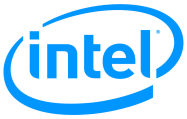
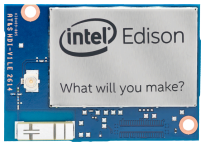




Intel® Edison Tutorial: SPI, PWM, and More GPIO

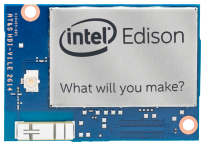
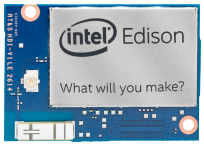


Table of Contents

Revision history		
Version	**Date**	**Comment**
1.0	1/11/2015	Initial release



Introduction

This tutorial will build on the document labelled ***Intel Edison Tutorial - GPIO and I2C Interfaces*** by building systems that use more interfaces and what is beneath the abstractions of I/O programming. In this tutorial, you will learn to:

1. Control the brightness of an LED using a PWM signal.
2. Control a servo using a PWM signal.
3. Establish an SPI communication between the Intel Edison and the Arduino Uno.
4. Access the GPIO pins without MRAA.
5. Configure the shield pins.

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

PWM – LED Control

Pulse-width modulation (PWM) is a modulation technique used to encode messages into a pulsing signal. The key metrics for a PWM signal are the **duty cycle** and the **period** or **frequency**. These are illustrated in the figure below.

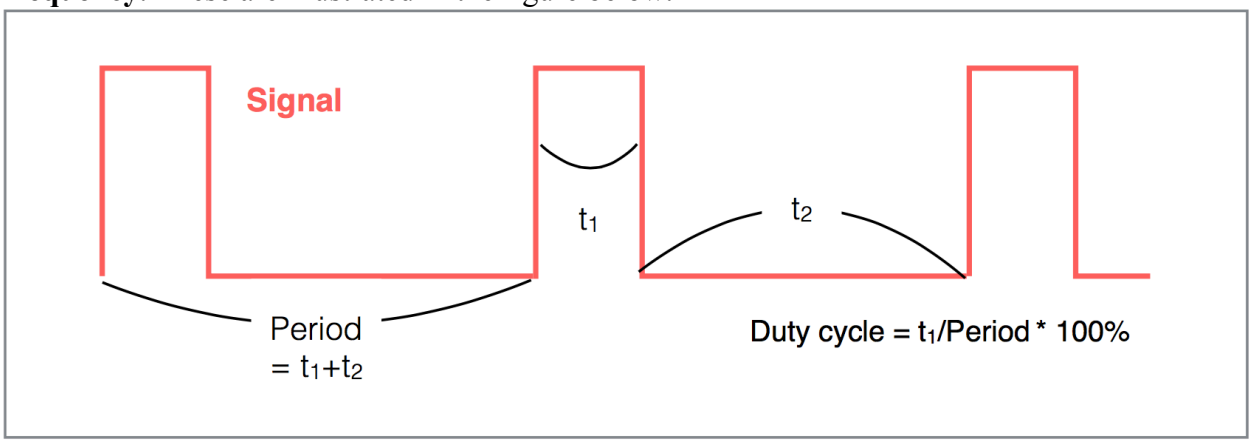


Figure 1: Key metrics for a PWM signal

PWM signals are typically used to approximate the generation of an analog signal. They are mainly used to control power supplied to electronic devices.

For more information about PWM, please refer to the below link:

https://en.wikipedia.org/wiki/Pulse-width_modulation

Follow the below steps in order to control the brightness of a LED by writing software on the Intel Edison.

1. Insert the Grove Base shield into the breakout.
2. Connect an LED to D6 (pin 6) using an LED socket, which has a current-limiting resistor to protect the LED from high current. (Note: digital pins with "~" before the number are available for PWM)

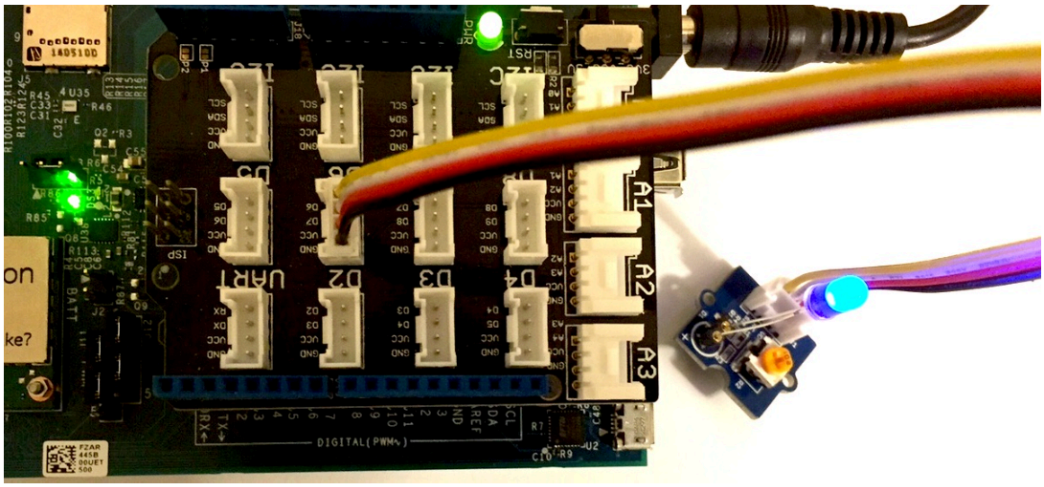
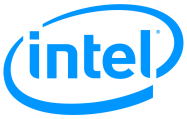
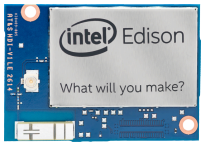


Figure 2: Hardware configuration for operation of LED via PWM signal





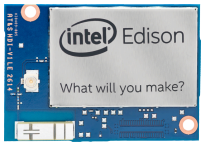
3. Power on the system with either a powered USB hub (connect it to your computer to supply more power to the Edison) or an external power supply.
4. Access the shell on your Intel Edison. For more information, please refer to the document labelled ***Intel Edison Tutorial – Introduction, Shell Access and SFTP***.
5. Navigate to **/home/root/tutorial5_examples** directory.
6. **$ vi pwm_led.c**
7. Type the following code.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <mraa/pwm.h>

#define MAXBUFSIZ 1024

int main(){
      float brightness;
      char user_input[MAXBUFSIZ];
      mraa_pwm_context pwm;
      pwm = mraa_pwm_init(6);

      if (pwm == NULL) {
            fprintf(stderr, "Failed to initialize.\n");
            return 1;
      }

      mraa_pwm_period_us(pwm,200);
      mraa_pwm_enable(pwm, 1);

      while(1){
            printf("Enter brightness value (0-100): ");
            scanf("%s", user_input);
            brightness = atof(user_input);
            if (brightness > 100 || brightness < 0)
                  printf("Error: Choose between 0 and 100\n");
            else {
                  brightness = brightness/100;
                  mraa_pwm_write(pwm, brightness);
            }
      }
      return 0;
}
```

Figure 3: Contents of C code source file pwm_led.c

1. **$ gcc -lmraa -o pwm_led pwm_led.c**
2. **$ ./pwm_led**

As previously mentioned PWM signals are typically used to approximate the generation of an analog signal. Consider the scenario where you wish to have the Intel Edison supply a 2.5V LED driving signal such that the attached LED is emitting light, but not at full intensity. However, the GPIO pins can only supply either GND voltage or VDD voltage. For the case of the Intel Edison, the VDD voltage is typically 5V. To resolve this issue, we can use PWM to switch the output voltage of a digital pin between 0V and 5V. The ratio of how long the PWM signal is **high** to



how long it is **low** is called the duty cycle.

$$D = \frac{t_{high}}{t_{high} + t_{low}}, f = \frac{1}{t_{high} + t_{low}}$$

The brightness is determined by duty cycle (how long the LED is on for a given period). Now, look at pwm_led.c and see how PWM is used to control the brightness. Please note that the full range of the LED brightness is not available with this program in order to prevent any damages to the devices.

Controlling the brightness on LED with PWM is possible because LEDs have very fast response. On the other hand, fluorescent light bulbs have slow response. Controlling the brightness of a fluorescent light bulb by providing a PWM signal without additional hardware may be challenging. Another device with very fast response is a servo. To build a system that controls a servo, please proceed to the next section labelled ***PWM – Servo Control***.

PWM – Servo Control

A servo motor is a device that contains gears that control the rotation of a shaft. Typical hobby servo motors have shafts which can be positioned at various angles (usually between 0 and 180 degrees). By providing an encoded PWM signal to the input of the servo motor, this angle can be precisely controlled.

For more information, please refer to the below links:

- http://www.seattlerobotics.org/guide/servos.html
- https://en.wikipedia.org/wiki/Servo_(radio_control)

The servo motor provided in the Grove – Starter Kit for Arduino responds to the width of a pulse (ie: t_1 from Figure 1 above). For most servos, a 0.5ms pulse width (t_1) results in the full left position and 2.5ms results (t_1) in the full right position. The servo also expects the period (ie: $T = t_1 + t_2$ from Figure 9 above) of the PWM signal to be approximately 20ms. Follow the steps below to implement a servo control system.

1. To ensure that PWM is not enabled, power down the Intel Edison. Remove all power supplies and cables from the Intel Edison once it is powered down.

   Wait 10 seconds, reconnect all cables, and access the shell on your Intel Edison.
2. Insert the Grove Base shield into the breakout.
3. Connect a servo to D6 and a rotary angle sensor to A0. Ensure that you have a power supply. If you do not have a power supply, the Intel Edison may get caught in a reboot loop as the USB interface does not provide sufficient current to the servo.

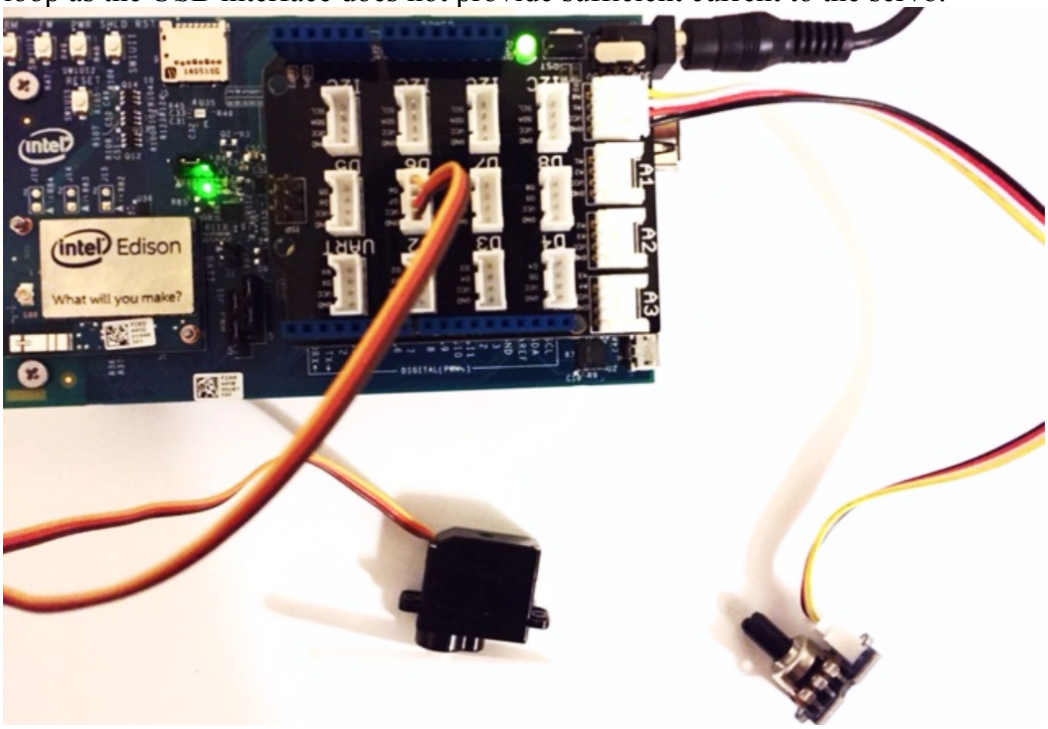
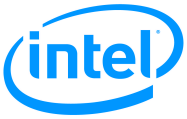
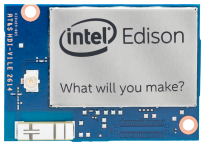


Figure 4: Hardware configuration to enable servo control via user interaction with rotary angle sensor



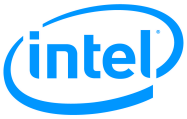


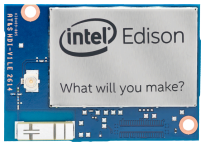
4. If the Edison reboots as soon as you connect a servo, your system is not supplied with enough power. Make sure to use a powered USB hub or an external power supply that can supply enough power.
5. Navigate to the directory named ***~/tutorial5_examples***.
6. **$ vi pwm_servo.c**
7. Type the following code.

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

sig_atomic_t volatile isrunning = 1;

void sig_handler(int sig) {
    if (sig == SIGINT)
        isrunning = 0;
}

int main(){
    signal(SIGINT, &sig_handler);

    uint16_t rotary_value = 0;
    float value = 0.0f;
    mraa_pwm_context pwm;
    mraa_aio_context rotary;

    pwm = mraa_pwm_init(6);
    rotary = mraa_aio_init(0);

    if (pwm == NULL || rotary == NULL) {
        return 1;
    }

    mraa_pwm_period_ms(pwm,20);
    mraa_pwm_enable(pwm, 1);

    while(isrunning){
        rotary_value = mraa_aio_read(rotary);
        //convert to 0.00 to 1.00 scale.
        value = ((float)rotary_value)/1023;

        //convert to 0.025 to ~0.1 scale (avoid rattle).
        value = value/13.33;
        value = value + 0.025;

        printf("%f\n", value);
        mraa_pwm_write(pwm, value);
        usleep(50000);
    }
    mraa_pwm_write(pwm, 0.025f);
    return 0;
}
```

Figure 5: Contents of C code source file to actuate servo based on user interaction with rotary angle sensor.

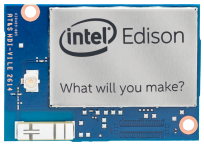
8. **$ gcc -lmraa -o pwm_servo pwm_servo.c**





9. **$ ./pwm_servo**
10. Turn the rotary angle sensor to control the servo.
    Note: If your Edison reboots, it needs a more powerful power supply.





SPI

The Serial Peripheral Interface (SPI) bus is a synchronous serial communication between one master device and one or more slave devices for a short distance. Along with I2C, SPI is primarily used in embedded systems. The motivation of the development of these buses is reducing the number of wires. A common way to connect peripherals to a CPU/microcontroller is connecting through parallel address and data busses. This way, a bus can result in a lot of wires on PCB (printed circuit board). A bus with many wires is not desirable for embedded systems. In comparison to parallel buses, SPI operates with only four wires. A figure below illustrates a single master to a single slave SPI bus.

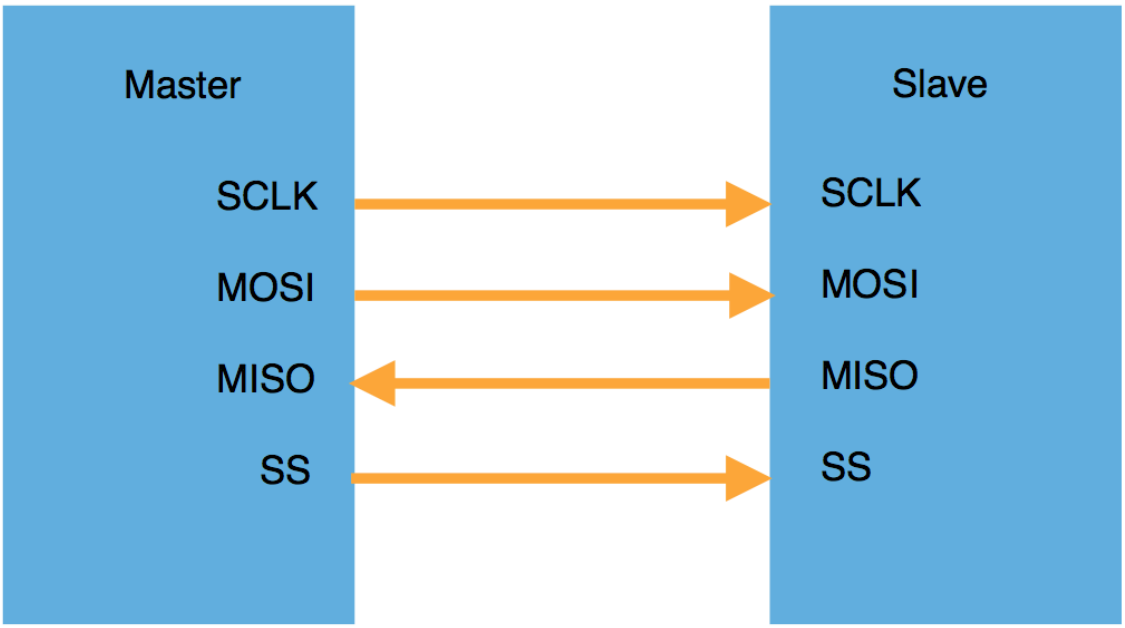


Figure 6: SPI between a single master and a single slave

SCLK (serial clock) is the clock line. The master device sends the clock signal to its slave devices through SCLK, which synchronizes the data communication. MOSI (master out, slave in) and MISO (master in, slave out) are the data lines. The reason why there are two data lines in SPI is to have full duplex communication (i.e. simultaneous communication in both directions). Every clock cycle, the master device sends a bit to the slave via MOSI and the slave device sends a bit to the master via MISO. SS (slave select) is the line, which the master device uses to select the slave device for data transmission/request by pulling down (i.e. making the signal to logic level 0).

There are four different transmission modes, which depend on the master devices configuration of the clock phase (CPHA) and the clock polarity (CPOL). The details of the modes, clock phase, and clock polarity are out of the scope of this tutorial. However, you are encouraged to learn about them.

In this tutorial, we will implement SPI in mode 0 (CPHA = 0 and CPOL = 0) between an Intel Edison and an Arduino Uno.

First, we need to configure the wire connection between these devices as shown in Figure 1. The Edison's Arduino-compatible breakout and the Arduino Uno have the same pinout as shown in Figure 2 below.

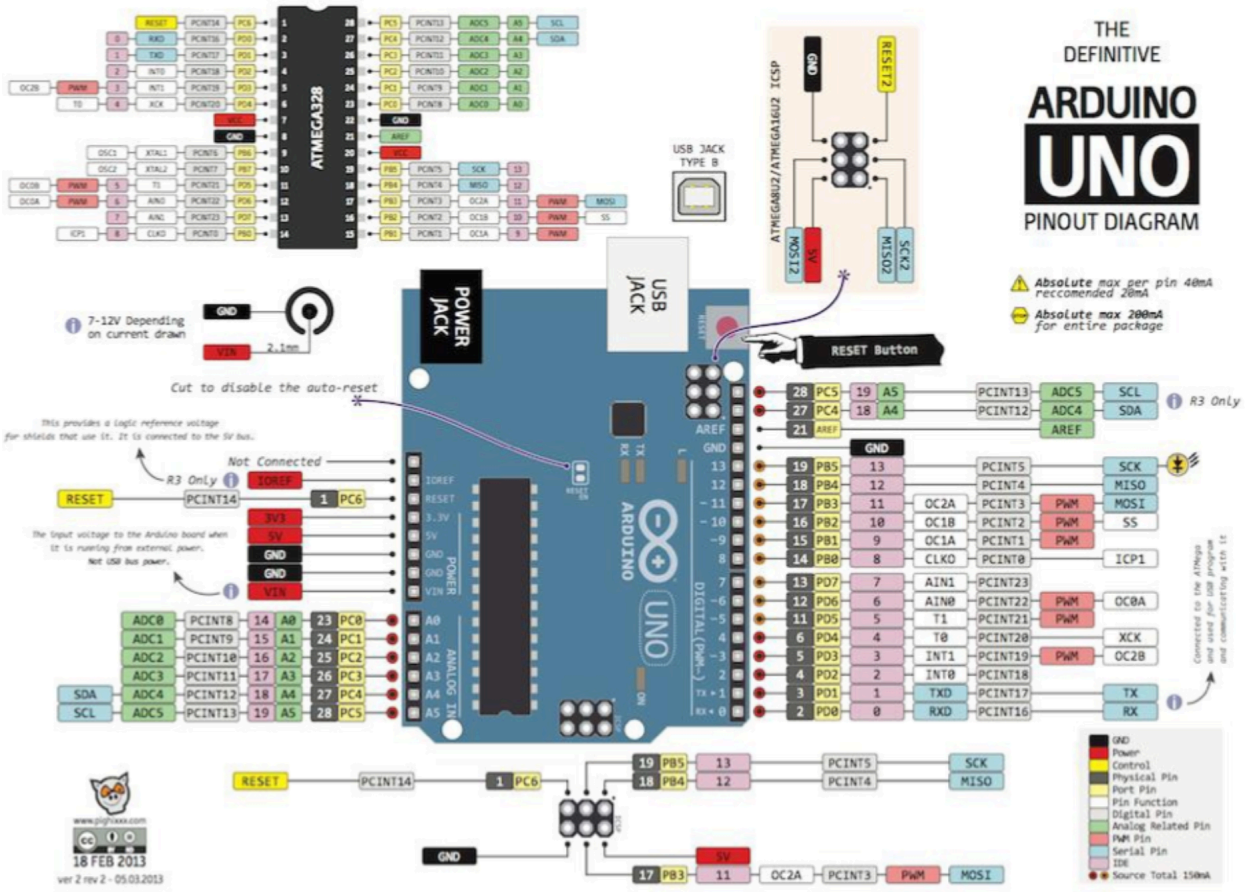
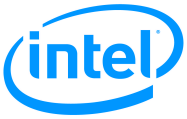
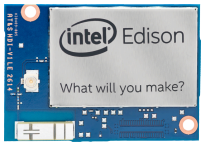


Figure 7: Pinout of the Arduino Uno board

Figure 2 shows that SCLK, MISO, MOSI, and SS are Pin 13, Pin 12, Pin 11, and Pin 10 respectively on both the Edison and the Arduino Uno. The Edison can only be the master device. Thus, we will configure the Edison as the master device and the Uno as the slave device. Please follow the steps below.

1. Connect the Edison and the Uno with four wires as described below.
    - Edison's Pin 13 to Uno's Pin 13 (SCLK)
    - Edison's Pin 12 to Uno's Pin 12 (MISO)
    - Edison's Pin 11 to Uno's Pin 11 (MOSI)
    - Edison's Pin 10 to Uno's Pin 10 (SS)
2. Serial connect or SSH into the Edison.
3. **$ mkdir ~/tutorial5_examples**
4. **$ cd ~/tutorial5_examples**
5. **$ vi spi.c**
6. Type the following C code.

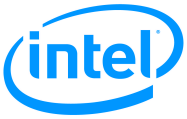


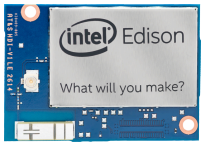



```c
#include <unistd.h>
#include <stdint.h>
#include <mraa/spi.h>

int main()
{
     mraa_spi_context spi;
     spi = mraa_spi_init(0);
     uint8_t data;
     int recv_int;
     int i;

     while (1) {
          for (i = 0; i < 10; i++) {
               data = i;
               //send data and receive
               recv_int = mraa_spi_write(spi, data);
               printf("Received: %d\n", recv_int);
               usleep(200000);
          }
     }

     return 0;
}
```

Figure 8: Sending data using the mraa library and the SPI protocol, spi.c

7. **$ gcc -lmraa -o spi spi.c**
8. As shown in the code above, MRAA library abstracts the hardware configuration of SPI. However, there is no library for the Arduino Uno to be set up as a slave device. It is important to understand the hardware of the Uno's microcontroller (ATmega328). We will come back to the details of this.
9. Connect the Uno to your computer and open Arduino IDE.
10. Select the right board and the port for the Uno.
11. Upload the following sketch to the Uno.

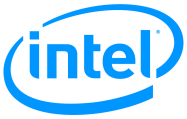


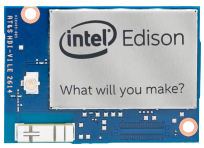



```
#include <SPI.h>

int recv;
int i;

void setup() {
  spi_slave_init();
  Serial.begin(9600);
  i = 10;
}

void loop() {
  recv = spi_transfer(i);
  if (i == 0) {
    i = 10;
  }
  i--;

  Serial.print("Received: ");
  Serial.println(recv);
}

void spi_slave_init() {
  //set the directions of the pins
  pinMode(SCK, INPUT);
  pinMode(MOSI, INPUT);
  pinMode(MISO, OUTPUT);
  pinMode(SS, INPUT);

  SPCR = 0x00;
  //enable SPI
  SPCR |= (1 << SPE);
}

//Receive and Send data
int spi_transfer(int send_data) {
  //This part is for reception
  int recv_data;
  while (!(SPSR & (1 << SPIF))) {
    //wait for complete transfer
  };
  recv_data = SPDR;
 //This part is for transmission
  SPDR = send_data;
  while (!(SPSR & (1 << SPIF))) {
    //wait for complete transfer
  };

  return recv_data;
}
```

Figure 9: Arduino sketch to receive SPI data

12. Open serial monitor on Arduino IDE.
13. Go back to the SSH session (or serial console).
14. **$ ./spi**





15. Now, you should see that the Edison receives decrementing integer data while the Arduino receives incrementing integer data.

The C code for the Edison is very straightforward. "**spi**" is initialized with the "**mraa_spi_init(0)**" function. Let's consider how this is done. Pins 10,11,12,13 (numbered IO10, IO11, IO12, and IO13) can have different signals as shown in the table below.

Shield pin	GPIO (Linux)	PWM (Linux)	Muxed functions	Notes
IO0	130		UART1_RXD	
IO1	131		UART1_TXD	
IO2	128		UART1_CTS	Note 1.
IO3	12	0	PWM0	Note 2.
IO4	129		UART1_RTS	Note 1.
IO5	13	1	PWM1	Note 2.
IO6	182	2	PWM2	Note 2.
IO7	48		—	
IO8	49		—	
IO9	183	3	PWM3	Note 2.
IO10	41	??	SPI_2_SS1	
			I2S_2_FS	Note 1.
			PWM4_OUT	Note 2.
IO11	43	??	SPI_2_TXD	
			I2S_2_TXD	Note 1.
			PWM5_OUT	Note 2.
IO12	42		SPI_2_RXD	
			I2S_2_RXD	Note 1.
IO13	40		SPI_2_CLK	
			I2S_2_CLK	Note 1.
IO14	44		AIN0	
IO15	45		AIN1	
IO16	46		AIN2	
IO17	47		AIN3	
IO18	14		AIN4	
			I2C_6_SDA	
IO19	165		AIN5	
			I2C_6_SCL	

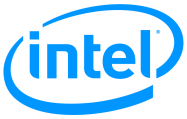
1 *Some additional functions are available on certain SoC pins, such as I2S and UART flow control, but they are not currently supported by the Arduino library. However, it may be possible to use these from Linux.*

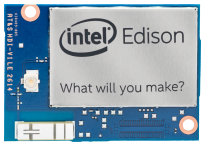
2 *Depends on PWM swizzler. The SoC offers only four PWM pins. A jumper pin matrix labeled "PWM swizzler" on the baseboard allows these four pins to be connected to any subset of the six shield-header pins normally used for PWM. From the factory, IO3, IO5, IO6, and IO9 will be connected to the four available SoC PWM pins as described above. You can manually alter these to connect IO10 or IO11.*

Figure 10 GPIO Mapping (Source: Intel)

The "**mraa_spi_init"** function sets the multiplexers to map the pins so that pins 13, 12, 11, 10 are SCLK, MISO, MOSI, SS. Then, it selects SPI mode 0 and sets transfer in most significant bit first mode. Later in the code, we use "**mraa_spi_write(spi, data)**" to transmit "**data**" to the slave device and this functions returns the received data from the slave.

Now, let's look at the Arduino sketch. First thing to consider is that there are no library functions available for an Arduino used as a slave device. For instance, a library function, "**SPI.begin()**", configures the directions of SCLK, MOSI, MISO, and SS as a master device. Since there is no





available function to initialize the Uno as a slave device, we have to write our own function, such as "**spi_slave_init()**" in the sketch above. The function configures the directions of SCLK, MOSI, MISO, and SS as a slave device.

The next line of the code includes "**SPCR**", which refers to SPI Control Register. A register is an 8-bit memory in a microcontroller. Three registers are used by the Uno for the SPI interface. Other two registers are SPI Status Register (**SPSR**) and SPI Data Register (**SPDR**). SPCR controls the SPI settings as described below. Each column represents each bit of the register.

7	6	5	4	3	2	1	0
SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0

Figure 11 SPCR

SPIE (**SP**I **I**nterrupt **E**nable): when 1, enable SPI interrupt.
SPE (**SP**I **E**nable): when 1, enable SPI.
DORD: when 1, send the least significant bit (LSB) first;
when 0, send the most significant bit (MSB) first.
MSTR: when 1, set the device as a master;
when 0, set the device as a slave.
CPOL: Setting the **C**lock **Pol**arity as 0 or 1.
CPHA: Setting the **C**lock **Pha**se as 0 or 1.
SPR1 and **SPR0** - Sets the SPI speed to maximum when 00 or minimum when 11.

In the sketch, we set all of these eight bits to 0 and doing so results in MSB first mode, slave mode, and mode 0 (CPOL = 0 and CPHA = 0). This setting adheres to the setting of the SPI connection's other end. The C code for the Edison initializes the Edison as a master with also MSB first mode and mode 0. Now, SPI needs to be enabled, so SPCR is bit-masked to set SPE as 1.

SPDR holds the data to be sent or received. We can use this register to read the received data and transmit data. However, we need to be careful because it is a serial-in shift register. In other words, one bit of data is transmitted/received at a time. Therefore, we need to wait until the whole 8-bit data is in the register. This is illustrated in the figure below. If the data to be received is 10101110_2, two least significant bits have not yet received in the sixth clock cycles.

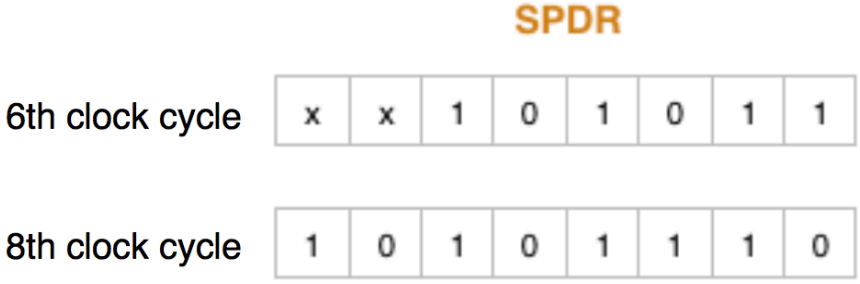


Figure 12 SPDR

In the Arduino sketch, while (!(SPSR & (1 << SPIF))) { } loop will make sure all eight bits are received. When the transmission of data is completed, SPI Interrupt Flag (SPIF) is set to 1. SPIF is a bit in SPI Status Register (SPSR).

Optional Practice

SD cards are SPI devices. Design and implement an SD card reader on the Edison using its SPI interface.

Hint:

1. As shown in the picture below, an SD card adapter's pins can be solder to header pins and then plugged into a breadboard.
2. There are example Arduino sketches on Arduino IDE.

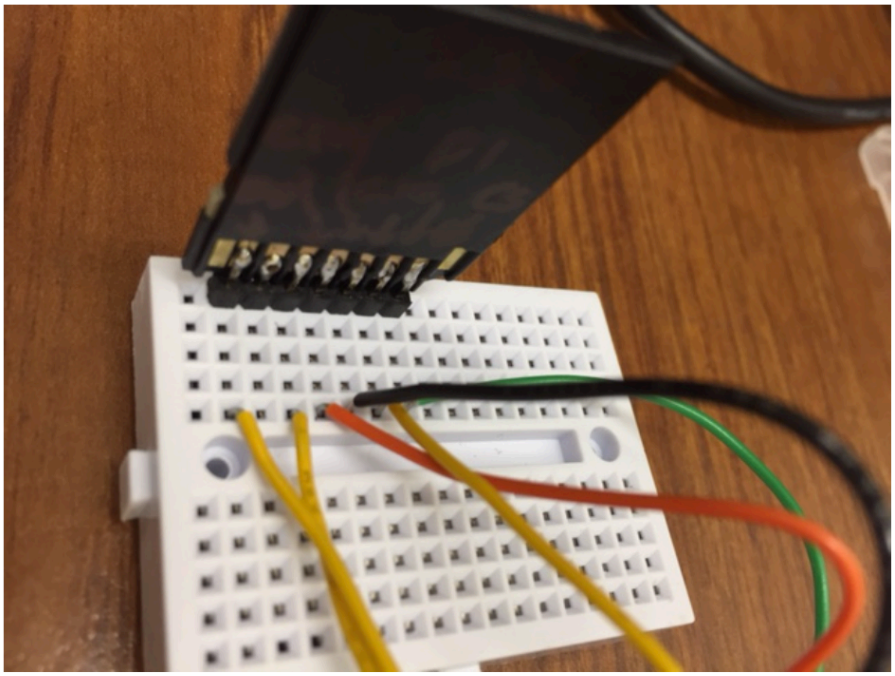
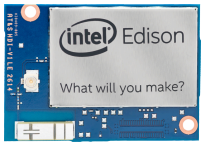


Figure 13 SD Card Reader



GPIO – Interaction Without libmraa

While the mraa library is convenient, there may be situations where you do not have access to it. To address this issue, this section will cover accessing the GPIO pins without the mraa library. Before going into the details, try the following demonstration.

1. Insert the Grove Base shield into the Edison.
2. Connect an LED to D7 (pin 7) using the LED socket.
3. Access the shell on the Intel Edison.
4. **$ echo 48 > /sys/class/gpio/export**

   The error message ***Device or resource busy*** indicates that the value 48 is already in the file.
   $ echo 48 > /sys/class/gpio/unexport
   $ echo 48 > /sys/class/gpio/export

5. **$ echo 255 > /sys/class/gpio/export**

   The error message Device or resource busy indicates that the value 48 is already in the file.
   $ echo 255 > /sys/class/gpio/unexport
   $ echo 255 > /sys/class/gpio/export

6. **$ echo 223 > /sys/class/gpio/export**

   The error message Device or resource busy indicates that the value 48 is already in the file.
   $ echo 255 > /sys/class/gpio/unexport
   $ echo 255 > /sys/class/gpio/export

7. **$ echo high > /sys/class/gpio/gpio255/direction**
8. **$ echo in > /sys/class/gpio/gpio223/direction**
9. **$ echo out > /sys/class/gpio/gpio48/direction**
10. **$ echo 1 > /sys/class/gpio/gpio48/value**
    Now the LED should turn on.
11. **$ echo 0 > /sys/class/gpio/gpio48/value**
    Now the LED should turn off.

First, we need to understand what “48”, “255”, and “223” are. The pin number we refer to, such as D7, is the Arduino shield pin number, which is not the same as the pin number understood by the embedded Linux. Go back to page 8 of this tutorial and take a look at the table. IO7 (digital pin 7) is mapped to GPIO 48 (Linux), which is a SoC (System-on-Chip) pin rather than a shield pin.

There are level-shifters between the SoC GPIO pins and the shield pins as shown in Figure 16 on the next page. These level-shifters must be configured for input/output direction before

configuring the SoC pin direction. The I/O direction of a level-shifter is set via controlling a port expander as shown in the picture. In Linux, this hardware configuration can be done by changing values in directories such as /sys/class/gpio/gpio255. For each shield pin, there is a dedicated directory with user space interfaces to control the hardware. Figure 17 on the next page shows which shield pin is associated with which directory in Linux. For instance, IO7 is associated with /sys/class/gpio/gpio255.

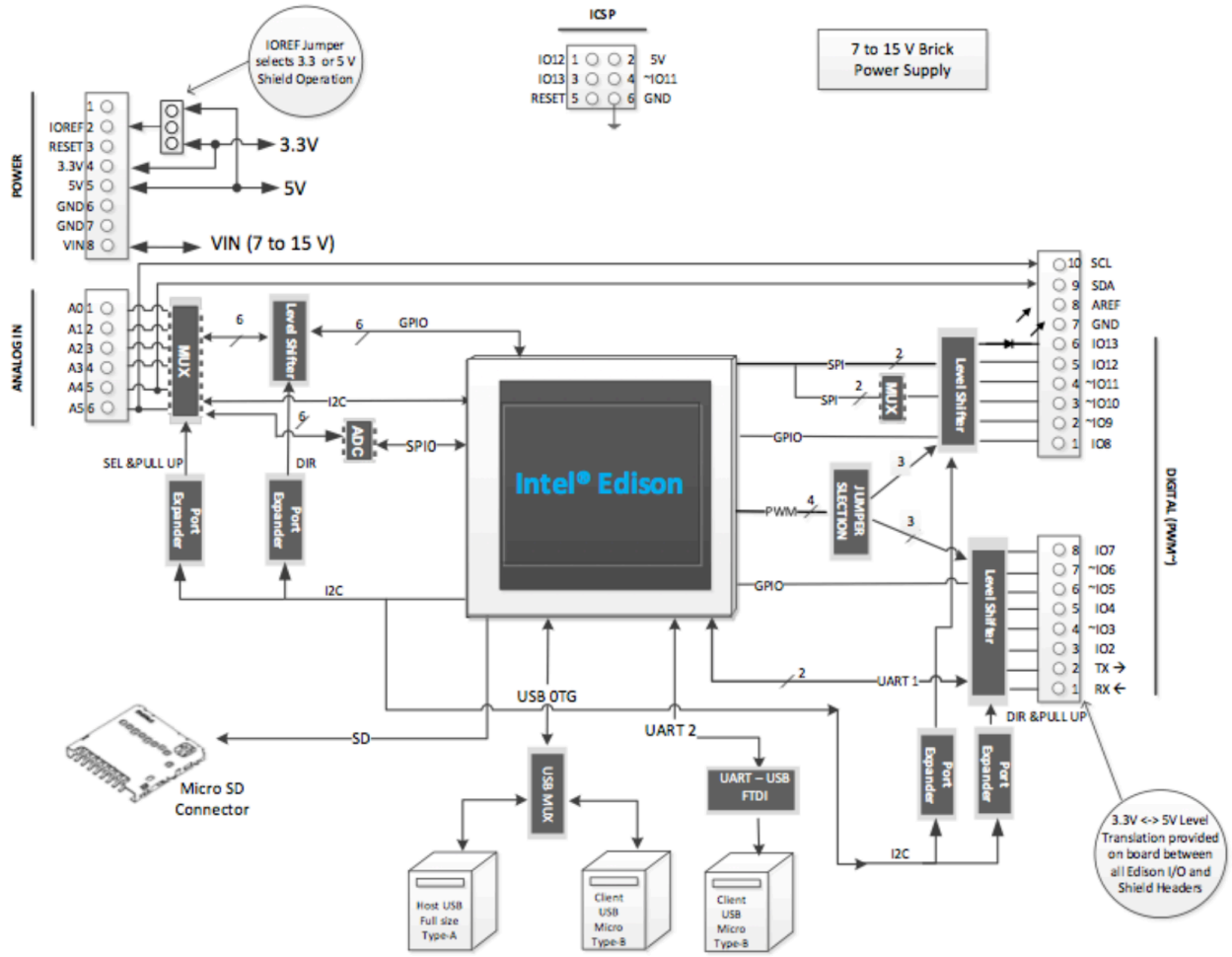
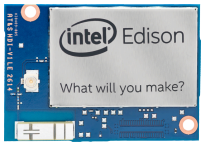


Figure 14 Intel Edison for Arduino Block Diagram (Source: Intel)

In steps 4, 6 and 8, we exported gpio48, gpio255, and gpio223. Exporting these means that we are making them available to use. You can enter "**ls**" to see all available GPIOs. "**echo 48 > /sys/class/gpio/export"** will add gpio48 to the list and **"echo 48 > /sys/class/gpio/unexport"** will delete gpio48 from the list. In the demonstration, we deleted the GPIOs and then made them available so that the GPIOs will have default configurations.

The right side of the table on the next page implies that we can enable/disable an external 47k ohm pull-up resistor. The pull-up resistor is disabled by default. You can make sure that it is disabled by entering "**echo in /sys/class/gpio/gpio223/direction**". You can enable it by setting it to "**high**". However, the details on pull-up/pull-down resistors are out of this tutorial's scope.



Shield pin	Output enable GPIO (high = output)			Pullup enable GPIO		
	Pin	Linux	Power-on default [1]	Pin	Linux	Power-on default [2]
IO0	U34_IO0.0	248	Pulled-down input	U39_IO0.0	216	Pulled up input
IO1	U34_IO0.1	249	Pulled-down input	U39_IO0.0	217	Pulled up input
IO2	U34_IO0.2	250	Pulled-down input	U39_IO0.0	218	Pulled up input
IO3	U34_IO0.3	251	Pulled-down input	U39_IO0.0	219	Pulled up input
IO4	U34_IO0.4	252	Pulled-down input	U39_IO0.0	220	Pulled up input
IO5	U34_IO0.5	253	Pulled-down input	U39_IO0.0	221	Pulled up input
IO6	U34_IO0.6	254	Pulled-down input	U39_IO0.0	222	Pulled up input
IO7	U34_IO0.7	255	Pulled-down input	U39_IO0.7	223	Pulled up input
IO8	U34_IO1.0	256	Pulled-down input	U39_IO0.7	224	Pulled up input
IO9	U34_IO1.1	257	Pulled-down input	U39_IO0.7	225	Pulled up input
IO10	U34_IO1.2	258	Pulled-down input	U39_IO0.7	226	Pulled up input
IO11	U34_IO1.3	259	Pulled-down input	U39_IO0.7	227	Pulled up input
IO12	U34_IO1.4	260	Pulled-down input	U39_IO0.7	228	Pulled up input
IO13	U34_IO1.5	261	Pulled-down input	U39_IO0.7	229	Pulled up input
IO14	U16_IO0.0	232	Pulled-down input	U17_IO1.0	208	Pulled up input
IO15	U16_IO0.1	233	Pulled-down input	U17_IO1.1	209	Pulled up input
IO16	U16_IO0.2	234	Pulled-down input	U17_IO1.2	210	Pulled up input
IO17	U16_IO0.3	235	Pulled-down input	U17_IO1.3	211	Pulled up input
IO18	U16_IO0.4	236	Pulled-down input	U17_IO1.4	212	Pulled up input
IO19	U16_IO0.5	237	Pulled-down input	U17_IO1.5	213	Pulled up input

1 *These pins are externally pulled down inputs at power-on. This effectively selects input direction for level shifters.*
2 *These pins are internally pulled up inputs at power-on. This effectively enables pullups (as 100 kohm + 47 kohm in series).*

Figure 15 Pin Direction and Pull-up Control (Source: Intel)

We set the I/O directions on gpio255 (level-shifter) and gpio48 (SoC) as output by entering commands, "**echo high > /sys/class/gpio/gpio255/direction**" and "**echo out > /sys/class/gpio/gpio48/direction**". Then, we can set the output as logical level 1 or logic level 0 by entering "**echo 1 > /sys/class/gpio/gpio48/value**" or "**echo 0 > /sys/class/gpio/gpio48/value**".

Optional practice

1. Repeat this demonstration for IO8.
2. Write a C program that blinks an LED without using the MRAA library or making a system call to run the commands presented above.
   Hint: you can open these interfaces and read/write to them.

Shield Pin Configuration

This section is an extension to the previous section. The table on page 8 shows that IO7 is available for GPIO only and "Muxed functions" column is blank for IO7. Let's look at IO10. Unlike IO7, IO10 is available for GPIO, SPI, I2C, and PWM. In this demonstration, we will set up IO10 as a GPIO input. Try the following steps.

1. Connect to a button sensor to the breakout as shown below.
   1) Yellow – Pin 10
   2) Red – VCC (5V)
   3) Black – GND

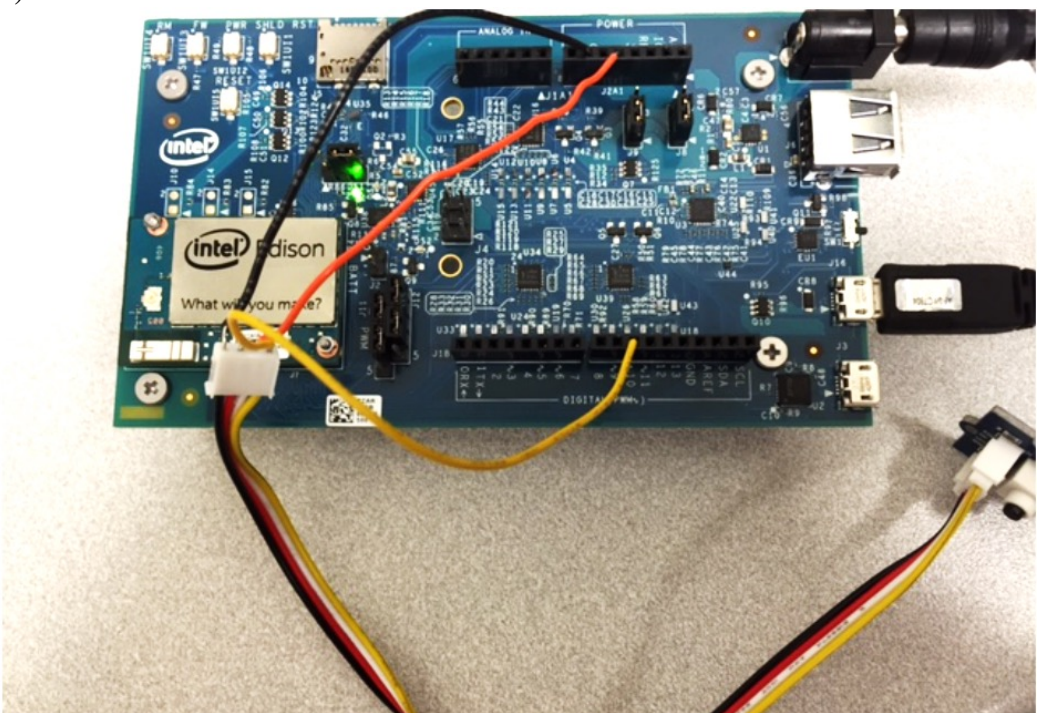
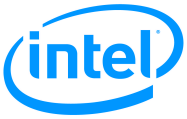
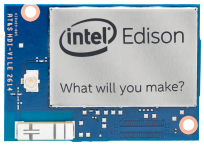


Figure 16 Hardware Setup for Shield Pin Configuration Demo

2. Export GPIOs.
   1) **$ echo 41 > /sys/class/gpio/export**
      - SoC pin number for Linux
   2) **$ echo 263 > /sys/class/gpio/export**
      - GPIO for multiplexing control.
      - Let's look at the block diagram on page 17. IO10 is connected to a level-shifter, which is connected to a multiplexer (MUX). With multiplexing control, we can set IO10 as a GPIO pin.
      - The table on page 21 shows the multiplexing control for pin configuration. The GPIO pin mux for IO10 can be accessed as GPIO 263 and GPIO 240. First, we need to set GPIO 263 as high to select GPIO/SPI (Step 4 below). Then, we need to set GPIO 240 as low to select GPIO (Step 5 below).
   3) **$ echo 240 > /sys/class/gpio/export**
      - GPIO for multiplexing control.
   4) **$ echo 258 > /sys/class/gpio/export**
      - GPIO for pin direction



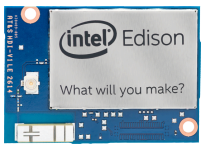


5) **$ echo 226 > /sys/class/gpio/export**
   - GPIO to enable/disable pullup resistor
6) **$ echo 214 > /sys/class/gpio/export**
   - GPIO that controls the TRI_STATE_ALL signal, which is used to connect/disconnect the shield pins.

3. **$ echo low > /sys/class/gpio/gpio214/direction**
   - Disconnect the shield pins before making changes.
4. **$ echo high > /sys/class/gpio/gpio263/direction**
   - Set GPIO 263 as high to select GPIO/SPI.
5. **$ echo low > /sys/class/gpio/gpio240/direction**
   - Set GPIO 240 as low to select GPIO.
6. **$ echo mode0 > /sys/kernel/debug/gpio_debug/gpio41/current_pinmux**
   - According to the table on the next page, mode 0 will select GPIO.
7. **$ echo low > /sys/class/gpio/gpio258/direction**
   - Set the pin direction as input.
8. **$ echo in > /sys/class/gpio/gpio226/direction**
   - Disable pull-up resistor
9. **$ echo in > /sys/class/gpio/gpio41/direction**
   - Set the pin direction as input.
10. **$ echo high > /sys/class/gpio/gpio214/direction**
    - Connect the shield pins.
11. You can now read digital input on pin 10 by entering "**cat /sys/class/gpio/gpio41/value**".
12. When the button is not pressed and the command in step 11 is entered, the output on the display is 0.
13. Enter the same command while the button is pressed. You should get 1.

Optional Practice

1. Repeat this demonstration for other pins with muxed functions (e.g. IO11).
2. Write a C code program that reads a button sensor or any other digital sensor without using the MRAA library or making a system call to run the commands presented above.



Shield pin	GPIO pin mux					SoC pin modes				
	Pin	Linux	0 (low)	1 (high)	Power-on default	Pin	Linux	0	1	2
IO0	-					GP130	130	GPIO	UART	
IO1	-					GP131	131	GPIO	UART	
IO2	-					GP128	128	GPIO	UART	
IO3	-					GP12	12	GPIO	PWM	
IO4	-					GP129	129	GPIO	UART	
IO5	-					GP13	13	GPIO	PWM	
IO6	-					GP182	182	GPIO	PWM	
IO7	-					GP48	48	GPIO		
IO8	-					GP49	49	GPIO		
IO9	-					GP183	183	GPIO	PWM	
IO10	U34_IO1.7	263	**PWM4_OUT**	GP41	Pulled down input	GP41	41	GPIO	I2S	
				SSP5_FS_1						
	U16_IO1.0	240	GP41	**SSP5_FS_1**	Pulled up input [1]	GP111	111	GPIO	SPI	
IO11	U34_IO1.6	262	**PWM5_OUT**	GP43	Pulled down input	GP43	43	GPIO	I2S	
				SSP5_TXD		GP115	115	GPIO	SPI	
	U16_IO1.1	241	GP43	**SSP5_TXD**	Pulled up input [1]					
IO12	U16_IO1.2	242	GP42	**SSP5_RXD**	Pulled up input [1]	GP42	42	GPIO	I2S	
						GP114	114	GPIO	SPI	
IO13	U16_IO1.3	243	GP40	**SSP5_CLK**	Pulled up input [1]	GP40	40	GPIO	I2S	
						GP109	109	GPIO	SPI	
IO14	U17_IO0.0	200	GP44	**A0**	Pulled up input [1]	GP44	44	GPIO		
IO15	U17_IO0.1	201	GP45	**A1**	Pulled up input [1]	GP45	45	GPIO		
IO16	U17_IO0.2	202	GP46	**A2**	Pulled up input [1]	GP46	46	GPIO		
IO17	U17_IO0.3	203	GP47	**A3**	Pulled up input [1]	GP47	47	GPIO		
IO18	U17_IO0.4	204	GP14	**A4**	Pulled up input [1]	GP14	14	GPIO	I2C-6	I2C-8
			I2C6_SCL			GP28	28	GPIO		
IO19	U17_IO0.5	205	GP165	**A5**	Pulled up input [1]	GP165	165	GPIO	I2C-6	I2C-8
			I2C6_SDA			GP27	27	GPIO		

1. *These pins are pulled up inputs at power-on. This effectively enables the mux switches (i.e. mux function 1 is selected).*

Figure 17 Pin Function Multiplexing Control (Source: Intel)

References

1. https://www.arduino.cc/en/Reference/SPI
2. http://iotdk.intel.com/docs/master/mraa/spi_8h.html
3. https://github.com/intel-iot-devkit/mraa/blob/master/docs/edison.md
4. https://www.arduino.cc/en/Tutorial/SPIEEPROM
5. http://download.intel.com/support/edison/sb/edisonarduino_hg_331191007.pdf
6. http://www.atmel.com/Images/doc2585.pdf
7. https://software.intel.com/sites/default/files/managed/ed/ec/ICS_Using_MRAA_WhitePaper.pdf
8. https://www.arduino.cc/en/Tutorial/PWM