adafruit learning system

Rotary Encoder in CircuitPython

Created by Kattni Rembor

Last updated on 2020-09-11 02:21:58 PM EDT

Overview

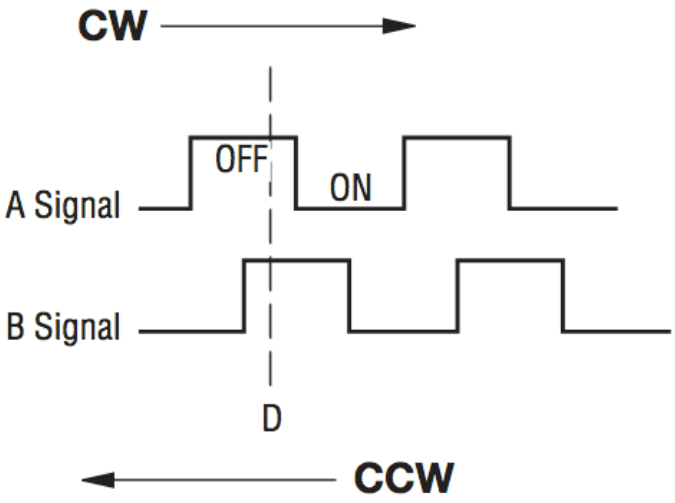
A rotary encoder is a *special* type of switch that converts the motion of the switch (clockwise or counterclockwise) into an output signal that can be used to determine what direction the knob is being rotated. **They're capable of continuous rotation - compare that to potentiometers that are only about 3/4 of a circle!**

Rotary encoders are not for situations where you need to know the exact position of the knob - for that, you'd want a potentiometer.

Rotary encoders are best in situations where you can use the change in position instead of the exact position. They're used in many applications such as the manual volume control or tuning on a car stereo.

This guide uses a mechanical, incremental rotary encoder. It has three pins: **A**, **C**, and **B**. **C** is the common ground for **A** and **B**. **A** and **B** are the signal pins. When you rotate the knob, A and B come into contact with the common ground pin, in a particular order depending on the direction you are rotating the knob. When each pin comes into contact with the common ground, they produce a signal. These signals are shifted out of phase with each other as one pin connects before the other pin. This is called quadrature encoding. You need to listen to those pins and the way they pulse in order to determine the direction and number of steps.

Quadrature Output Table



When rotating clockwise, the A pin connects first, followed by the B pin. When rotating counterclockwise, the B pin connects first, followed by the A pin. By tracking when each pin is connects to and disconnects from ground, we can use the signal changes determine which direction the knob is being rotated. The order in which they change is based on the direction the knob is being rotated. Tracking the signal changes allows us to use the rotary encoder as an incremental input.

Rotary encoders are really nice interface options but require special hardware or software support because of that high-speed pulsing.

In Arduino, you have to write code to manually track these signal changes and use that information to determine the direction of rotation and incremental changes. Luck for you, as of Circuit Python 3.0 we now have **rotaryio** support! Yay! It makes working with rotary encoders super easy!

This guide will show you how to use a rotary encoder with CircuitPython. We'll cover a basic example to get familiar with the rotary encoder. Then you'll learn how to use it as a volume control knob for your computer!

Hardware

To follow along with this guide, you'll need the following parts:

- **A rotary encoder** (https://adafru.it/Bzs)**.** This guide uses a 24-pulse mechanical, incremental rotary encoder with a push button switch. It fits onto a breadboard to make it easy to connect to a microcontroller board.
- **An ItsyBitsy M0 Express** (https://adafru.it/Bl9) or other CircuitPython capable board**.** You'll need a microcontroller board to run the code in this guide. This guide uses the ItsyBitsy M0 Express. You can use *any* CircuitPython Express board. Note that at this time, Gemma M0, Trinket M0, STM32, and other non-Express boards do not support `rotaryio`
- **A USB Micro/A data cable** (https://adafru.it/iia)**.** You'll need a known-good USB data cable! If your board isn't working, try a different cable. There are many USB cables that are only used for charging, and those will not work for this project.
- **A breadboard** (https://adafru.it/keP) **and jumper wires** (https://adafru.it/sd3)**.** You'll need these to connect up the rotary encoder and the ItsyBitsy M0 Express.

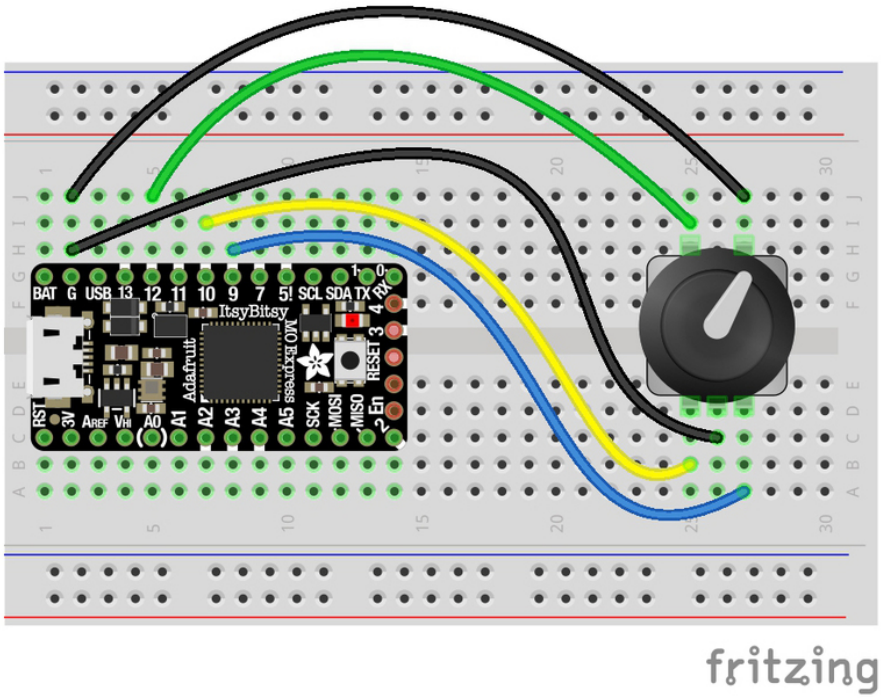
You can use any CircuitPython Express board. Note that at this time, Gemma M0, Trinket M0, the STM32, and other non-Express boards do not support rotaryio

Wiring

The rotary encoder has 5 pins total. Three are for the rotary encoder and two are for the button switch. The rotary encoder pins are **A**, **C** (common ground), and **B**, in that order when the rotary encoder is upright, i.e. oriented so the three pins are facing you.

We're going to be using both the rotary encoder and the push button switch for this guide. The following wiring diagram shows wiring up the rotary encoder to the ItsyBitsy to match the code example.

You can wire it however you like to your CircuitPython Express board - just remember to also update the pin #'s in the code!



- Connect one **button switch pin on the rotary encoder** to **ground on the ItsyBitsy (top black wire)**
- Connect one **button switch pin on the rotary encoder** to the **D12 pin on the ItsyBitsy (green wire)**
- Connect the **C (common ground) pin on the rotary encoder** to the **ground pin on the ItsyBitsy (middle black wire)**
- Connect the **A pin on the rotary encoder** to the **D10 pin on the ItsyBitsy (yellow wire)**
- Connect the **B pin on the rotary encoder** to the **D9 pin on the ItsyBitsy (blue wire)**

You can use up to 8 rotary encoders, and the two pulse pins (A and B) have to be on separate external interrupt pins. However, just about all the pins have different interrupts so just pick two pins and see if CircuitPython runs without throwing an exception!

CircuitPython

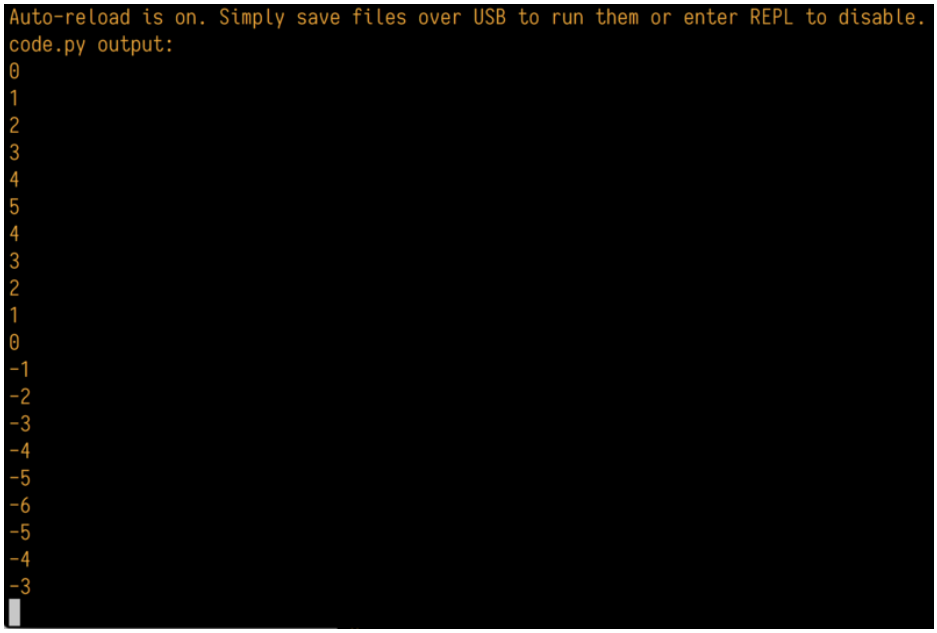
For this guide you must be running the latest CircuitPython. Download the latest release from the CircuitPython.org Downloads page (https://adafru.it/Em8). Follow these steps to install or update CircuitPython (https://adafru.it/Bzt).

To access the REPL/serial console, we assume you are using the Mu editor which is both a great Python editor and it has Adafruit support baked in to access the REPL via USB. You can follow this guide (https://adafru.it/Bid) to get Mu set up.

CircuitPython 4.x and later come with `rotaryio` built in. `rotaryio` makes it easy to use rotary encoders. Let's take a look at a simple example.

```
Temporarily unable to load content:
```

Open the serial console (REPL) and try turning your rotary encoder. Depending on the direction you turn it, you'll see the number go up or down.



Let's look at the code.

Library Imports

First we import the libraries we need `rotaryio` and `board`. These are both built into CircuitPython and do not require any external library files.

Create Encoder

Next we create the encoder to use in our code. We're using an incremental encoder, so we're going to set it up as such. It requires you to provide the pins that you've wired your rotary encoder to. We used D10 and D9. So, we assign `encoder = rotaryio.IncrementalEncoder(board.D10, board.D9)`.

We assign `last_position = None` so we can use this variable later to help identify when the rotary encoder is moving.

The Loop!

Inside our loop we begin by assigning `position = encoder.position`. This allows us to use `position` to identify the current position of the rotary encoder.

Then we have an `if` block. It starts with `if last_position is None or position != last_position:`. `!=` means not equal to. So, this says, if the variable `last_position` is equal to `None` or `position` is not equal to `last_position`, then do whatever is indented under this line. In this case, we have `print(position)` intended under it which `prints` the current position to the serial console.

Last we assign `last_position = position`. This is how we begin to track the current position of the rotary encoder.

Let's break it down. The first part of our `if` statement is true when `last_position` is equal to `None`. Remember, we assigned `last_position = None` before our loop, so when the loop runs for the first time, this is true. So the first time the loop runs, it will `print` the position of the rotary encoder to the serial console, and the first position is always `0`.

The second part of our if statement is true when `position` is not equal to `last_position`. Each time we run the loop, we assign `last_position = position`. If no changes to `position` have occurred, the two are the same and the second part of the `if` statement is false. So if you do not move the rotary encoder at all, the code will not print anything to the serial console. But, as soon as you do move it, `position` is no longer the same value, and `position` is no longer equal to `last_position` (since they're set to be equal). So, each time you move it, the second part of the `if` statement is true, and it prints the position to the serial console.

You can use this code to utilize the rotary encoder as an input. Let's take a look at an example.

Rotary Encoder Volume Knob and Play/Pause Button

This code allows the rotary encoder to control the volume on your computer, as well as play and pause music playing. It uses HID to emulate the media control keys on a keyboard. Be aware that once you save this code to the board, the knob will control volume and play/pause. Any other media controls will continue to work the same.

Copy the following code to your **code.py** file on your **CIRCUITPY** drive.

```
Temporarily unable to load content:
```

Now, while playing music or a video, try rotating the knob slowly to the right. The volume increases! Try turning it to the left. The volume decreases. Try pressing down on the knob to pause if it's playing and play music if it's paused!

Let's take a look at the code.

First we `import` all of the libraries we need for this program: `rotaryio`, `board`, `digitalio`, `ConsumerControl` from `adafruit_hid.consumer_control`, and `ConsumerControlCode` from `adafruit_hid.consumer_control_code`. The `adafruit_hid` library has many features, and since we're only using two, we specifically import the parts we need so we're not importing a bunch of features we don't need.

We setup the push button switch. We assign `button = digitalio.DigitalInOut(board.D12)`. We wired it up to D12, so that's the pin we provide. Then we set direction to `INPUT` and pull `UP`.

Next we create the rotary encoder object. We are using an incremental encoder, with the A pin wired to D10 and the B pin wired to D9. So, we assign `encoder = rotaryio.IncrementalEncoder(board.D10, board.D9)`.

Then we instantiate the HID object by assigning `cc = ConsumerControl()`. Now we can use `cc` to call the HID functions.

With basic code, when the button switch is pressed, it will spam the state change. That is great if you're trying to track the state or `print("Button pressed!")`. However, in our case, we're going to be using it to send a play/pause command. The play/pause command works by sending a single command to the player, which depending on the current state of the player, it will play or pause. If the player is currently playing, it will pause it. If it is currently paused, it will play. Therefore, if we were to use basic button code, it would spam playing and pausing every time you pressed the button. This won't work for our purposes. So, we're going to create a simple state machine (https://adafru.it/BlU) to see each button press as a single press, and allow us to use it for playing and pausing. To begin, we set `button_state = None` for use later with our state machine.

To change the volume, we're going to track the change in rotational position. If you rotate it three steps clockwise, it will increment the volume by three. If you rotate it three steps counterclockwise, it will decrement the volume by three. To track a change in position, you need to keep track of the last position and the current position. So, we assign `last_position = encoder.position` to create the initial position to track. This means wherever the rotary encoder is when the code first runs will be considered "0". So, if you don't like where the initial position is, you can move your rotary encoder to that position and reload the code.

Inside our loop, we find the current position by assigning `current_position = encoder.position`. Then, we find the change in position by subtracting `last_position` from `current_position`.

Next we have two code blocks that are quite similar. `if` and `elif` statements say, if the statement is valid, run the code indented underneath. `for _ in range(position_change):` takes the change in position we just calculated, and increments the volume by that amount with `cc.send(ConsumerControlCode.VOLUME_INCREMENT)`. So, `if` the `position_change` is greater than `0`, the volume will increase by `position_change` steps.

The second block of code is essentially the same, however, in this case, `if position_change < 0`, we take the negative `position_change` and decrement the volume by that amount.

Now that we've tracked the change and applied it to the volume, we set `last_position = current_position` so we can begin tracking again from that point.

Last, we have our simple state machine. We start by setting the the initial state. Remember, we assigned `button_state = None` before our loop. The first line checks to see `if` the button is pressed AND the `button_state is None`, and if both of these things are valid, sets `button_state = "pressed"`. Both of these things are valid on the first click because we've now changed `button_state`. If you press and hold the button, nothing will appear to happen because the only thing occurring on the button press is the change in `button_state`.

Next we check to see `if` the button is released AND `button_state` is equal to `"pressed"`. If both of these are valid, it `prints` `"Button pressed."` to the serial console, sends the play/pause command with `cc.send(ConsumerControlCode.PLAY_PAUSE)`, and then sets `button_state = None`.

Now we're back to the previous state and able to begin again - if you were to press the button now, the button is pressed and the `button_state is None` which is what's required for the first part of the state machine!