**Efficiently Reading Quadrature With Interrupts**

Once you start wanting to control the position of a motor through feedback control, you will likely end up using a rotary encoder as the feedback device. Encoders can be broadly categorized as:

* Absolute position
* Incremental position

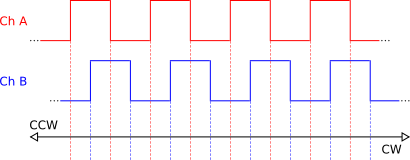
Absolute position encoders will tell you what the angular position is all the time, even between power cycles. Most use "grey code" (a modified form of binary) with several "tracks" (bits) to read position.

Incremental position encoders will tell you what the angular position is, but only relative to where it was when you started paying attention (usually on power-up). Two common types of incremental outputs are:

* Incremental (clever name)
* Quadrature

Incremental is rather useless for position control because it doesn't give you any information about what direction you are turning, just that you are turning. Quadrature encoders give you direction as well as incremental position. This article deals with efficiently reading quadrature output on an Arduino by utilizing external interrupts and some AVR bare-bones code.

**What is Quadrature?**

[](http://3.bp.blogspot.com/-866gNB4lUF8/U9w5qch1pXI/AAAAAAAAOvY/r6hN62tIz74/s1600/quadrature.png)

There are two channels of output in quadrature lovingly referred to as channels A and B. They are each a square wave, but are offset from each other by 90 degrees. Whether channel A is leading or lagging channel B depends on the direction the shaft is turning, which is what allows you to determine direction. For example, both channels are low and then channel A goes high, you know that you are spinning CCW. If channel B had instead gone high before channel A, you would then know you are spinning CW. Of course, this can be deduced starting from any state as can be seen in the diagram. The output channels can be produced by a variety of means, usually either magnets in a disk attached to the shaft and a pair of hall effect sensors, or a disk with slots cut out and a pair of optical sensors looking through the slots. A google image search of "quadrature encoder" will come up with plenty of pictures to explain it.   
  
How many magnets or slots are in a revolution of the encoder is known as pulses per revolution or p/r. Only the pulses on one channel are counted, the other channel will of necessity have the same number of pulses. Encoders can range anywhere from <10 p/r to 1000's of p/r, the higher numbers giving a higher resolution. If you are looking for maximum resolution, multiply the p/r by 4 since for each pulse there are two detectable events (rising edge and falling edge) on each channel. For example, a 1024 p/r encoder will give you 1024 pulses on channel A and 1024 pulses on channel B. If you detect both rising and falling edges on each channel, you have 4096 events in one revolution, giving an angular resolution of $$\frac{360^{\circ}}{1024 \tfrac{pulses}{rev} \times 4 \tfrac{events}{pulse}} = 0.0879^{\circ}/event$$

**Selecting an Encoder**

Selecting the best encoder for your design is more than just getting the highest resolution you can afford since the higher the p/r the faster you have to read the events. If you need to keep track of your position while spinning at 1,000 rpm with a 1024 p/r encoder, that means that between each event there is only  
  
$$\frac{1}{1024 \tfrac{pulses}{rev} \times 4 \tfrac{events}{pulse} \times \frac{1000 \tfrac{rev}{min}}{60 \tfrac{sec}{min}} } = 0.000014648 sec \approx 15\mu s$$  
You can see how a higher resolution than needed could result in not being able to keep up with the incoming information. Of course, if you needed to you could always just look for one edge on one channel and increase the 15 microseconds to 60 microseconds, but you loose resolution and direction when doing so.

From this example we can see some important factors that must be considered when selecting an appropriate encoder:

* What is your required angular resolution
* What is your maximum speed you need to be able to track
* How fast can you read and process the encoder output

The remainder of this article will deal with the third bullet point, allowing you to read the encoder output as fast as possible. This will give you more freedom when considering the other two points.

**Connect Quadrature Output to External Interrupts**

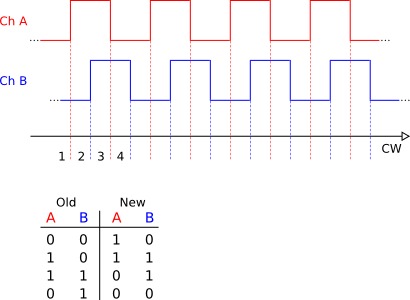
Too often I see people trying to connect the quadrature output to some digital inputs and read them in the program loop. The problem with this is that if your loop is slower than the incoming quadrature information, then you loose track of where you are. Wouldn't it be nice if there was a way to have the encoder tell us when it's ready to send information, rather than asking for updates every time around the loop?  
  
There is! It's called external interrupts, and an Arduino has two of them. Baically, you just connect the two quadrature output channels to the two interrups pins (digital pins 2 and 3 on a standard form factor arduino) and connect the interrupts in software:

1. volatile long enc\_count = 0;
3. void setup() {
4. *// all your normal setup code*
5. attachInterrupt(0,encoder\_isr,CHANGE);
6. attachInterrupt(1,encoder\_isr,CHANGE);
7. }

Some important things here. First, I've defined enc\_count as a global variable for future use. Next, an ISR (Interrupt Service Routine) is a bit of code that gets called whenever the interrupt is triggered. ISR's take no inputs and give no outputs (hence the need for a global enc\_count). You'll define this bit of code yourself, and I'll cover that a bit later. Both interrupts (interrupts 0 and 1 on pins 2 and 3, respectively) are calling the same ISR. In general the can call different ISR's, but for our purposes we want them both to behave the same. Next, we see the word "CHANGE". This is telling the program to interrupt every time there is a change on the pin, whether it goes from low to high, or high to low. Other options in place of "CHANGE" are "FALLING", "RISING", and "LOW". Check out <http://arduino.cc/en/Reference/AttachInterrupt> for additional info on using interrupts.  
  
So what happens is your loop is running happily along with no thought of checking the quadrature. Once one of the pins changes, the loop is interrupted, the ISR is executed, and once that's done the loop picks up where it left off. You must be careful to make the ISR as fast as possible, because your regular loop stops dead in it's tracks, and if you have any time sensitive stuff in the loop it may not work right. Also, you have to make sure the ISR ends before another one is called.

**The Look-up Table**

I wish I could claim credit for what I'm about to share with you. But the truth is, I can't. I found the concept in some dusty corner of the internet. At the time I didn't understand what was going on, but through copy and pasting code and some trial and error I was able to get it working for the project I was working on at the time. Since then I have figured out what was going on, and have expanded and generalized the concept a fair amount. I went to find the website I originally got this from, but was unable to locate it. If anybody has some idea of where it came from, please post in the comments.  
  
UPDATE: The source has been found! [http://www.circuitsathome.com/mcu/reading-rotary-encoder-on-arduino](http://www.circuitsathome.com/mcu/reading-rotary-encoder-on-arduino/)

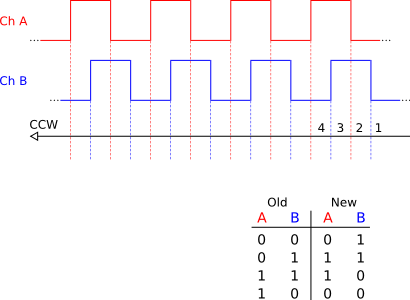
[](http://3.bp.blogspot.com/-z-JNB_pMrSg/U9w52Ly30bI/AAAAAAAAOvo/33mT_0UdOmA/s1600/quadrature_cw.png)

The idea here is simple enough: whenever there is a call to the ISR, the previous levels of the channels is stored in memory and the new levels are read. For convenience I will refer to the levels of the channels as xx where the first number is the level of channel A and the second number is the level of channel B. For example, progressing through states 1-4, the channel levels are written as:

1. 00
2. 10
3. 11
4. 01

Imagine we were in state 1 and both channels were reading low, and then the rising edge of channel A triggers an ISR and puts us into state 2. The old reading of "00" is remembered for future reference and the new levels "10" are read. Let's concatenate these two pieces of information into "0010" for convenience. Progressing from state 2 to state 3, we are then left with "1011", where "10" is the levels the channels were at in state 2, and "11" is the levels the channels are at in state 3. Continuing through the remaining state changes, we go through "1101" and "0100" to complete one period. There's nothing too tricky about any of this. We're just listing the 4 numbers in each row of the table in the image. If we were to continue to a state 5 and beyond, the strings of numbers would then repeat.

We can go through a similar exercise in the reverse direction and get the following:

[](http://3.bp.blogspot.com/-rQHd9oSDrWg/U9w51-OjZGI/AAAAAAAAOvk/9-dDVrgTofw/s1600/quadrature_ccw.png)

With a quick examination of the two tables we see that there are 8 unique strings of numbers, 4 describing clockwise rotation, and 4 describing counter-clockwise rotation. Now, all those rows look an awful lot like binary numbers, so let's make it official and call them binary. With 4 bits there are 16 unique numbers represented in binary (2^4 = 16). So I'm just going to fill out a table with all 16 numbers, and where ever there's a clockwise rotation I'll add a 1 in a new column, where ever there's a counter-clockwise rotation I'll add a -1, and everywhere else I'll add a 0.

[Obsah obrázku stůl

Popis byl vytvořen automaticky](http://3.bp.blogspot.com/-ndRzMPOSBZo/U9w53h6jUTI/AAAAAAAAOv8/Gg892hjSyc8/s1600/quadrature_table.png)

By now maybe you see where this is going. It's quite elegant, really. If you don't see it yet, don't worry. I'll continue.

Typically, what I see done by others when reading quadrature is to check a bunch of conditions and either add 1 or subtract 1 from the encoder variable. What if instead we had a predefined array 16 entries long, and the index of the array told us whether to add or subtract 1 from our encoder variable?

1. int8\_t lookup\_table[] = {0,-1,1,0,1,0,0,-1,-1,0,0,1,0,1,-1,0};

Seems simple enough, but how do we get which index we're looking for? If you're not too familiar with how a compiler takes the program you write and turns it into a program readable by a computer/microcontroller, it might surprise you to hear that it doesn't matter one bit if you enter a number in decimal or binary (or octal or hex), the end result will be the same:

lookup\_table[5] == lookup\_table[0b0101]

So now, if we find a way to store the previous and current channel levels as a binary number, we can use that information to index the look-up table.

**Binary Operators and PINs**

I'll skip straight to the punchline here. The ISR that gets called whenever there's an event on either of the channels looks like this:

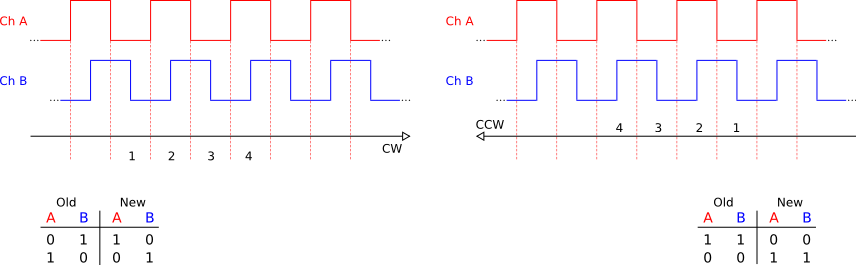
1. void encoder\_isr() {
2. static int8\_t lookup\_table[] = {0,-1,1,0,1,0,0,-1,-1,0,0,1,0,1,-1,0};
3. static uint8\_t enc\_val = 0;
5. enc\_val = enc\_val << 2;
6. enc\_val = enc\_val | ((PIND & 0b1100) >> 2)
8. enc\_count = enc\_count + lookup\_table[enc\_val & 0b1111];
9. }

This might look like a cryptic mess to you, but don't fret. It's not too difficult. Let's go through line by line.  
  
Line 8 is just the ISR function. Remember that ISR's take no inputs and give no outputs..  
  
Line 9 is the look-up table we worked out previously. The "static" in the front means that this is only set the first time the code is run, and it's remembered (and not reset) each subsequent time through the function.  
  
Line 10 is for holding the binary number derived from the quadrature channels. We'll use this to index into the look-up table.  
  
Line 12 is where we keep track of what the channels were last time around. the "<< 2" you see there says to shift the binary bits 2 places to the left. For example:  
  
00110011 << 2 = 11001100  
  
This is taking the 2 channel levels from last time and moving them left to match what's in our table images above. Whenever you shift bits, 0's will always fill in the new slots.  
  
Line 13 is a little more complicated. You see a ">> 2" in there, you can probably guess it means take whatever is in "(PIND & 0b1100)" and shift it 2 places to the right. "PIND" is a very handy, very fast way of reading digital pins on an Arduino. Whenever the code comes across the "PIND" word, it takes the input readings on pins 0-7 and gives them as an 8-bit binary number with each bit representing a single pin. If pins 2, 4 and 7 are high, and all the others low, PIND would return "10010100". There are other PINs on an Arduino, and similar ways of writing to the pins and setting the pull-up resistors. I'll refer you to <http://www.arduino.cc/en/Reference/PortManipulation> for further reading on the subject.  
  
The "& 0b1100" is what is sometimes referred to as a bit mask. When you use a single "&" like this (as opposed to the double "&&" in logical statements) it is a bit-wise AND, meaning each bit in the first number is ANDed with the corresponding bit in the second number to give an output. When reading the quadrature channels, we only care about the value on pins 2 and 3, but PIND is giving us 8 pins. So we apply a mask to ignore all the extraneous information:  
  
  10010100  PIND  
& 00001100  MASK  
----------  
  00000100  OUTPUT  
  
The output only gets a "1" when both PIND and the mask have a "1" in that location, just like you would expect from an AND operation.  
  
The last thing in line 6 is the "|". You're probably familiar with the OR logical operator "||". Just like the single "&", the single "|" works on a bit-wise level. So we are taking whatever is in enc\_val and ORing it with everything to the left of it. Let's say enc\_val contains the number 11001100, then:  
  
  11001100 | ((PIND & 1100) >> 2)  
= 11001100 | ((10010100 & 00001100) >> 2)  
= 11001100 | (00000100 >> 2)  
= 11001100 | 00000001  
  
So:  
  
  11001100  
| 00000001  
----------  
  11001101  
  
After all that, what's left in enc\_val is 11001101, where the "01" on the far right is the current levels of channels A and B, and the "11" just left of those is what the channel levels were the previous time through the ISR. the "1100" on the left is leftover information that isn't important to us any more.  
  
Line 15 should be pretty straight forward after all that. You can see that there's another bit mask going on:  
  
  enc\_val & 1111  
= 11001101 & 00001111  
  
So:  
  
  11001101  
& 00001111  
----------  
  00001101  
  
The binary number 00001101 is equal to the decimal number 13, and is the index into the look-up table we've been after all along. To finish off, enc\_count get's incremented by whatever value is in lookup\_table[13], which is 1. Pretty nifty, eh?

**Using Only 1 Interrupt**

If you're using an arduino Mega, or the new Due, you have more external interrupts to work with, so if you want to hook up multiple encoders or you have interrupts dedicated to other hardware you're probably covered. But if you're working on an Uno or one of it's predecessors, you are limited to only 2 interrupts.

It is possible to use this same look-up technique with only 1 interrupt per encoder. The trade off is that you will loose half of your resolution. In this case, you would hook up the other channel to a regular digital pin, and then rework your look-up table keeping in mind that you can only detect when one of the channels is changing.

[](http://3.bp.blogspot.com/-9WVs1ImbheQ/U9w53Hm10FI/AAAAAAAAOv4/H313k8qFmnw/s1600/quadrature_half_res.png)

You might think I'm missing some information on the B channel, but remember that the microcontroller only sees when A changes, and reads B at that time. In the CW direction, when state 2 started A was high and B was low. When it gets to state 3, A is low and B is high. We have no information about when B changed from low to high, only that it is now high. That's why we loose half the resolution when using only one interrupt.

Working out the entire look-up table:

[Obsah obrázku stůl

Popis byl vytvořen automaticky](http://4.bp.blogspot.com/-9vT8QVtvlug/U9w537vwMsI/AAAAAAAAOwE/fxmM6ao7DyI/s1600/quadrature_table_half_res.png)

If you instead connected channel B to the interrupt, this table would be different. One way to tell if you've done the table correctly is that the direction column should always be symmetric about the middle. That is, entry 1 should equal entry 16, entry 2 = entry 15, entry 3 = entry 14, etc. Meeting this condition doesn't guarantee that you've done it right, but if you don't meet this condition I guarantee you've done it wrong.

You will need to make some changes to the ISR as well. Line 6 in the ISR above assumed that the quadrature channels were hooked up the pins 2 and 3. For convenience in the code, I suggest you keep the two channels adjacent to each other whenever possible, i.e. pins 3 and 4 or pins 1 and 2. But remember the pins 0 and 1 are used for TX and RX.

If you connect channel A to the interrupt on pin 3, and channel B to pin 4, the ISR becomes:

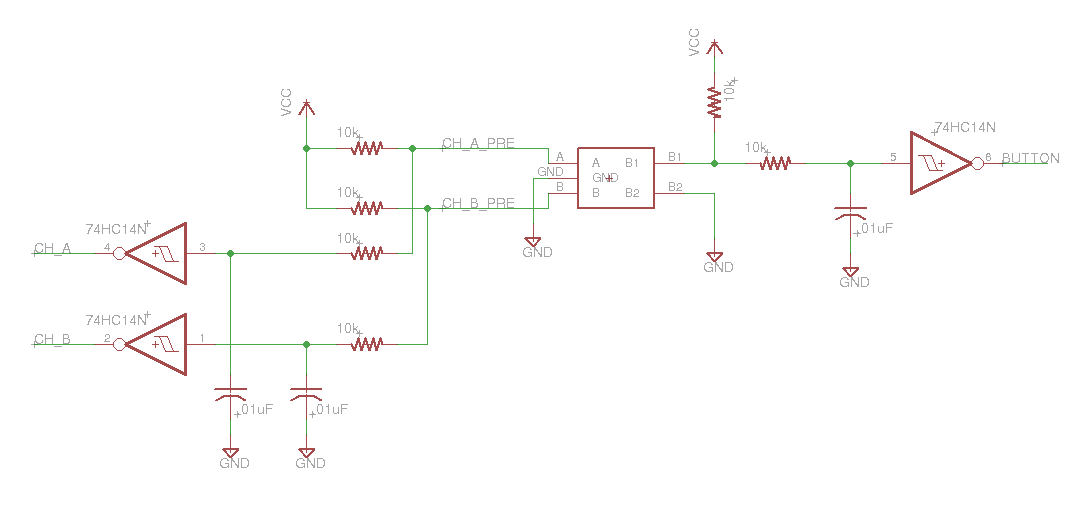
1. void encoder\_isr() {
2. static int8\_t lookup\_table[] = {0,0,0,-1,0,0,1,0,0,1,0,0,-1,0,0,0};
3. static uint8\_t enc\_val = 0;
5. enc\_val = enc\_val << 2;
6. enc\_val = enc\_val | ((PIND & 0b11000) >> 3)
8. enc\_count = enc\_count + lookup\_table[enc\_val & 0b1111];
9. }

**A Word on Debouncing**

Usually with optical or hall effect encoders, there is no bouncing of the channels that you need to worry about. But sometimes you want to use an encoder as an input knob, and these encoders usually have mechanical switches for the channels and are subject to bouncing. The proper way to handle bounce (called debouncing) is to build a low-pass filter out of a capacitor and resistor, and pass that through a schmitt trigger. Some people like to deal with debouncing in software, but that adds a lot of overhead to the code.

Using this look-up table method, you simultaneously eliminate most of the ill effects of bouncing. In the event that either one of the channels bounces, either you will switch back and forth between a +1 and -1 and finally settle on the correct value, or you will come up with a nonsensical combination of channel levels, in which case the look-up table says to add 0 to the encoder counter. Magical!

I've done this a few times, and it works quite well most of the time. There's a few times when the program seems to think the encoder has jumped back a tick, but you can usually ignore it without too much effort. I have also built circuits with the proper debouncing hardware, and the results are flawless. Keep that in mind when your designing your next project with an encoder knob input.  
  
Here's an example schematic of proper debouncing hardware. The encoder is in the middle with a schmitt trigger on each of channels A, B and the encoder button. There's 6 schmitt triggers in the 74HC14N IC, so you only need that one for up to two encoders with buttons.

[](http://2.bp.blogspot.com/-qbaQv5-r0rA/U9w51yYwM2I/AAAAAAAAOvg/dlT2PNZHLz0/s1600/encoder_debounce.png)

# **Rotary Encoder: Immediately Tame your Noisy Encoder! Find out how to Instantly Stop Switch Bounce using one of 2 software methods. Easily get Reliable Operation from your Encoder.**

\*\*\*above-1st-paragraph-new.shtml\*\*  
  
Arduino rotary encoders :Are you struggling to get one reasonable output from your encoder? Encoders are notoriously difficult to get a single output change because of switch bounce. What tends to happen is that multiple increments or decrements occur.

* Find out **two reliable methods** to tame your noisy encoder.
* Get **consistent output**from an encoder.
* Use them to ensure **one position** move results on only **one change** in output.

In this page the Keys KY-040 encoder is used throughout.

A rotary encoder is an input device that you can rotate in either direction continuously. As you turn the device it generates digital pulses to show the direction of rotation using two phased output signals. These two outputs also indicate single position movements, so you can use them in control panels to increment or decrement parameters.

Hair Saved!  
Neil Gleaden : Been pulling my hair out trying to debounce a KY-040 on the STM32, Your "Robust Rotary encoder reading" worked perfectly. You've saved me from having to wear a hat, Many thanks.  
Facebook comment on page.

The Scope Lies! (or needs a closer look;)  
Jessie Kropp : Fantastic. Thank you. I was about to give up on coding for this rotary. On the scope it all looked good but it's amazing how quickly the MCU can read and mis-interpret things (or my bad code I should say).  
Facebook comment on page.

The type of encoder used below for demonstration is also known as an incremental rotary encoder since it generates pulses indicating single step changes. Other types generate an absolute output i.e. the same output number (4 or more bits depending on the accuracy required) is generated for a specific position of the encoder and you would use these in robotics applications.

**Note:** Rotary encoders produce extremely noisy output oscillations due to switch bounce and the information on this page gives you **two techniques to eliminate that noise**. The first is a simple filter method and the second method uses table decoding to get really good output from low quality devices.

The purpose of this tutorial is to provide an example for the Arduino of a simple rotary encoder implementation.

Rotary Encoders allow you to easily increase or decrease parameters by a single value.

As well as generating directional information and step change pulses the device has a physical feedback mechanism that lets you feel when you move from one position to the next. These points are known as detents and range from 12 to 24 positions within a 360º rotation. For the device used here there are 20 detents.

Unlike a potentiometer the rotary encoder has no end stops so you can use one to continually increase or decrease a parameter (once decoded by the microcontroller) and there is no need to set the control position back to a start point (there is none).

They also often have a push button switch built into the shaft which is useful for menu selection etc.

You can use them for many applications ranging from:

* Volume control
* Illumination level control
* Parameter control e.g. for a process e.g. speed, height, temperature etc.
* Menu selection (the push button is useful here).

Since the outputs are digital signals you can process them using a microcontroller and use the result in any way you want i.e. to change the value of a variable representing a system parameter.

In fact rotary encoders look simple, but there is quite a lot going on inside these small devices (~11mm x ~13mm ).

Here's what is going on:

* Two outputs providing quadrature coded signals.
* Physical position feedback and bump stop (known as detents) - for this device there are 20 detents.
* Main shaft push button (a push to make switch).

As you turn the control knob you can feel the each of the 'detent' position stops, so you know when you have turned the device by exactly one position. This provides fine grained physical feedback allowing exact parameter changing. This is very different to using a potentiometer to set the volume level etc. where there is no physical feedback.

# **Quadrature Phase Shift Encoding**

This rather technical sounding encoding method is in fact very simple. all it means is that two signals are offset from one another by a quarter of a period (or phase shifted by 90?). It also falls out in the wash that the signals generated are grey coded which just means that no two signals edges are aligned i.e. signal outputs do not change state at the same time.

Gray coding is useful for electro-mechanical devices to generate signals that are unambiguous. For example if the output was binary coded then at the point of transition (due to small delays in signal paths) you might decode a completely erroneous value i.e. at the point of transition any codes could be generated.

This could be a problem especially if only combinatorial logic is used as the decoder. Gray code stops that from happening (although it does not stop switch bounce).

The following diagram shows the rotary encoder waveform output on pins A and B (CLK) and (DT) respectively.

**[Source PEC11L datasheet]**

**Note:** The D in the diagram above shows where the detent position is located. In fact this is where the outputs are not connected to ground hence they are pulled high by the 10k resistors on the breakout board.

# **Inside the Rotary Encoder**

The following diagram shows the inner workings of the rotary encoder. Each of the three connections 8A, 8B and 8C is formed of a spring arm that pushes down on the substrate.

There are three signals, one connected to the metal substrate (Ground) and two others that move over the alternating substrate pattern. So the outputs are shorted to ground as the device is rotated and then are left floating (unconnected) when the contact is in the substrate gap.

Note how the spring arm contacts are physically offset by a quarter of the period (defined by the physical substrate) - contacts 8B and 8C in the diagram below - this is how the quadrature encoded outputs are generated.

**Source: Expired Patent (Now in the public domain).**

**Note:** The contacts 8A, 8B, 8C are springs that bounce on and off the contact substrate causing the output signals to bounce between high and low i.e. switch bounce.  
  
However spring contact 8A, connecting to pin 8D, won't bounce as it is on a substrate with no breaks. This should be used as ground with 8E and 8F connected to pullup resistors.

# **Types of Rotary Encoder**

## **Contacting Incremental Rotary Encoders**

This is the type of device used in the demonstration on this page. At each detent position two quadrature signals are generated indicating a single position change and showing the direction of rotation.

This particular device has quite a high rotational life - 100k rotations (see datasheet) - but since there is physical contact, the device will eventually wear out. In the Bourns [catalogue](http://www.anglia.com/literature/bourns_encoders.pdf) other physical devices range from 15k to 200k maximum rotations.

## **Optical Encoder**

The PEC11Lhas a maximum RPM of 60RPM whereas optical encoders, in that catalogue, have a 10 million revolution life and can operate at 3000rpm - these are the types you could use for measurements in high speed machinery but see magnetic encoder below which has an even higher life and of course a higher cost!

## **Magnetic Encoder**

For even higher rotational life, a magnetic encoder offers the best choice (since there is no physical contact within the device) the only part that will wear out is the shaft bearings. These offer a rotational life of 100 million revolutions!

These devices come in 4 different flavours:

1. Incremental Quadrature (same as the PEC11L used here).
2. Direction/Step encoder - offers better resolution (up to 512 pulses per revolution).
3. Absolute encoder - allows the absolute position detection of the encoder (1024 codes define the position).
4. PWM Encoder - generates a PWM output from 1us to 1024us width - advantages claimed, are noise immunity and faster data acquisition.

# **Measurements: Using Rotary Incremental Encoders**

The following examples cover the following measurements:

* Velocity
* logarithmic change

## **Velocity**

You might want to measure velocity to use as a parameter in your code e.g. if you turn the wheel faster then perform a different action e.g. change the parameter at a different rate.

## **Logarithmic**

This is a parameter adjustment that measures the rotation speed and if found to be constant periodically increases the parameter. This is very useful for devices that have a large range of control e.g. a DDS (Direct Digital Synthesis) that can output frequency form 1 to 10MHz. You really don't want to sit there turning the knob by 1Hz periods to get to 10MHz!

# **Decoding Methods**

There are two ways to decode rotary encoder outputs:

* Polling
* Interrupts

On the KY-040 there are two signals labelled DT and CLK meaning CLOCK and DATA. If you look at the timing diagrams for these signals it seems obvious to use the CLOCK as a clock and read the DATA input on the rising edge of the clock. However this ignores the fact that the signals bounce all over the place.

If you use the CLK signal as an interrupt you will get into deep trouble as the random bouncing of the input will trigger interrupts all the time (and not at the time when you want to read the data signal) so you will get incorrect data.

There are polling methods to decode the grey coded signals using a state machine so that bouncing signals are ignored i.e. error states are ignored. These are quite complex and sometimes get out of sync.

The way I use these devices is a combination of a small amount of smoothing capacitor and a simple digital rotary switch debounce algorithm. (See code below). This provides easy to understand code (also small code size) and works to accurately obtain individual detent position information and also accurate directional rotation information.

Sometimes, though, you may have a very poor quality encoder and that needs more effort to decode and in that case you will need to look at the more complex robust decoder code [here](https://www.best-microcontroller-projects.com/rotary-encoder.html#Taming_Noisy_Rotary_Encoders).

## **Device Decoding Techniques**

You can use many clever ways to decode the outputs involving complex state machines and gray decoding algorithms. Some use interrupts, and most use polling.

The problem with connecting the outputs to an interrupt pin is that you have no control over the bounce that you may experience and the processor could be interrupted too much to do any useful work (and may even hang) and will get the wrong value from the data input anyway.

**Warning:** Rotary encoders are extremely noisy because of switch bounce due to the internal construction of the device (using physical contact springs that bounce over the substrate connections). This makes it extremely difficult to accurately decode the device outputs.

**However see my new technique - the last code example code.**

Switch bounce occurs as the contacts are springs that bounce on and off a substrate contact - even the data sheet indicates the switch bounce you can get for each turn is up to 10ms (Bourns PEC11L).

**[Source PEC11L datasheet]**

You can see that the signals labelled A and B can be changed to CLK (clock) and DT (data) and that a rising edge on a clock signal (A) will give a a logic low on DT (B) if turned clockwise, and a logic high if turned anti-clockwise (when turned in the opposite direction the falling edge becomes the rising edge!).

### Capacitor Smoothing

Adding a huge smoothing capacitor (and resistor see diagram below and replace the 0.01uF with 470nF as an example of a "too big" capacitor" - which is suggested by some people) to stop the bounce will stop the bounce but also slows the input signal level to the point where it will go through the undefined logic input level (below the top threshold VIH and above the lower threshold VIL) of the microcontroller. In this input region noise on that input could (and often does!) trigger the input to high or low causing oscillation i.e. making more bounce signals and not solving the problem at all.

**Note:**Some microcontroller inputs have built in Schmitt trigger inputs.

You can get around this by using a schmitt trigger device such as a 74HC14 to create the correct fast edge signals but you may alter the timing too much to get a useful output signal.

### RC pair and digital Filter

One way I have found is to use a small smoothing capacitor resistor pair along with a digital debounce filter. This allows individual detent positions to be accurately identified (slow turning of the control shaft is accurately decoded). At faster revolutions codes are missed but the real point of the rotary encoder is to allow accurate individual detent (and direction) detection. You don't need to know exact detent stops for fast revolutions - all you need is to know that the user wants to increase the parameter faster.

### Digital Debounce Filter

The digital filter is made up of a single 16 bit integer variable into which you shift the current state of the input pin:

state=(state<<1) | digitalRead(CLK\_PIN) | 0xe000;

This is a very compact filter - each time round the loop a new bit is shifted left (at bit 0). The "or" action with 0xe000 defines the number of iterations i.e. the top 3 bits are blocked off leaving the rest as useful inputs. The idea is that you test for the state 0xf000 which can only occur if there was a sequence of 1 0000 0000 0000 inputs meaning that the signal has been stable for 12 iterations around the loop i.e. not bouncing around.

# **Arduino Rotary Encoder Datasheet**

The rotary encoder used in the KY-040 looks like a Bourns PEC11L device - you can download that rotary encoder datasheet from the link below. All that the breakout board does is add two 10k pullup resistors (R2 and R3) while the space for the switch pullup has been left blank.

Download [PEC11L datasheet](https://www.best-microcontroller-projects.com/support-files/pec11l.pdf).

# **Arduino Rotary Encoder Software Setup:**

IDE Version Used : 1.6.4  
Board used : Arduino Uno R3

# **Rotary Encoder Hardware Setup**

Device Used : KY-040 (breakout board)

Other components 10k resistor and 10nF capacitor - only for the clock signal, connected as shown below:

**[Source PEC11L datasheet]**

Note: 10k and 10n are extra to the breakout board (A and B have the 10k pullups on the board). Only add them to the clock signal (A).

# **Example Rotary Encoder Code:**

This is a an **arduino ky-040 rotary encoder example** that shows you how to decode the 20 turn encoder by removing switch bounce. using a digital filter technique. The operation of this digital filter is discussed [here](https://www.best-microcontroller-projects.com/rotary-encoder.html#Digital_Debounce_Filter).

**Note:** The quality of the encoder will affect the output signal (one of mine skips codes whereas a higher quality one does not!).

Copy Sketch

#define CLK\_PIN 2

#define DATA\_PIN 7

#define YLED A1

*////////////////////////////////////////////////////*

**void** setup() {

pinMode(CLK\_PIN,INPUT);

pinMode(DATA\_PIN,INPUT);

pinMode(YLED,OUTPUT);

Serial.begin(9600);

Serial.println("Rotary Encoder KY-040");

}

*////////////////////////////////////////////////////*

**void** loop() {

**static** uint16\_t state=0,counter=0;

delayMicroseconds(100); *// Simulate doing somehing else as well.*

state=(state<<1) | digitalRead(CLK\_PIN) | 0xe000;

**if** (state==0xf000){

state=0x0000;

**if**(digitalRead(DATA\_PIN))

counter++;

**else**

counter--;

Serial.println(counter);

}

}

# **Taming Noisy Rotary Encoders**

The keyes-040 encoder can be very noisy due to switch bounce and you may need to use a much more robust way of decoding it - I have one that is fairly well behaved and one that is massively noisy.

The following example uses a table decode method which requires more code than in the previous example but is capable of reading rotary encoders without needing any debounce capacitors at all. (However check that this works with your own hardware to make sure of this).

The way it works is to encode the outputs of the decoder as a binary number. To do this you can define the CLK as an LSB binary digit and DATA as the MSB binary digit.

Then observe the valid states that the outputs can occupy i.e. the ones shown in the diagram below by dotted lines.

Because the outputs are in quadrature and because this results in a gray code output no output changes state at the same time as another. What this means is that only one of the two outputs will be bouncing around at any transition edge. This means bouncing signals can be easily ignored because the bouncing mostly produces invalid encoder states.

If you look at the diagram above you can see that there are four states (11, 10, 00, 01). In addition to that, there are only 8 ways that you can move from one state to the next including going backwards (Anti Clockwise).

For clockwise motion you can only perform the following actions:

(11 > 10), (10 > 00), (00 > 01) and (01 >11)

Similarly only the following encoder output transitions are valid for Anti-Clockwise rotation:

(01 > 00), (00 > 10), (10 > 11), and (11 > 01)

You can find other rotary decoding methods (including this one) [here](http://web.engr.oregonstate.edu/~traylor/ece473/student_projects/ReadingEncoderSwitches.pdf).

The idea behind the table method is that you store the previous state and current state and set them out as a binary code. In this way the table directly encodes the transition of valid outputs - the main purpose behind the technique is that invalid ones caused by switch bounce are discarded.

## **Valid Code Output**

So for the clockwise direction above, there are four valid outputs (where the 2 MSBits are the previous state and the 2 LSBits are the current state):

1110

1000

0001

0111

Only these are valid states. In theory only these should be output by the rotary encoder but in practice switch bounce generates other codes.

For the opposite direction (Anti Clockwise) the following codes are valid:

0100

0010

1011

1101

To allow the microcontroller to check for valid codes and ignore invalid ones a table is needed (with the 4 bit PSNS - Previous State Next State - code as the input):

|  |  |  |
| --- | --- | --- |
|  | | |
| **PSNS (Prev State, Next State)** | **Valid code** | **Direction** |
| 0000 | X | X |
| 0001 | Valid | CW |
| 0010 | Valid | CCW |
| 0011 | X | X |
| 0100 | Valid | CCW |
| 0101 | X | X |
| 0110 | X | X |
| 0111 | Valid | CW |
| 1000 | Valid | CW |
| 1001 | X | X |
| 1010 | X | X |
| 1011 | Valid | CCW |
| 1100 | X | X |
| 1101 | Valid | CCW |
| 1110 | Valid | CW |
| 1111 | X | X |

Coding this into a C table and replacing CW with 1 and CCW with -1 and invalid as 0 results in the following:

rot\_enc\_table[]= {0,1,-1,0,-1,0,0,1,1,0,0,-1,0,-1,1,0};

You can find code that uses this method elsewhere on the web ( I may have swapped -1 for 1 - does not really matter just swap SIG A and SIG B) but that method takes any CW or CCW valid output as a real transition and so for a "detent" to "detent" movement four CWs or four CCW states are returned (the detent position is shown in the diagram below). The problem is that bouncing may cause the a state change backwards until the switch settles and goes forwards again.

# **Improved Table Decode Method**

By using the following code you can see the outputs generated between each detent. The code simply generates a newline when it finds either 7 or 0xB. These are the last codes generated when performing a detent to detent rotation.

## **Rotary Encoder Quality Test Program**

Use the following program to see how good/bad your encoder is (observe typical results below).

Copy Sketch

*#define* CLK 2

*#define* DATA 7

*#define* BUTTON A5

*#define* YLED A2

**void** setup() {

pinMode(CLK, INPUT);

pinMode(CLK, INPUT\_PULLUP);

pinMode(DATA, INPUT);

pinMode(DATA, INPUT\_PULLUP);

pinMode(BUTTON, INPUT);

pinMode(BUTTON, INPUT\_PULLUP);

pinMode(YLED,OUTPUT);

Serial.begin (115200);

Serial.println("KY-040 Quality test:");

}

**static** uint8\_t prevNextCode = 0;

**void** loop() {

uint32\_t pwas=0;

**if**( read\_rotary() ) {

Serial.print(prevNextCode&0xf,HEX);Serial.print(" ");

**if** ( (prevNextCode&0x0f)==0x0b) Serial.println("eleven ");

**if** ( (prevNextCode&0x0f)==0x07) Serial.println("seven ");

}

**if** (digitalRead(BUTTON)==0) {

delay(10);

**if** (digitalRead(BUTTON)==0) {

Serial.println("Next Detent");

**while**(digitalRead(BUTTON)==0);

}

}

}

*// A vald CW or CCW move returns 1, invalid returns 0.*

int8\_t read\_rotary() {

**static** int8\_t rot\_enc\_table[] = {0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0};

prevNextCode <<= 2;

**if** (digitalRead(DATA)) prevNextCode |= 0x02;

**if** (digitalRead(CLK)) prevNextCode |= 0x01;

prevNextCode &= 0x0f;

**return** ( rot\_enc\_table[( prevNextCode & 0x0f )]);

}

Using the program above I pushed the rotary encoder pushbutton to generate the text "Next Detent" before turning the encoder to the next detent position. This allows you to see all the codes that were generated during one position change.

You can see that some of the rotations caused a lot of codes but the codes were only able to go back by one state before returning to the correct one. More importantly you can see that the final 2 codes are always the same matching the 2 last nibbles for complete rotation sequences : D42B and E817.

For the "bad" rotary encoder the following output was generated:

Poor Quality Rotary Encoder

KY-040 Quality test:

D 4 2 8 2 B eleven

Next Detent

D 4 2 8 2 B eleven

Next Detent

D 4 1 4 2 B eleven

Next Detent

E 8 2 8 2 8 2 8 1 4 1 7 seven

D 7 seven

Next Detent

E B eleven

E 8 2 8 2 8 1 7 seven

Next Detent

E 8 2 8 2 8 2 8 1 4 1 7 seven

Next Detent

E B eleven

E 8 2 8 2 8 1 4 1 4 1 7 seven

Next Detent

E 8 1 4 1 4 1 4 1 4 1 4 1 7 seven

Next Detent

E 8 1 4 1 7 seven

Next Detent

E 8 1 7 seven

Next Detent

E B eleven

E B eleven

E B eleven

E B eleven

E B eleven

E B eleven

E B eleven

E 8 2 8 2 8 1 4 1 7 seven

Next Detent

For the good quality encoder the following output was generated:

Rotary Encoder test better quality encoder.

KY-040 Quality test:

E 8 1 7 seven

Next Detent

E 8 1 7 seven

Next Detent

E 8 1 7 seven

Next Detent

E 8 1 7 seven

Next Detent

E 8 1 7 seven

E B eleven

Next Detent

E B eleven

E B eleven

E B eleven

E B eleven

E B eleven

E B eleven

E B eleven

E B eleven

E B eleven

E 8 1 7 seven

Next Detent

D 4 2 B eleven

D 7 seven

D 7 seven

Next Detent

D 4 2 B eleven

Next Detent

D 4 2 B eleven

Next Detent

You can see that there is quite a difference between the two with the 1st one generating far more code outputs (due to switch bouncing). The actual codes that indicate a single detent-to-detent movement are E817 and D42B which are the same values (shown in the discussion on valid binary codes above) for the "valid" prevstate,nextstate coding.

You can see that there is a lot of bouncing while the switch is between detents but not when it reaches the end. All the code outputs start with the correct code either E or D then bounce around a lot and then end with the final two codes.

## **Operation of Improved Table Decode Code**

The code below looks for the last two states to indicate a valid rotary code output ( 0x2b and 0x17 ). This gives a double processing debounce - the first being a "valid" output and the second being a "valid rotation". This works very well and even allows the rotary encoder to return to its original position (rotate CW 20 positions then rotate CCW 20 positions) with no missing codes - even for an extremely noisy rotary encoder.

You may be able to use the full 16 bit hex code for less noisy ones (or high quality ones). Your results may vary.

Note that this is with direct connection to the encoder - no debounce resistors or capacitors (only the 10k resistor pull ups on the breakout board).

## **Code For Improved Table Decode**

Copy Sketch

*// Robust Rotary encoder reading*

*//*

*// Copyright John Main - best-microcontroller-projects.com*

*//*

*#define* CLK 2

*#define* DATA 7

**void** setup() {

pinMode(CLK, INPUT);

pinMode(CLK, INPUT\_PULLUP);

pinMode(DATA, INPUT);

pinMode(DATA, INPUT\_PULLUP);

Serial.begin (115200);

Serial.println("KY-040 Start:");

}

**static** uint8\_t prevNextCode = 0;

**static** uint16\_t store=0;

**void** loop() {

**static** int8\_t c,val;

**if**( val=read\_rotary() ) {

c +=val;

Serial.print(c);Serial.print(" ");

**if** ( prevNextCode==0x0b) {

Serial.print("eleven ");

Serial.println(store,HEX);

}

**if** ( prevNextCode==0x07) {

Serial.print("seven ");

Serial.println(store,HEX);

}

}

}

*// A vald CW or CCW move returns 1, invalid returns 0.*

int8\_t read\_rotary() {

**static** int8\_t rot\_enc\_table[] = {0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0};

prevNextCode <<= 2;

**if** (digitalRead(DATA)) prevNextCode |= 0x02;

**if** (digitalRead(CLK)) prevNextCode |= 0x01;

prevNextCode &= 0x0f;

*// If valid then store as 16 bit data.*

**if** (rot\_enc\_table[prevNextCode] ) {

store <<= 4;

store |= prevNextCode;

*//if (store==0xd42b) return 1;*

*//if (store==0xe817) return -1;*

**if** ((store&0xff)==0x2b) **return** -1;

**if** ((store&0xff)==0x17) **return** 1;

}

**return** 0;

}

----