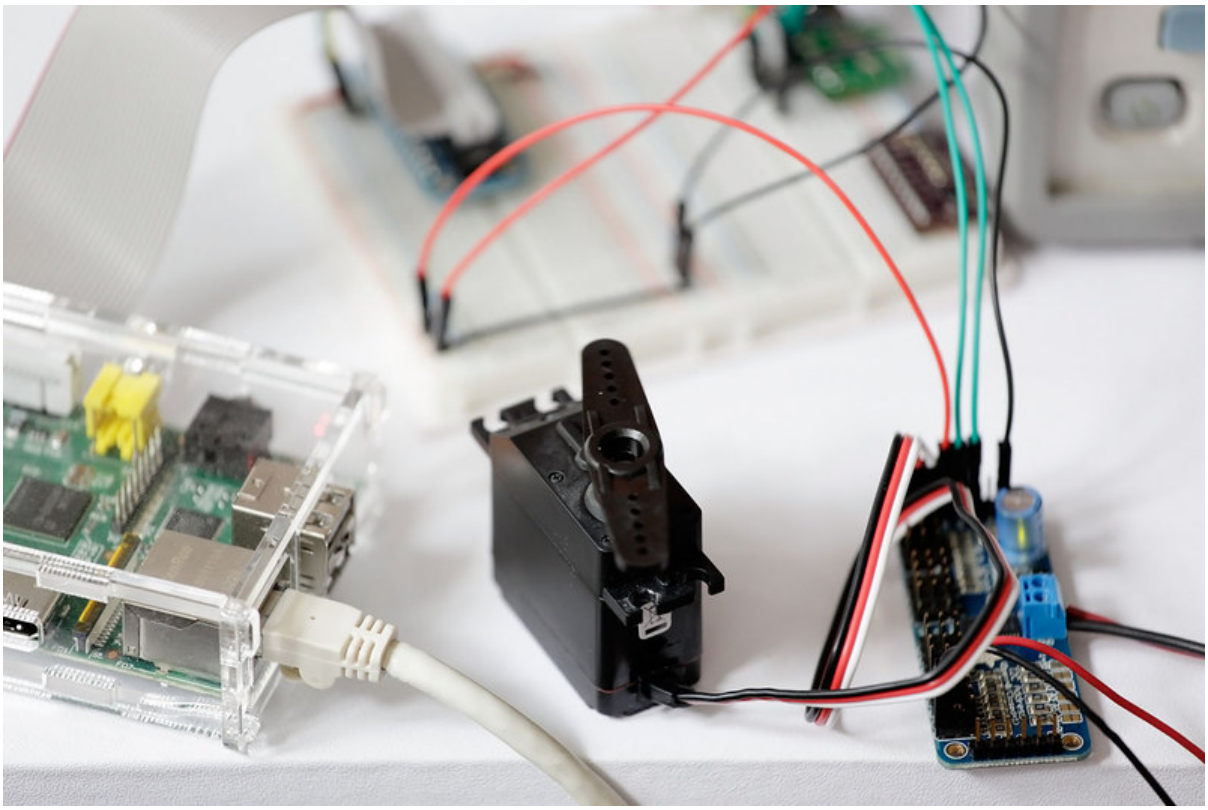
Adafruit 16 Channel Servo Driver with Raspberry Pi

Created by Kevin Townsend

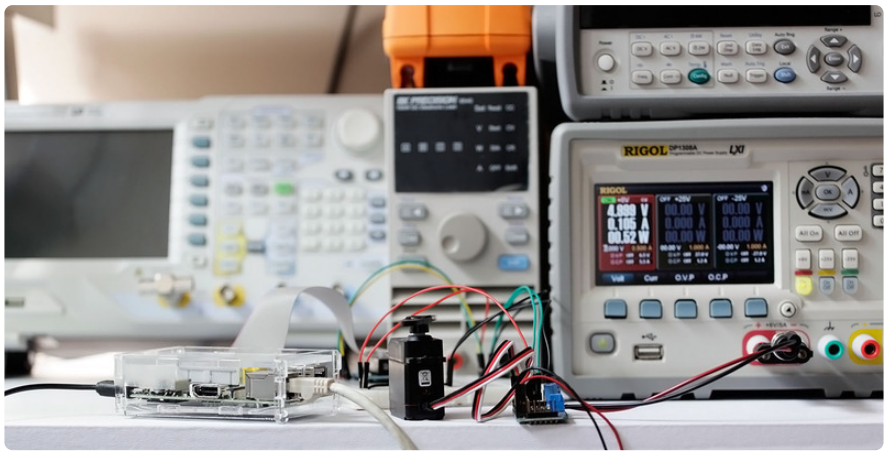


https://learn.adafruit.com/adafruit-16-channel-servo-driver-with-raspberry-pi

Last updated on 2024-03-08 01:44:00 PM EST

Table of Contents

Overview



Servo motors are often driven using the PWM outputs available on most embedded MCUs. But while the Pi does have native HW support for PWM, there is only one PWM channel available to users at GPIO18. That kind of limits your options if you need to drive more than one servo or if you also want to dim an LED or do some sort of other PWM goodness as well. Thankfully ... the PI **does** have HW I2C available, which we can use to communicate with a PWM driver like the PCA9685, used on Adafruit's 16-channel 12-bit PWM/Servo Driver!

Using this breakout, you can easily drive up to 16 servo motors on your Raspberry Pi using our painless Python library and this tutorial.

Note this cannot be used for driving anything other than analog (1-2 millisecond pulse drive) servos. DC motors, AC motors and 100% digital servos are not going to work. (Note that most 'digital servos' still use the analog pulse interface and are suitable for use with this controller.)

This board can also be used to control 16 PWMs in general for LED lighting and such but we're focusing on Servos in this tutorial!

What you'll need

We'll be using the following items in this tutorial:

- Adafruit 16-Channel 12-bit PWM/Servo Driver (http://adafru.it/815)
- Servo Motor: Standard Servo (http://adafru.it/155) or Continuous Rotation Servo (http://adafru.it/154)
- Female 2.1mm DC Power Adapter (http://adafru.it/368)

- 5V 2A Power Supply (http://adafru.it/276)

Configuring Your Pi for I2C

Before you can get started with I2C on the Pi, you'll need to run through a couple quick steps from the console.

If you are running Rasbian and are familiar with Terminal commands, then the description below will be sufficient.

If not, then to learn more about how to setup I2C with Raspbian, then take a minor diversion to this Adafruit Tutorial: http://learn.adafruit.com/adafruits-raspberry-pi-lesson-4-gpio-setup/configuring-i2c (https://adafru.it/aTI)

When you are ready to continue, enter the following commands to add SMBus support (which includes I2C) to Python:

```
sudo apt-get install python-smbus
sudo apt-get install i2c-tools
```

i2c-tools isn't strictly required, but it's a useful package since you can use it to scan for any I2C or SMBus devices connected to your board. If you know something is connected, but you don't know it's 7-bit I2C address, this library has a great little tool to help you find it. **python-smbus** is required, it adds the I2C support for python!

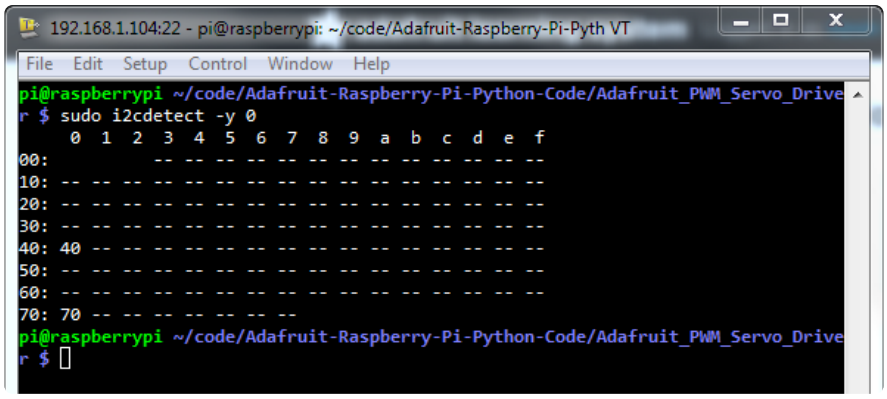
If you have an Original Raspberry Pi (Sold before October 2012) - the I2C is port 0:

```
sudo i2cdetect -y 0
```

If you have a second rev Raspberry Pi, the I2C is on port 1:

```
sudo i2cdetect -y 1
```

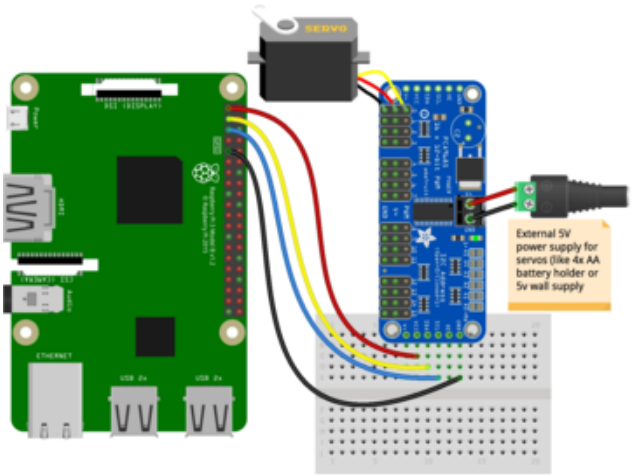
This will search /dev/i2c-0 or /dev/i2c-1 for all address, and if an Adafruit PWM breakout is properly connected and it's set to it's default address -- meaning none of the 6 address solder jumpers at the top of the board have been soldered shut -- it should show up at 0x40 (binary 1000000) as follows:



Once both of these packages have been installed, you have everything you need to get started accessing I2C and SMBus devices in Python.

Hooking it Up

The easiest way to hook the servo breakout up to your Pi is using a breadboard and connecting it to the Pi using I2C:



Pi 3V3 to **breakout VCC**
Pi GND to **breakout GND**
Pi SCL to **breakout SCL**
Pi SDA to **breakout SDA**
Servo orange wire to **breakout PWM on channel 0**
Servo red wire to **breakout V+ on channel 0**
Servo brown wire to **breakout Gnd on channel 0**
Check your servo's datasheet to verify which wires go to which pin!

Don't try to power your servos from the RasPi's 5V power, you can easily cause a power supply brown-out and mess up your Pi! Use a separate 5v 2A or 4A adapter

VCC = the digital supply for the IC (3.3V!), V+ = the supply for the servo motors (typically 5V). Be sure not to confuse the two or you may end up with burnt Pi!

The PCA9685 (the actual chip that drives the servos) is powered by the 3.3V supply on the Pi (labelled **VCC** on the servo breakout). Because the servos have different power requirements -- typically a 5V supply and as much as a couple hundred mA per

servo -- they're powered from a separate power supply, labelled **V+**.

In the example image above with a single servo motor, we are powering the motor from an external 5V power supply connected to the terminal block on the breakout board via a DC power adapter (http://adafru.it/368). Make sure you connect the wires correctly, with +/+ and GND/GND.

Why not use the +5V supply on the Raspberry Pi?

Switching directions on the servo can cause a lot of noise on the supply, and the servo(s) will cause the voltage to fluctuate significantly, which is a bad situation for the Pi. It's highly recommended to use an external 5V supply with servo motors to avoid problems caused by voltage drops on the Pi's 5V line.

When to add an optional Capacitor to the driver board

We have a spot on the PCB for soldering in an electrolytic capacitor. Based on your usage, you may or may not need a capacitor. If you are driving a lot of servos from a power supply that dips a lot when the servos move, **n** * **100uF** where **n** is the number of servos is a good place to start - eg **470uF** or more for 5 servos. Since its so dependent on servo current draw, the torque on each motor, and what power supply, there is no "one magic capacitor value" we can suggest which is why we don't include a capacitor in the kit.

Using the Adafruit Library

It's easy to control servos with the Adafruit 16-channel servo driver. There are multiple CircuitPython libraries available to work with the different features of this board including Adafruit CircuitPython PCA9685 (https://adafru.it/tZF), and Adafruit CircuitPython ServoKit (https://adafru.it/Dpu). These libraries make it easy to write Python code to control servo motors.

You can use this breakout with your Raspberry Pi and Python thanks to Adafruit_Blinka, our CircuitPython-for-Python compatibility library (https://adafru.it/BSN).

Python Installation of ServoKit Library

You'll need to install the Adafruit_Blinka library that provides the CircuitPython support in Python. This may also require enabling I2C on your platform and verifying you are running Python 3. Since each platform is a little different, and Linux changes

often, please visit the CircuitPython on Linux guide to get your computer ready (https://adafru.it/BSN)!

Once that's done, from your command line run the following command:

- `sudo pip3 install adafruit-circuitpython-servokit`

If your default Python is version 3 you may need to run 'pip' instead. Just make sure you aren't trying to use CircuitPython on Python 2.x, it isn't supported!

Controlling Servos

We've written a handy CircuitPython library for the various PWM/Servo boards called Adafruit CircuitPython ServoKit (https://adafru.it/Dpu) that handles all the complicated setup for you. All you need to do is import the appropriate class from the library, and then all the features of that class are available for use. We're going to show you how to import the `ServoKit` class and use it to control servo motors with the Adafruit 16-chanel servo driver breakout.

First you'll need to import and initialize the `ServoKit` class. You must specify the number of channels available on your board. The breakout has 16 channels, so when you create the class object, you will specify `16`.

```
from adafruit_servokit import ServoKit
kit = ServoKit(channels=16)
```

Now you're ready to control both standard and continuous rotation servos.

Standard Servos

To control a standard servo, you need to specify the channel the servo is connected to. You can then control movement by setting `angle` to the number of degrees.

For example to move the servo connected to channel `0` to `180` degrees:

```
kit.servo[0].angle = 180
```

To return the servo to `0` degrees:

```
kit.servo[0].angle = 0
```

With a standard servo, you specify the position as an angle. The angle will always be between 0 and the actuation range. The default is 180 degrees but your servo may have a smaller sweep. You can change the total angle by setting `actuation_range`.

For example, to set the actuation range to 160 degrees:

```
kit.servo[0].actuation_range = 160
```

Often the range an individual servo recognises varies a bit from other servos. If the servo didn't sweep the full expected range, then try adjusting the minimum and maximum pulse widths using `set_pulse_width_range(min_pulse, max_pulse)`.

To set the pulse width range to a minimum of 1000 and a maximum of 2000:

```
kit.servo[0].set_pulse_width_range(1000, 2000)
```

That's all there is to controlling standard servos with the PWM/Servo HAT or Bonnet, Python and `ServoKit`!

Continuous Rotation Servos

To control a continuous rotation servo, you must specify the channel the servo is on. Then you can control movement using `throttle`.

For example, to start the continuous rotation servo connected to channel `1` to full throttle forwards:

```
kit.continuous_servo[1].throttle = 1
```

To start the continuous rotation servo connected to channel `1` to full reverse throttle:

```
kit.continuous_servo[1].throttle = -1
```

To set half throttle, use a decimal:

```
kit.continuous_servo[1].throttle = 0.5
```

And, to stop continuous rotation servo movement set `throttle` to `0`:

```
kit.continuous_servo[1].throttle = 0
```

That's all there is to controlling continuous rotation servos with the PWM/Servo breakout, Python and `ServoKit`!

Python Docs

Python Docs (https://adafru.it/Dkx)

Library Reference

The driver consists of the following functions, which you can use to drive the underlying hardware when writing your own application in Python:

set_pqm_freq(freq)

Description

This function can be used to adjust the PWM frequency, which determines how many full 'pulses' per second are generated by the IC. Stated differently, the frequency determines how 'long' each pulse is in duration from start to finish, taking into account both the high and low segments of the pulse.
Frequency is important in PWM, since setting the frequency too high with a very small duty cycle can cause problems, since the 'rise time' of the signal (the time it takes to go from 0V to VCC) may be longer than the time the signal is active, and the PWM output will appear smoothed out and may not even reach VCC, potentially causing a number of problems.

Arguments

- **freq**: A number representing the frequency in Hz, between 40 and 1000

Example

The following code will set the PWM frequency to the maximum value of 1000Hz:

```
pwm.set_pwm_freq(1000)
```

set_pwm(channel, on, off)

Description

This function sets the start (on) and end (off) of the high segment of the PWM pulse on a specific channel. You specify the 'tick' value between 0..4095 when the signal will turn on, and when it will turn of. Channel indicates which of the 16 PWM outputs should be updated with the new values.

Arguments

- **channel**: The channel that should be updated with the new values (0..15)
- **on**: The tick (between 0..4095) when the signal should transition from low to high
- **off**:the tick (between 0..4095) when the signal should transition from high to low

Example

The following example will cause channel 15 to start low, go high around 25% into the pulse (tick 1024 out of 4096), transition back to low 75% into the pulse (tick 3072), and remain low for the last 25% of the pulse:

```
pwm.set_pwm(15, 1024, 3072)
```

If you need to calculate pulse-width in microseconds, you can do that by first figuring out how long each cycle is. That would be 1/**freq** where **freq** is the PWM frequency you set above. For 1000 Hz, that would be 1 millisecond. Then divide by 4096 to get the time per tick, eg 1 millisecond / 4096 = ~0.25 microseconds. If you want a pulse that is 10 microseconds long, divide the time by time-per-tick (10us / 0.25 us = 40) then turn on at tick 0 and turn off at tick 40.