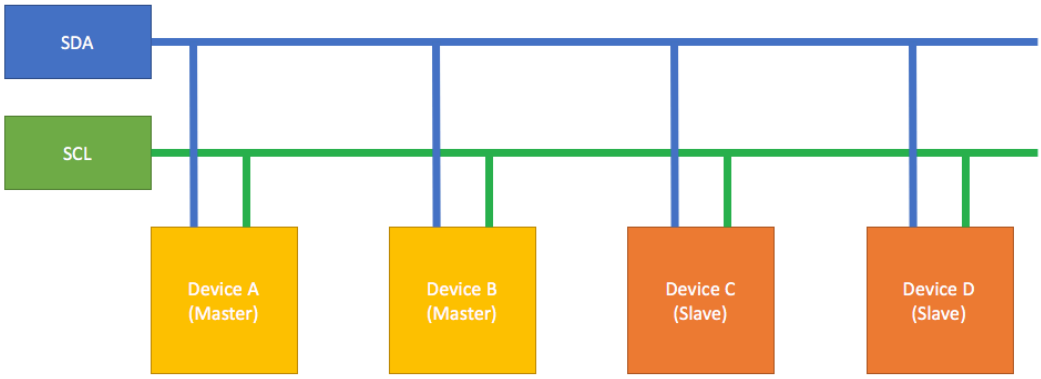


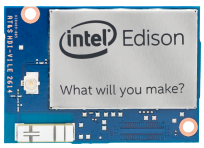

Figure 11: Example of I2C wiring

From Figure 11 it can be seen that the $I^2C$ protocol uses two data lines. The first is **SDA**, which is the **data line**. The second is **SCL**, which is the **clock line**.

There are two roles for each node: **master** and **slave**. The **master** is responsible for **generating the clock**, and **initiating communication** with the **slave**. The **slave listens for the clock**, and **responds when addressed by the master**.

While the details of the protocol are not required to use $I^2C$ to communicate with devices, an explanation of it is provided below.

1. The master initiates communication by sending a **start bit**.
    a. If there are multiple masters, ensure they are **all** configured in **multi-master mode**.
    b. If there are multiple masters configured in multi-master mode attempting to communicate with slaves at the same time, the master devices will **arbitrate** in order to **decide which master communicates first**.




c. Devices configured as **multi-masters** will check if the communication bus is busy prior to initiating communication.
d. If a device is configured in **single-master** (device A) mode when there is a **single-master** or **multi-master** (device B) already communicating, the **single-master** device's (device A's) behavior is undefined. It (device A) may interrupt the other device's (device B's) communication.

2. The master specifies the **address** of the device it wishes to communicate with.
   a. The address is a 7-bit integer. This means $I^2C$ typically supports up to 128 devices at a time.
   b. Nodes are **not allowed** to have the **same addresses** if they are on the same bus. Hardware can usually be reconfigured to have a new address.
   c. If a system uses chips with identical hard-coded $I^2C$ addresses, a **multiplexer** can be used.
3. The slave responds with an **ACK** (acknowledgement) bit.
4. The master transmits the data.
   a. If the master **does not** send a **stop** bit, the slave node will interpret the **next byte** to be the next **part of the same message**.
5. The master transmits a **stop bit**.
6. If the master requests a response from the slave, the slave responds with data.

For more details regarding the $I^2C$ protocol, please visit the website listed below.
http://www.i2c-bus.org/

## Implementation

This section will provide instructions on how to interface with an $I^2C$ device such as the Grove – LCD RGB Backlight V4.0 screen.

1. Power the Intel Edison down.
2. Remove all power and USB cables from the Intel Edison.
3. Assemble the Edison, a base shield, and a LCD RGB screen as shown in the picture below. The LCD RGB screen will receive $I^2C$ messages if it is connected to any of the four slots labelled as **I2C**.

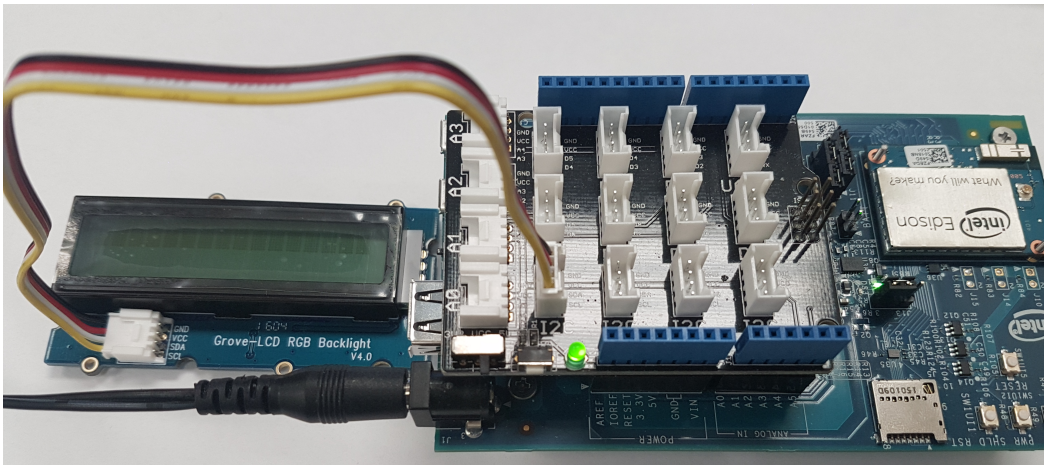
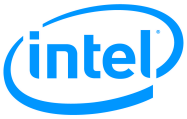
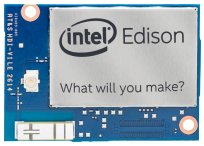



**Figure 12: Hardware configuration to demonstrate communication using the I2C protocol**



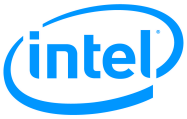


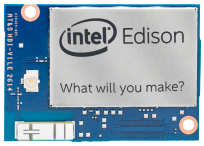

4. Ensure the Grove Base Shield is operating at 5V.
5. Connect the USB and power cables to the Intel Edison.
6. Access the shell on the Intel Edison.
7. **$ vi rgb_screen.c**
8. Enter the C code from Figure 13 into the Vim editor.

```c
#include <mraa/i2c.h>

// define the registers and addresses for the RGB backlit screen
#define RGB_ADDRESS (0xc4 >> 1)
#define REG_MODE1 0x00
#define REG_MODE2 0x01
#define REG_OUTPUT 0x08

#define REG_RED 0x04
#define REG_GREEN 0x03
#define REG_BLUE 0x02

// function that will send the byte 'dta' to the address 'addr' on the I2C bus 'rgb_i2c'
void rgb_command(mraa_i2c_context rgb_i2c, uint8_t addr, uint8_t dta)
{
    uint8_t to_send[2] = {addr, dta};
    mraa_i2c_write(rgb_i2c, to_send, 2);
}

// function to set the color of the RGB backlit screen
void set_RGB_color(mraa_i2c_context rgb_i2c, uint8_t r, uint8_t g, uint8_t b)
{
    rgb_command(rgb_i2c, REG_RED, r);
    rgb_command(rgb_i2c, REG_GREEN, g);
    rgb_command(rgb_i2c, REG_BLUE, b);
}

int main()
{
    mraa_i2c_context rgb_i2c;

    // initialize the mraa library
    mraa_init();

    rgb_i2c = mraa_i2c_init(0);

    // set up the i2c busses
    mraa_i2c_address(rgb_i2c, RGB_ADDRESS);

    // set up the rgb backlight
    rgb_command(rgb_i2c, REG_MODE1, 0);
    rgb_command(rgb_i2c, REG_OUTPUT, 0xFF);
    rgb_command(rgb_i2c, REG_MODE2, 0x20);

    // set the RGB values to 0, 0, 255, indicating the screen should be completely blue
    set_RGB_color(rgb_i2c, 0, 0, 255);
}
```

Figure 13: C code source file rgb_screen.c

9. **$ gcc -lmraa -o rgb_screen rgb_screen.c**
10. **$ ./rgb_screen**
11. Notice how the screen displays blue light.
12. Edit the system such that the screen displays red light.






For more information about interfacing with $I^2C$ peripheral devices through the mraa library, please refer to the link shown below.

http://iotdk.intel.com/docs/master/mraa/i2c_8h.html

## Tasks

1. Use the rotary sensor to adjust the amount of blue present on the RGB screen

   **Notes**:
   a. mraa_aio_read() returns a 10 bit integer by default.
      set_rgb_color() requires color values to be specified by 8 bit integers.

      How can you scale a 10bit integer to an 8bit integer?
   b. Experiment with the mraa_aio_read_float() function.
   c. Experiment with the mraa_aio_set_bit() function.

2. Implement a system where you can control each of the three colors on the RGB screen using different kinds of sensors.