Quadrature Encoder Information & Potential CCL Implementation

Group 15 - Microchip

Introduction & Use Cases

A quadrature (AKA rotary) encoder is the basis for input of many devices and embedded systems. Most implementations of dials for selecting a certain option are created using a rotary encoder. It can also be used to determine the input position, speed, and direction of the rotational input, which has applications in creating self-stabilizing structures and other various user-centric systems.

In embedded systems, resources are generally quite limited; thus we must consider polling times for sensors and how that may impact our timing requirements and overall system. One example of where a rotary encoder might be used is in a self-stabilizing, inverted pendulum. This system uses stepper motors and a rotary encoder in order to determine the position of the pendulum, and move the motors appropriately to maintain the inverted state of the pendulum.

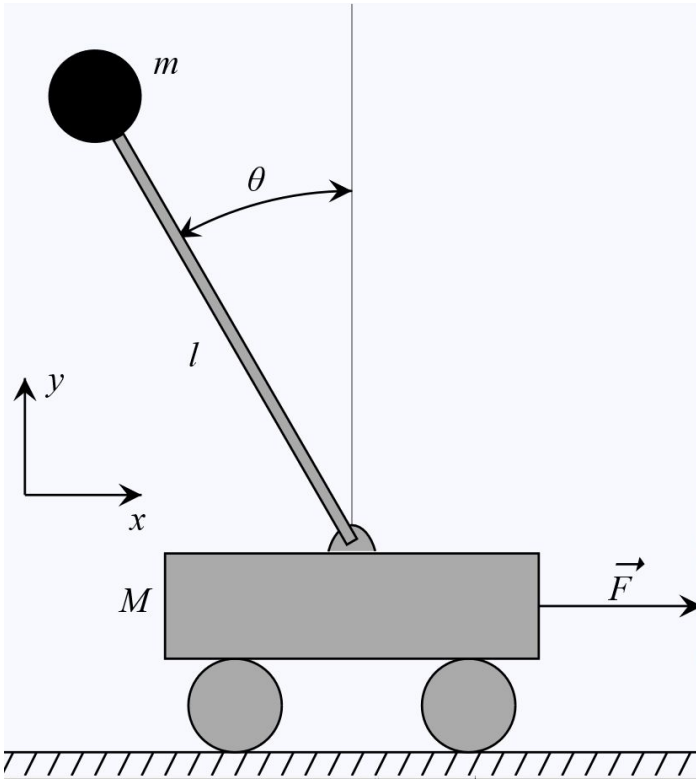


Figure 1: Free body diagram of an inverted pendulum. By exerting a force on the fulcrum of the pendulum, the angular displacement relative to the vertical axis can be changed / maintained [1].

As seen in Figure 1, the position of the pendulum has a certain displacement from the vertical axis. The direction and speed of the pendulum can be measured through using a rotary encoder connected to the pendulum at the fulcrum, and using those metrics the fulcrum can be moved to counteract the force of gravity. Although this system is fairly simple, it's an important demonstration of self-stabilizing structures, which are used in many more complex applications.

Fundamental Operation

At the most basic level, a rotary encoder requires four inputs / outputs to function. It requires a supply voltage and ground, and has two output lines used to determine the phase and direction of motion. Often times these basic rotary encoders will have a fifth output for a button signal, but that is beyond the scope of this document.

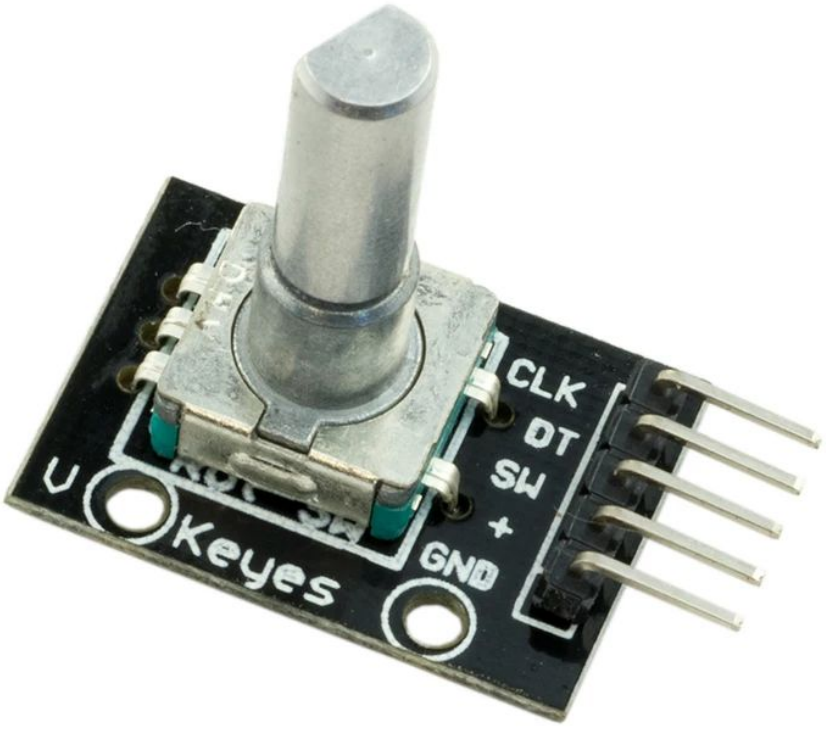


Figure 2: A low-cost rotary encoder with an integrated button [2].

The two signal lines (marked CLK and DT in Figure 2) are the lines for determining the phase and changing state of the encoder, and it's these lines that we'll be using

in combination with the CCL blocks on the ATmega4809 to decode this movement. There are four phases that the encoder can exist in, and depending on how these states change the direction of the turn can be determined. The speed of the motion can be determined by comparing how frequently the phase of the two signal lines changes relative to a reference clock on the microcontroller. Since the frequency of the reference clock is known, and the total number of phases on the encoder is fixed, the number of phase changes per unit of time can be calculated with relative ease.

Signal Generation

Each of the signal lines on a rotary encoder are used to determine the direction of rotation. This is possible because the phase changes between each signal line are offset from each other by 90 degrees, but are otherwise the same. With this simple rotary encoder, we have four possible phases / states that the encoder can exist in. Figure 3 demonstrates these phases; the two signal lines can both be low, both be high, or one is high and one is low. Depending on the transitions between these states, it is possible to determine the direction that the encoder is being rotated.

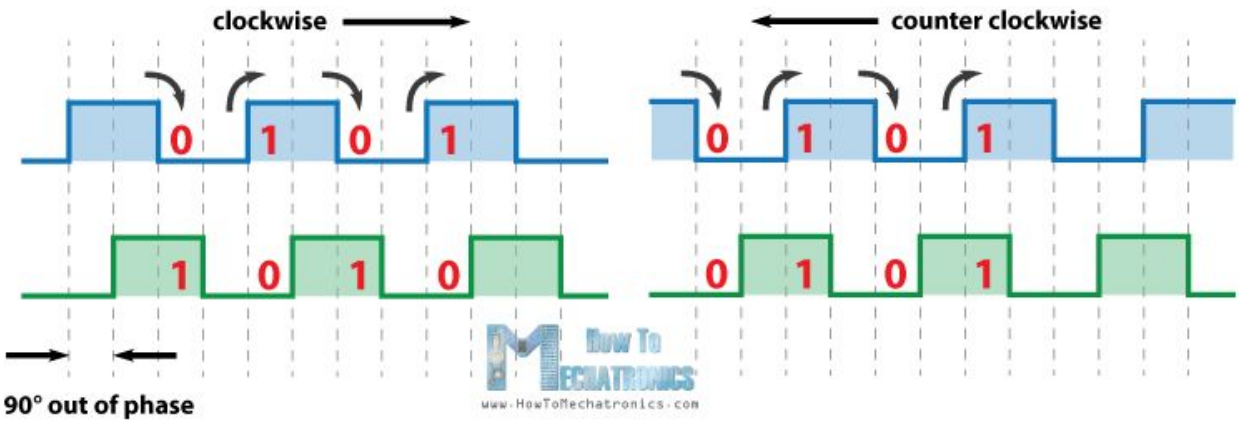


Figure 3: Basic phase diagram for a rotary encoder. The diagram on the left shows the encoder being rotated clockwise and the various phase changes can be observed. The right diagram has the encoder being rotated counter clockwise, and the phase changes are reversed from the ones on the left [3].

In this project, we'll be directly interfacing with the signal lines from a rotary encoder to the configured CCL inputs on the ATmega4809 in order to decode these inputs. By configuring one of the CCL blocks to read both the input lines, so long as the previous state is known, it is possible to give a binary output for the direction of motion without doing any operations on the main hardware thread. It should be possible to decode this input and provide the proper output while also doing some other time-sensitive operation on the core, and not have one operation impact the other. This provides us with what is essentially parallelism on an embedded system, which when utilized properly can be very powerful.

CIPs & CCL Blocks

The microcontroller being used for this project is the ATmega4809, which hosts many Core-Independent Peripherals (CIPs) that can perform operations in parallel with the main thread of execution. The CIPs that we are most interested in with this project are the Configurable Combinational Logic blocks, which when configured correctly will allow us to fully implement a quadrature decoder without having to do any computation on the core. This has many advantages, but the primary reason to use the CCL blocks to implement this decoder is to increase the responsiveness of the system. Considering that the microcontroller being used has a maximum clock frequency of 20 MHz, any time spent handling interrupts on the main core can have a significant impact on the time that a given result is generated, which for many time-sensitive applications will create other issues within the system.

Figure 4 is a suitable demonstration of how utilizing CIPs (and ultimately the CCL blocks in our project) can mitigate some of these response timing issues. The waveforms shown are two different implementations of the same functionality on a microcontroller which has CIPs built in. The blue line in each graph is being generated on an output pin by the core toggling that pin every 100 ns, which results in a square wave with a frequency of 10 MHz (approaching the maximum possible frequency for this microcontroller). The orange line remains low until a button is pressed, where on the falling edge of the press a 2 ms pulse is generated on a separate output pin. The primary difference between these two implementations is that the top graph is showing the output while using hardware interrupts to detect the button press, while the bottom graph is utilizing one of the included CIPs to both poll the button line and generate a 2 ms pulse upon detecting the falling edge of the button press.

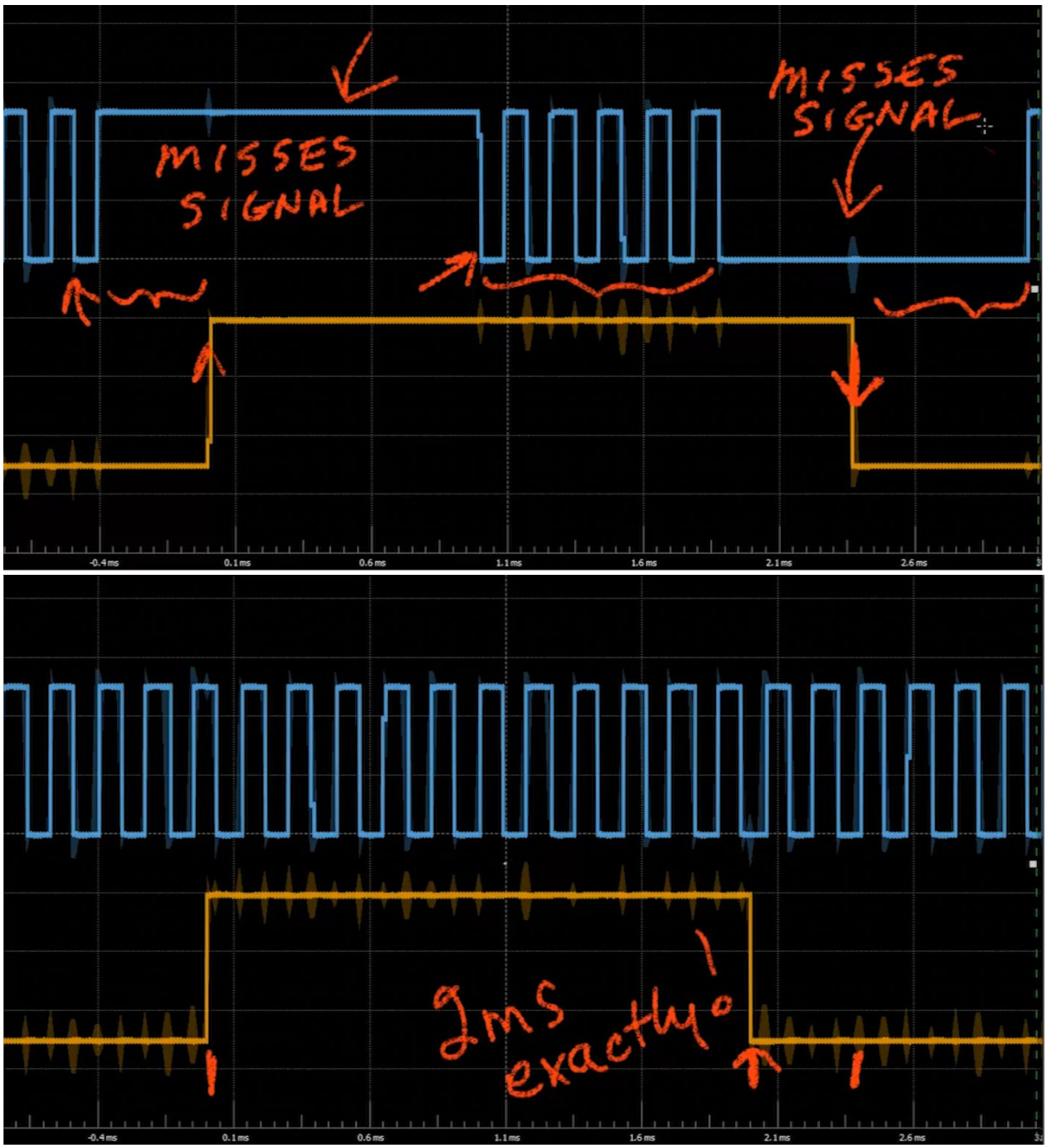


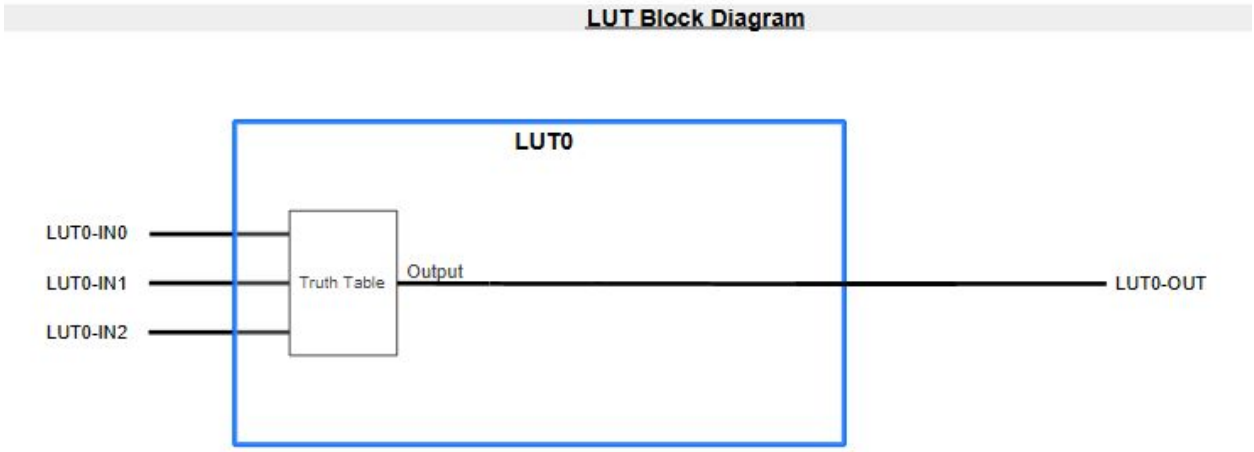
Figure 4: Demo of an AVR microcontroller utilizing one of the included CIPs to generate a 2 ms pulse upon detecting the falling edge of the button press. In both of the above waveforms, the blue line represents the voltage of a output pin on the microcontroller being handled by the core, and the orange line is the 2 ms output pulse generated as a result of the button press [4].

As is shown in Figure 4, the 10 MHz square wave is interrupted two separate times which results in the current state of the pin being held while the interrupt is handled. This happens because the interrupt for the button is being handled in software on the main execution thread. When the button is pressed, the core has to stop executing the toggle loop in order to enter the interrupt service routine that creates the 2 ms pulse. The second interrupt happens to set the pin low again after the 2 ms have elapsed, creating another delay in the square wave. A notable issue with the top implementation is that the 2 ms delay ends up becoming closer to 2.5 ms due to the interrupt handling. This is a 25% increase over the expected value, and anything that may rely on the pulse being exactly 2 ms will be affected significantly.

The second implementation's output is shown in the bottom waveform, and this implementation utilizes one of the microcontroller's peripherals to handle the button press and pulse generation. Implementing this functionality utilizing the CIPs allows for parallel operation, and fixes the issues seen in the previous example. Since the CIP is a separate piece of hardware from the core, when the button press is detected, there is no impact on the core; as such, the 10 MHz square wave is left unaffected. Furthermore, the pulse that was generated was actually 2 ms because the hardware handling this was fully being dedicated to that task and didn't have to do any extra interrupt handling which would otherwise delay the pulse.

Designing CCL Logic in MCC

MPLAB X is an IDE based on Netbeans designed specifically to be used in conjunction with both AVR and PIC microcontrollers. The MPLAB X Code Configurator (MCC) is a plug in for the IDE that will generate portions of code based on parameters defined through the GUI. This is primarily useful to this project for generating the code to utilize the CCL blocks for the quadrature decoder. Figure 5 depicts the default configuration for one of the four provided CCL blocks in the ATmega4809, where up to 3 inputs can be mapped to a single output through user-defined logic. This logic can be defined through a truth table, or through configurable logic gates, all of which can be designed in MCC's GUI.



Custom

IN2	IN1	IN0	OUT
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Figure 5: Block diagram showing the inputs and outputs for one of the four combinational logic blocks, as well as the customizable truth table which utilizes the three CCL inputs.

On top of the customizable logic, there are also options in the GUI to specify where the inputs are coming from (whether it be pin I/O, or another internal signal / reference), as well as the clock input and input filter mode. Figure 6 shows these options, which are available for each of the four CCL blocks.

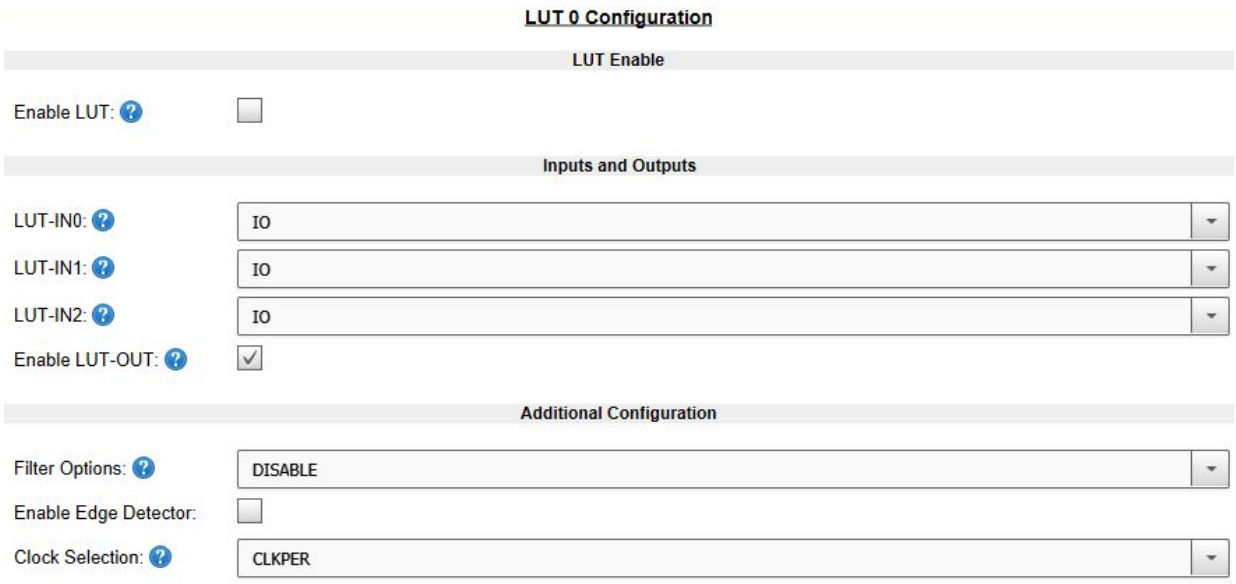


Figure 6: Configuration options for one of the 4 combinational logic blocks in the ATmega4809.

Detecting State Change Using CCL

Not only can the CCL blocks be used to design custom combinational logic, but they are also capable of detecting changes in state of a given line. As depicted in Figure 7, each CCL block is equipped to handle interrupts, which may involve detecting the rising edge, falling edge, or both on a given line. This will be the main piece of functionality from the CCL peripheral that allows us to decode the input from a rotary encoder. Since each rotation of a rotary encoder changes the phase of the two output lines, detecting how these lines change will give insight into what the direction of rotation is.

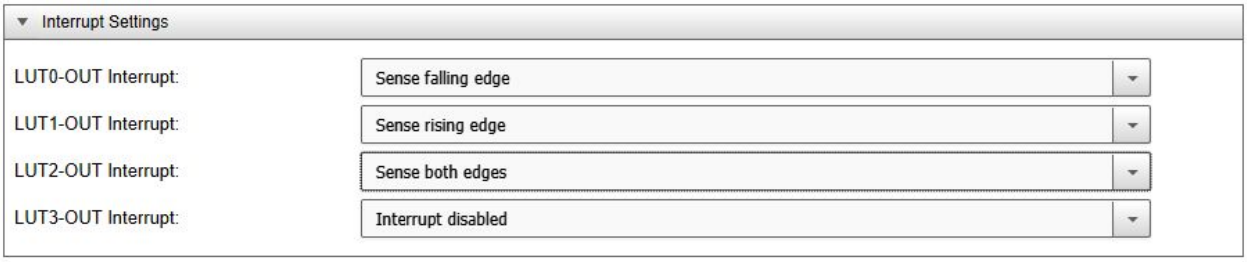


Figure 7: Options for detecting signal changes through the CCL peripheral, which is what makes implementing a quadrature decoder only using the CCL possible.

Not only does the interrupt functionality of the CCL allow us to determine the direction of rotation, but it also gives us the ability to measure the angular speed of the rotary encoder. This can be done by comparing the frequency of each phase change with a given internal clock on the ATmega4809. Measuring the number of detected phase changes against the reference clock means that we can determine how many times the phase changes occur per second. If we also know how many total positions the rotary encoder can exist in, then we can convert those phase changes per second to rotations per second, giving us the angular speed. This data is likely important to many projects that utilize a rotary encoder, and being able to generate this information without impacting the main execution thread is extremely useful.

References

1. https://en.wikipedia.org/wiki/Inverted_pendulum#/media/File:Cart-pendulum.svg
2. https://thepihut.com/products/keyes-rotary-encoder-module
3. https://howtomechatronics.com/tutorials/arduino/rotary-encoder-works-use-arduino/
4. https://www.youtube.com/watch?v=TcqpmupVCXQ