

IoT Platform Tutorial 8: More Interfaces

In Hwan “Chris” Baek
chris.inhwan.baek@gmail.com

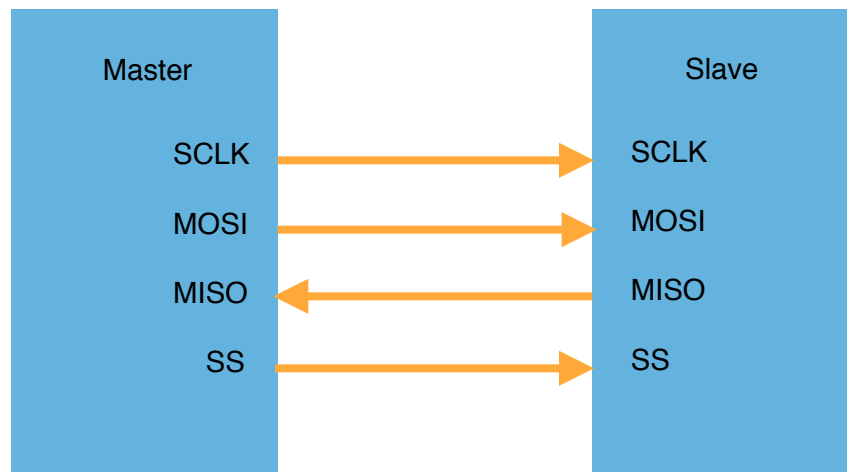
Introduction

Tutorial 4 introduced the basics of I/O interfaces on Intel Edison. This tutorial will guide you through even more interfaces and what is beneath the abstractions of I/O programming. In this tutorial, you will learn to:

1. Establish an SPI communication between the Intel Edison and the Arduino Uno,
2. Control a servo using the Intel Edison’s PWM,
3. Access the GPIO without MRAA, and
4. Configure the shield pins.

SPI

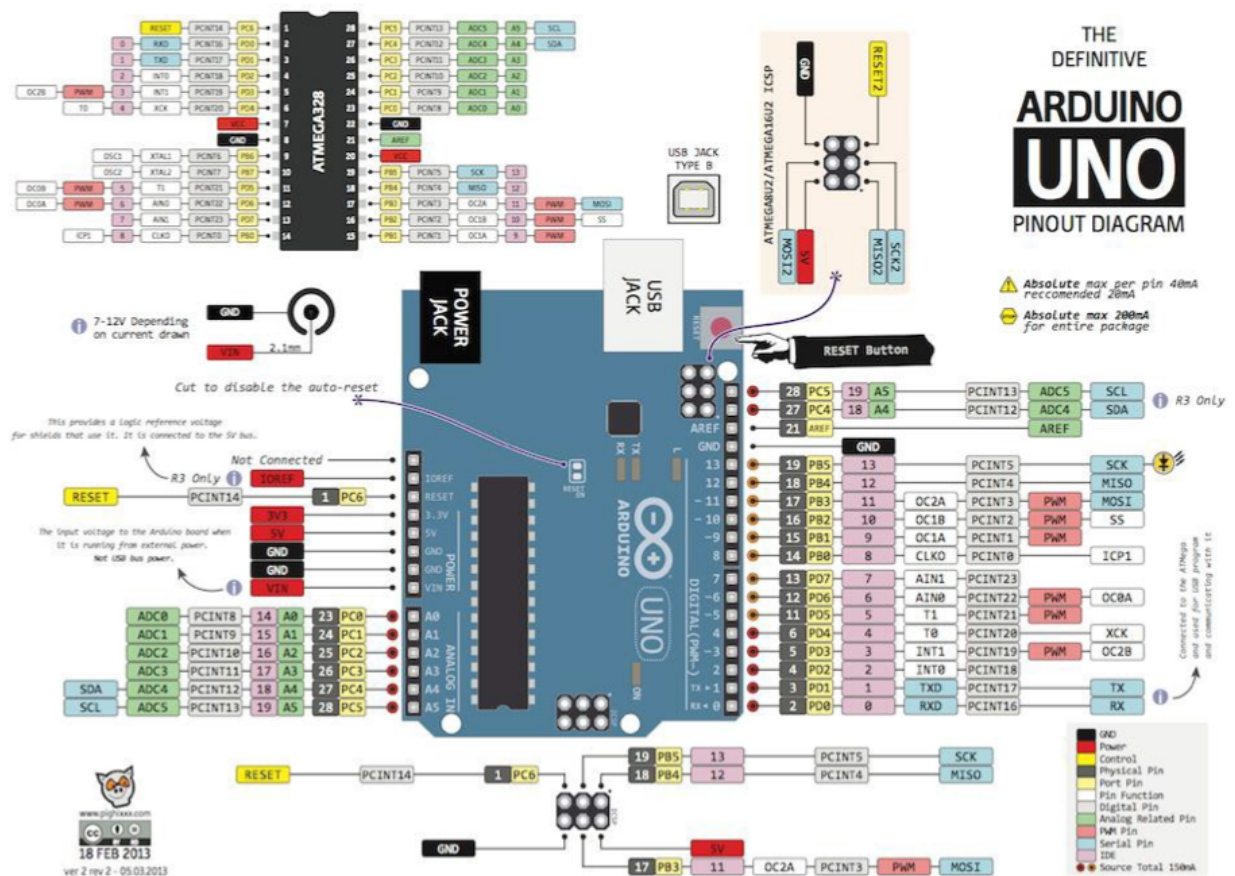
The Serial Peripheral Interface (SPI) bus is a synchronous serial communication between one master device and one or more slave devices for a short distance. Along with I2C, SPI is primarily used in embedded systems. The motivation of the development of these buses is the small number of wires. A common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



SPI between a single master and a single slave

SCLK (serial clock) is the clock line. The master device send the clock signal to its slave devices through SCLK and SCLK synchronizes the data communication. MOSI (master out, slave in) and MISO (master in, slave out) are the data lines. There are two data lines so that SPI has full duplex communication (i.e. simultaneous communication in both directions). Every clock

In this tutorial, we will implement SPI in mode 0 (CPHA = 0 and CPOL = 0) between an Intel Edison and an Arduino Uno. The Edison's Arduino-compatible breakout and the Arduino Uno have the same pinout shown as below.



SCLK, MISO, MOSI, and SS are Pin 13, Pin 12, Pin 11, and Pin 10 respectively. 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. Follow the steps on the next page.

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. SSH into the Edison.
3. **\$ mkdir tutorial8_examples**
4. **\$ cd tutorial8_examples**
5. **\$ vi spi.c**
6. Type the following C code.

```
#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;
}
```

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.

```

#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;
}

```

12. Open the serial monitor.
13. Go back to the SSH session and run the “**spi**” program by entering “**./spi**”
14. 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.

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 you may have not been so familiar with. "**SPCR**" refers to SPI Control Register. A register is a 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

SPIE (SPI Interrupt Enable): when 1, enable SPI interrupt.

SPE (SPI Enable): 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 **P**olarity as 0 or 1.

CPHA: Setting the **C**lock **P**hase 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.



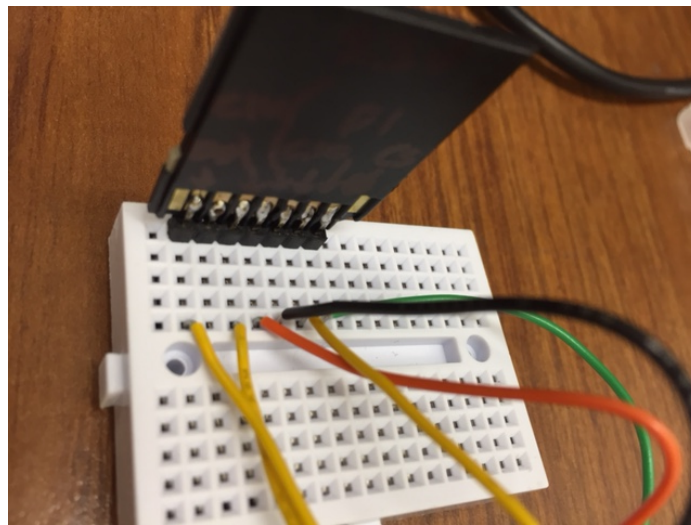
In the Arduino sketch, `while (!(SPSR & (1 << SPIF))) { }` loop will make sure all eight bits are received. When the transmission of data is completed, SPI Interrupt Flag (SPIF) is set to 1. SPIF is a bit in SPI Status Register (SPSR).

Optional Practice

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

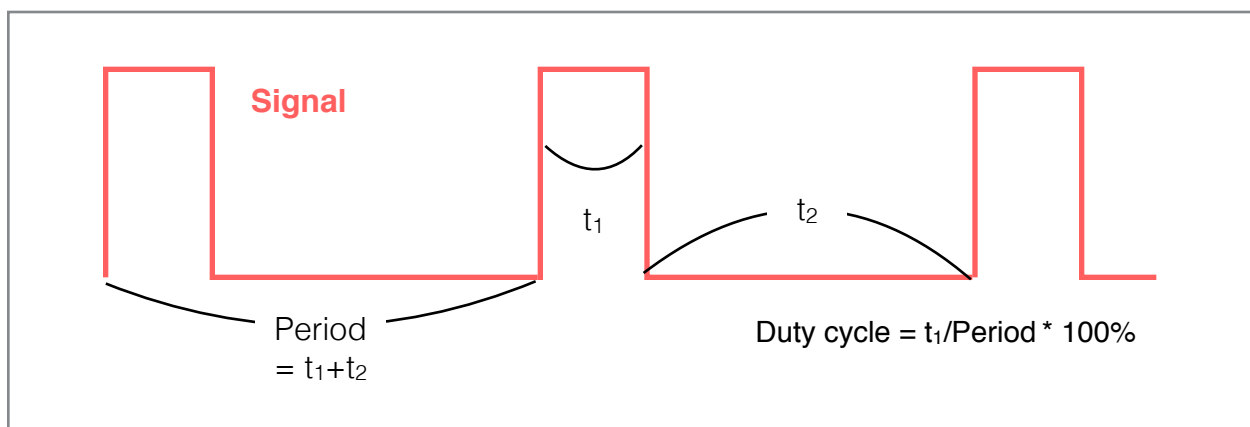
Hint:

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.



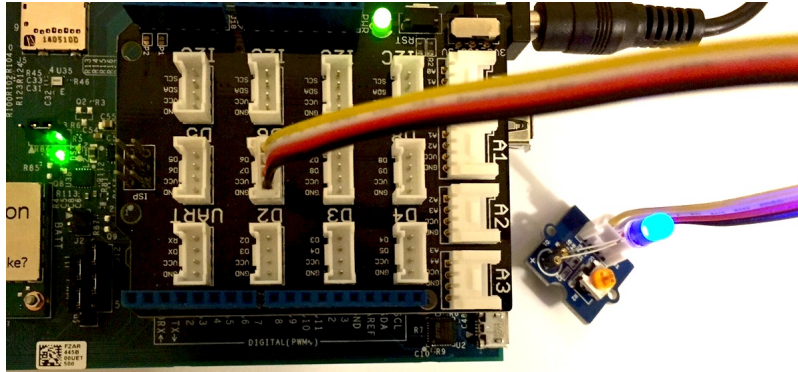
PWM

Pulse-width modulation (PWM) is a type of digital output, which is composed of the period and the duty cycle [7]. The period is the length of time between one pulse and the following one. The duty cycle is the percentage of the signal in digital high value. These are illustrated below.

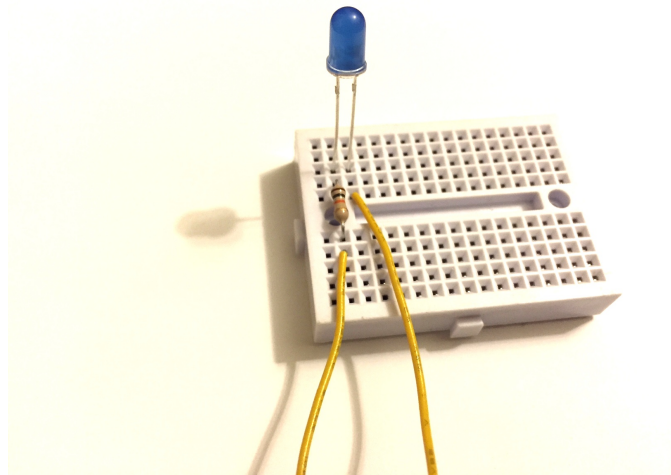


Before we jump into the details, let's try a simple PWM demonstration. In this demonstration, we will control the brightness of a LED. Follow the steps below.

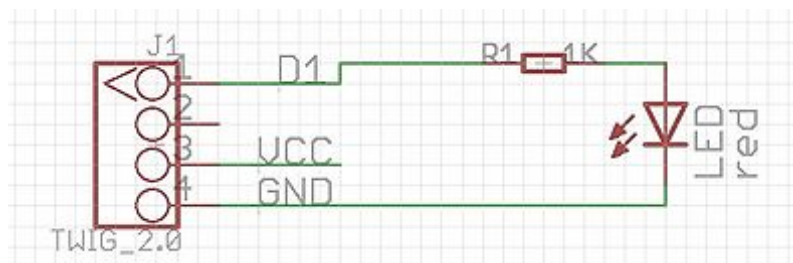
1. Insert the Grove Base shield into the Edison.
2. Connect a LED to D6 (pin 6) using the LED socket, which has a current-limiting resistor to protect the LED from high current. (Note: digital pins with “~” before the number are available for PWM)



3. Alternatively, you may use a breadboard, 1k ohm resistor, and wires as shown in the picture below. (**Caution**: wrong configuration may damage devices)



Follow the schematic of the Grove LED socket.



(Image Source: Seeed Studio)

4. Power the Edison with both the power adapter and micro USB cable via multi-gadget port.
5. SSH into the Edison.
6. navigate to /home/root/tutorial8_examples.
7. **\$ vi pwm_led.c**
8. 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 initialized.\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;
}
```

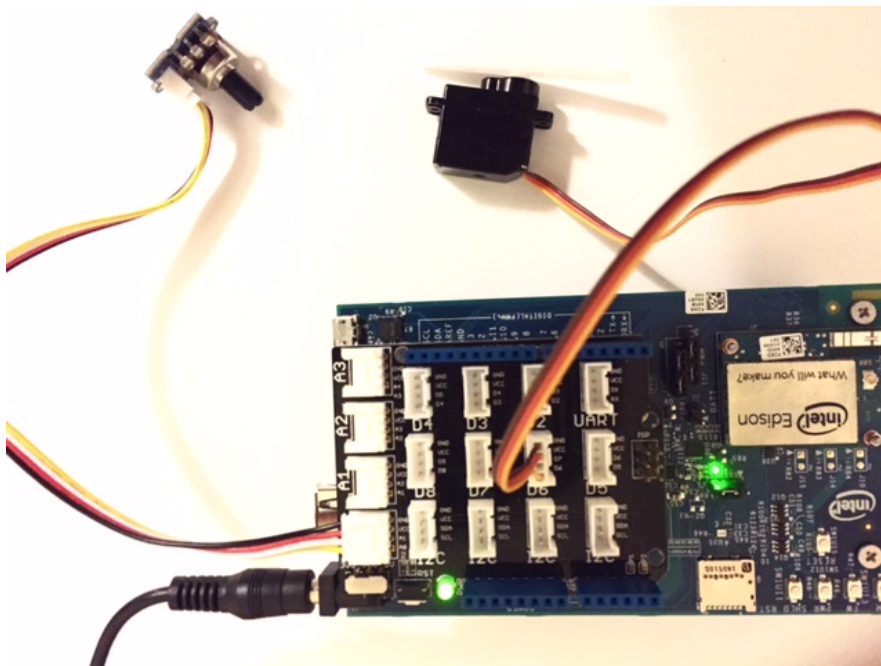
9. **\$ gcc -lmraa -o pwm_led pwm_led.c**
10. Wrong use of PWM may damage devices. The code does not disable PWM. Thus, just to be safe, keep in mind that you must completely shutdown the Edison to disable PWM after running the pwm_led program.
11. **\$/pwm_led**

By modulating the pulsing signals, we can get analog results with digital means [8]. Let's assume that the Edison's digital output is configured to operate at 5V. Medium brightness on the LED can be achieved by supplying $\sim 2.5V$ to the LED. However, the output can only be either 0V or 5V on a digital pin. Instead of supplying 2.5V, we can use PWM to switch the output voltage of a digital pin between 0V and 5V very rapidly. The brightness is determined by duty cycle (how long the LED is on for a given period). Now, look at `pwm_led.c` code 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, so the brightness may not be controlled with PWM. Another device with very fast response is a servo. Therefore, we can use PWM to control a servo. The next demonstration is a PWM-controlled servo.

A servo's position is determined by the pulse width (not the duty cycle). For most servo, 0.5ms pulse width results in the full left position and 2.5ms results in the full right position. We can use PWM to control a servo by changing the pulse width. A typical value for the period is $\sim 20ms$. Follow the steps below to implement a servo control system.

1. To ensure that PWM is not enabled, reboot the Edison.
2. Insert the Grove Base shield into the Edison.
3. As shown in the picture below, connect the servo to D6 and the rotary angle sensor to A0.



4. If the Edison reboots at the moment when the servo is connected, not enough power is supplied to the Edison. Seek for advice if you have this problem.
5. SSH into the Edison.
6. navigate to `/home/root/tutorial8_examples`.
7. `$ vi pwm_servo.c`

8. Type the following code.

```
#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);
        if (value >= 1.0f) {
            value = 0.0f;
        }
    }
    mraa_pwm_write(pwm, 0.025f);
    return 0;
}
```

9. **\$ gcc -lmraa -o pwm_servo pwm_servo.c**
10. Like the previous demonstration, keep in mind to shutdown the board to disable PWM after running the program.
11. **\$./pwm_servo**
12. Turn the rotary angle sensor to control the servo.

GPIO Revisited

In this section, we will access GPIOs without the MRAA library. The embedded Linux provides user space interfaces for GPIOs. 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. SSH into the Edison.
4. **\$ echo 48 > /sys/class/gpio/export**
5. Ignore “*Device or resource busy*” error message if you get it. This message means that the gpio is already exported. Enter “**echo 48 > /sys/class/gpio/unexport**”, then run the command in step 4 again.
6. **\$ echo 255 > /sys/class/gpio/export**
7. Ignore “*Device or resource busy*” error message if you get it. This message means that the gpio is already exported. Enter “**echo 255 > /sys/class/gpio/unexport**”, then run the command in step 4 again.
8. **\$ echo 223 > /sys/class/gpio/export**
9. Ignore “*Device or resource busy*” error message if you get it. This message means that the gpio is already exported. Enter “**echo 223 > /sys/class/gpio/unexport**”, then run the command in step 4 again.
10. **\$ echo high > /sys/class/gpio/gpio255/direction**
11. **\$ echo in > /sys/class/gpio/gpio223/direction**
12. **\$ echo out > /sys/class/gpio/gpio48/direction**
13. **\$ echo 1 > /sys/class/gpio/gpio48/value**
Now the LED should turn on.
14. **\$ 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 5 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 the picture 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. The table 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.

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 page 13 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 pullup/pulldown resistors are out of this tutorial’s scope.

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**”.

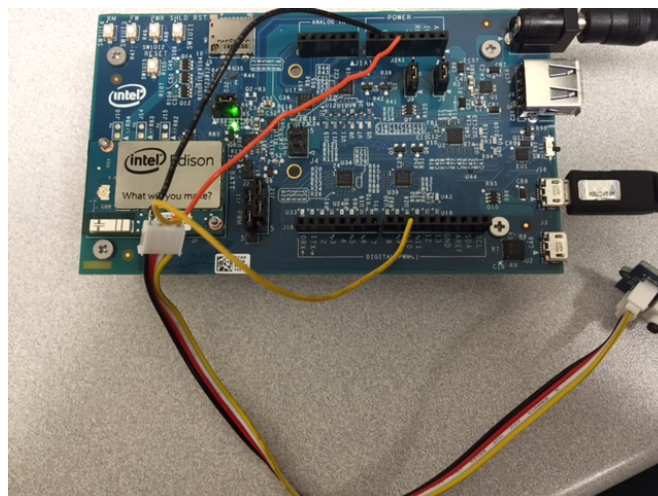
Optional practice

1. Repeat this demonstration for IO8.
2. Write a C program that blink 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 sections is an extension to the previous section. The table on page 5 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 Edison as shown below.
 - yellow - Pin 10
 - red - VCC (5V)
 - black - GND



2. Export GPIOs.
 - i. **\$ echo 41 > /sys/class/gpio/export**
 - SoC pin number for Linux
 - ii. **\$ echo 263 > /sys/class/gpio/export**
 - GPIO for multiplexing control.
 - Let's look at the block diagram on page 13. 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 the next page 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).
 - iii. **\$ echo 240 > /sys/class/gpio/export**
 - GPIO for multiplexing control.
 - iv. **\$ echo 258 > /sys/class/gpio/export**
 - GPIO for pin direction
 - v. **\$ echo 226 > /sys/class/gpio/export**
 - GPIO to enable/disable pullup resistor
 - vi. **\$ 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 pullup 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).

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>