Makeatronics

February 10, 2013

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

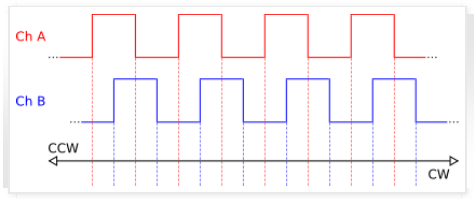
Absolute position encoders will tell you what the angular position is all the time, even between power cycles. Most use "grey code" (a modified form of binary) with several "tracks" (bits) to read position.

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?



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

Search This Blog

Search

Popular Posts

BLDC Motor Control

Efficiently Reading Quadrature With Interrupts

24V AC Solid State Relay Board

Raspberry Pi Thermostat Hookups

Smart BLDC Commutator - Hardware

Thermostat Software

3D Printer Motor Control - Part 1

BLDC Hall Effect Sensors

Building a 3D Printer

Sous Vide Part 3: Advanced Development

Labels

3d printer attiny battery bldc motor closed loop control encoder external interrupt feedback graph IR organization pcb programmer quadrature raspberry pi resistors sensors software sous vide store temperature control thermostat triac usb

About Me

Nich Fugal

View my complete profile

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:

```
volatile long enc_count = 0;

void setup() {
    // all your normal setup code
    attachInterrupt(0,encoder_isr,CHANGE);
    attachInterrupt(1,encoder_isr,CHANGE);
}
```

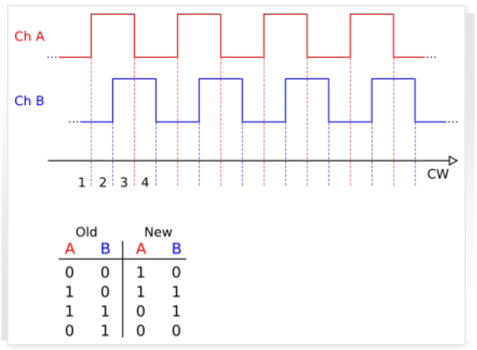
Some important things here. First, I've defined `enc_count` as a global variable for future use. Next, an ISR (Interrupt Service Routine) is a bit of code that gets called whenever the interrupt is triggered. ISR's take no inputs and give no outputs (hence the need for a global `enc_count`). You'll define 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



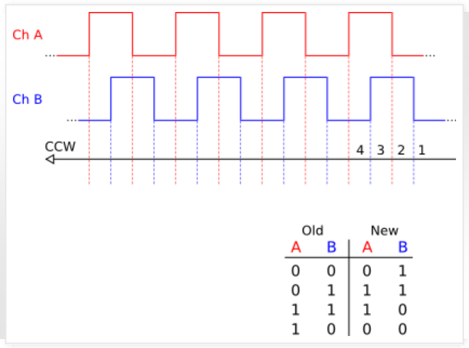
Old A	Old B	New A	New B
0	0	1	0
1	0	1	1
1	1	0	1
0	1	0	0

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`

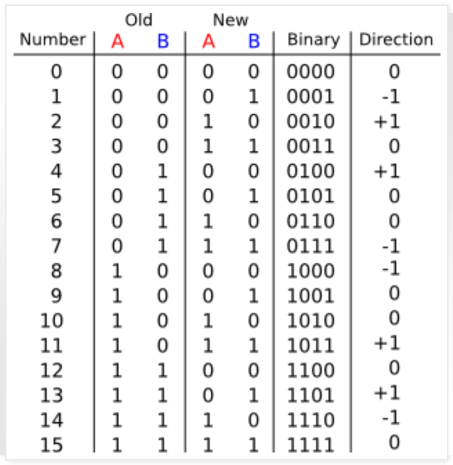
Imagine we were in state 1 and both channels were reading low, and then the rising edge of channel A triggers an ISR and puts us into state 2. The old reading of "`00`" is remembered for future reference and the new levels "`10`" are read. Let's concatenate these two pieces of information into "`0010`" 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:



Old A	Old B	New A	New B
0	0	0	1
0	1	1	1
1	1	1	0
1	0	0	0

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`.



	Old		New			
Number	A	B	A	B	Binary	Direction
0	0	0	0	0	0000	0
1	0	0	0	1	0001	-1
2	0	0	1	0	0010	+1
3	0	0	1	1	0011	0
4	0	1	0	0	0100	+1
5	0	1	0	1	0101	0
6	0	1	1	0	0110	0
7	0	1	1	1	0111	-1
8	1	0	0	0	1000	-1
9	1	0	0	1	1001	0
10	1	0	1	0	1010	0
11	1	0	1	1	1011	+1
12	1	1	0	0	1100	0
13	1	1	0	1	1101	+1
14	1	1	1	0	1110	-1
15	1	1	1	1	1111	0

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?

```
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:

```
 8. void encoder_isr() {
 9.     static int8_t lookup_table[] =
{0,-1,1,0,1,0,0,-1,-1,0,0,1,0,1,-1,0};
10.     static uint8_t enc_val = 0;
11.
12.     enc_val = enc_val << 2;
13.     enc_val = enc_val | ((PIND & 0b1100) >> 2)
14.
15.     enc_count = enc_count + lookup_table[enc_val & 0b1111];
16. }
```

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 `PIN`s 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:

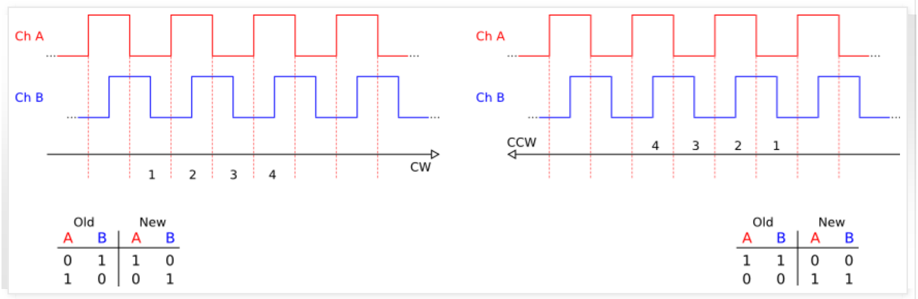
```
  11001101
& 00001111
----------
  00001101
```

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.

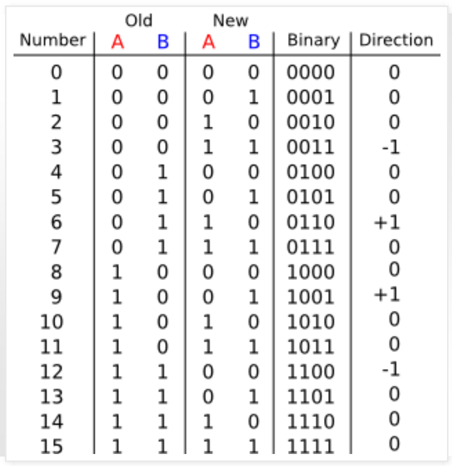


Old A	Old B	New A	New B
0	1	1	0
1	0	0	1

Old A	Old B	New A	New B
1	1	0	0
0	0	1	1

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:



	Old		New			
Number	A	B	A	B	Binary	Direction
0	0	0	0	0	0000	0
1	0	0	0	1	0001	0
2	0	0	1	0	0010	0
3	0	0	1	1	0011	-1
4	0	1	0	0	0100	0
5	0	1	0	1	0101	0
6	0	1	1	0	0110	+1
7	0	1	1	1	0111	0
8	1	0	0	0	1000	0
9	1	0	0	1	1001	+1
10	1	0	1	0	1010	0
11	1	0	1	1	1011	0
12	1	1	0	0	1100	-1
13	1	1	0	1	1101	0
14	1	1	1	0	1110	0
15	1	1	1	1	1111	0

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:

```
void encoder_isr() {
    static int8_t lookup_table[] = {0,0,0,-1,0,0,1,0,0,1,0,0,-1,0,0,0};
    static uint8_t enc_val = 0;

    enc_val = enc_val << 2;
    enc_val = enc_val | ((PIND & 0b11000) >> 3)

    enc_count = enc_count + lookup_table[enc_val & 0b1111];
}
```

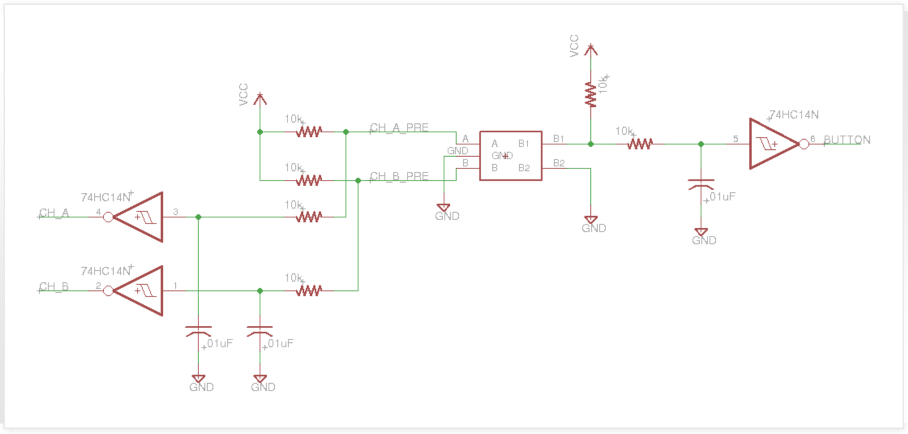
A Word on Debouncing

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.



Labels: closed loop control, encoder, external interrupt, feedback, quadrature, sensors, software

33 comments:

Kevin April 28, 2013 11:35 PM

Here is some of the original code I used with an Arduino project to read rotary encoders properly.

http://www.circuitsathome.com/mcu/reading-rotary-encoder-on-arduino/comment-page-1#comments

Reply

Replies

Nich Fugal April 29, 2013 12:35 AM

Eureka! That's where I got the lookup table from. Try as I might, I just couldn't find it again when I wrote this post.

Reply

ckiick April 29, 2013 2:13 PM

[quote]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. [/quote]

Not quite correct. In a lot of cases (robots for example) you already know which direction the motor is turning, since you were the one who turned it on in the first place. Incremental encoders are very useful in those cases because what you want to know is how far and how fast the motor is turning, but you don't need to compute the direction. There are lots of applications in position control for incremental encoders.
They are also more efficient since they take much less processing than quadrature encoders, so you can handle faster speeds (more pulses/sec) or higher resolutions.

Reply

Replies

Nich Fugal April 29, 2013 6:26 PM

[quote] In a lot of cases (robots for example) you already know which direction the motor is turning, since you were the one who turned it on in the first place. [/quote]

While this is certainly the case in many circumstances, you must exercise caution when assuming the motor is turning the same direction you commanded it to. I learned this lesson the hard way, melting the motor windings when the controller went unstable.

The problem lies in overshoot, when the motor passes your set point and the inertia keeps it going in the same direction even when the controller attempts to reverse it. If you assume the motor changes direction instantaneously you will at best lose your position forever, or at worst send the controller unstable.

See the following simulation. Magenta is the motor position (given a step command to 1 at t=1), cyan is motor velocity, yellow is motor command. Notice how the control signal (yellow) goes negative while velocity (cyan) is still positive.

https://docs.google.com/file/d/0B9DuE3f5Cnm7UEtoaEJjczBaUkE/edit?usp=sharing

Reply

Rob Barris April 29, 2013 5:50 PM

You can also cleanly visualize this by drawing a square between (0,0) and (1,1) and mapping out each high/low combination to a corner of the square - transitions then march between corners of the square, either CW or CCW and matching the device of choice if you drew the chart correctly.

Reply

Anonymous January 18, 2014 1:39 PM

Thanks a million for the very clear and proper write up. Quite stellar what you have written, actually. Kudos.

I searched and found many quadrature how-to's for the Arduino that failed to understand how to utilize all four phases of the grey code. Or they got things completely wrong leaving me to wonder if they ever tried to actually get the software and hardware to work together.

Also, your look-up table solution for the ISR is just great. Interrupt routines need to use as few processor cycles as possible and likely yours would be hard to improve upon. Your solution will likely become part of my library of routines and maybe you should consider submitting it for inclusion as a standard Arduino library.

I look forward to getting my recently arrived shaft encoder interfaced to my Arduino project. It's a VFO for a ham radio project and I can't wait to get it on the air!

-Paul

Reply

Unknown March 05, 2014 4:40 AM

can you provide script to move forward a robot to specific distance using encoder and interrupts

Reply

Anonymous May 28, 2014 5:43 PM

Keep in mind that there is one seldom seen but still possible 'bug' in this usage. The global variable is a long (4 bytes) and when the main loop is reading the value for some usage it's possible the the ISR can trigger and increment or decrements the variable while the main loop is reading this variable and can miss that a byte in the long variable may have carried over or borrowed from an adjacent byte of the long variable, the so called 'atomic' problem, can seen as a large change in value. The solution is that the main loop function before reading the global variable should perform a nointerrupt() statement, and after reading the variable perform a interrupt() statement.

Reply

Replies

Nich Fugal July 07, 2014 10:37 PM

Thanks for pointing that out. Turning the interrupts off and on again should work for user input knobs or other slow speed applications. I worry though at high speed that turning the interrupts off could lead to missing a pulse and throwing off your position.

Another option would be to set an increment flag in the ISR and an int_8 increment value with either -1 or 1 in it. Then the main loop could check the flag and add the increment value to the global long. Haven't thought that one all the way through yet, but seems like it should work.

Unknown August 22, 2016 12:10 PM

Turn the interrupts off. DOn't worry about missing the nest pulse. Pulses do not happen at random times. You KNOW the minimum time until there nest pulse because you KNOW the maximum speed of the motor. Having the main lop check a int_8 means that the loop must run VERY faster than the pulses come in. Then way bother with interrupts if you are going to poll so quickly? If you disable the interrupts then you know the ISP is atomic and the main code can't be running.

It's a common trick: Disable interrupts when inside a "critical section" of code. You see this used inside device drivers al the time. The key is that Interrupts are predictable

Reply

James July 18, 2014 9:02 PM

Terrific piece of work Nick. I used on the Uno and got a version running for the Due. Thanks for that.

Reply

Dave June 17, 2015 6:19 AM

Hey,

Thanks for the great tutorial. One question though;
Why did you put the "lookup_table" array and the "enc_val" into the ISR function ? Those are just the initializations. So, shouldn't they be at the very beginning. Well, since they are static, nothing changes so it is also okay that way. However, in terms of speed of the ISR function, would it make any difference ? reading two lines more ?

Reply

Replies

Nich Fugal June 22, 2015 9:03 PM

They are inside the function to avoid cluttering the global space with unneeded variables, and since they are declared as static the compiler optimizes the code so that it's not re-evaluated with every call to the function.

If it were evaluating those two lines each time, then yes there would be a slight speed degradation. You'd have to count the number of clock cycles it would take to evaluate each line. But as I once read somewhere, never underestimate the power of optimization in the compiler.

Reply

Anonymous January 13, 2016 4:23 AM

Thanks for the info. I tried it on a nano board with 2 interrups and works fine.

Reply

Anonymous February 08, 2016 12:18 PM

Perfect tutorial !Thumbs up !
my cfg: ATMEGA16, 16MHz, two INT used
problem : each encoder turn generates 4 interrupts
Why ? Is it of encoder type ?

Reply

Frank February 18, 2016 4:42 AM

Hi, i wanna try to use an encoder to navigate in my Display(LCD).
Im useing 2 Interrupts (rising edge).

Will this code be working?

```
SIGNAL(INT0_vect) // Chanal A will call this ISR
{
CH_A=1; // Set globle var to 1
If(CH_A && CH_B)
{
i++; // increment
CH_A=0;CH_B=0;
}
}


SIGNAL(INT1_vect) // Chanal B will call this ISR
{
CH_B=1; // Set globle var to 1
If(CH_A && CH_B)
{
i--; // decrement
CH_A=0;CH_B=0;
}
}
```

Reply

Anonymous July 26, 2016 2:01 AM

No

Reply

Replies

Anonymous August 02, 2016 4:54 PM

Yes

Reply

Unknown November 04, 2016 1:06 PM

Hi, I know you're probably not checking this anymore but I just want to say thanks! arduinos pretty new to me so when I started working with fast encoders i quickly saw the issue with 10ms sampling times, this bit of code works great! currrently using it to work on closed loop control of pololu gear motors.

Reply

Anonymous August 22, 2017 11:17 PM

Thanks so much for the excellent explanation!

Reply

Anonymous March 17, 2018 4:13 AM

Hello author,
thanks for valuable info!!!

I succesfully transferred the code to my Atmega169 with Atmel Studio.
But, one thing I had to correct:
LINE 12: enc_val = (enc_val << 2) & 0b00001111;
If not, ANDed, the value of enc_val becomes greater than 15 with multiple ISR calls.

Greetz,
Mike

Reply

sumit June 09, 2018 3:12 AM

Hello,

Can we measure the time of the leading pulse & ignore the lagging pulse for each individual direction . i.e for both CW & CCW Direction

Reply

Replies

sumit June 09, 2018 3:13 AM

I mean time between the leading pulses

Reply

Anonymous February 17, 2019 1:15 PM

Thank you, thank you, thank you. After surfing different sites and forums to piece together an understanding of the lookup table, I found this site. I'm not using the same programming language as you, but thanks to your perfect explanation, I had no issue writing code for this in another language. You've taken the mystery out of this encoder debouncing method for me.

Reply

Unknown August 12, 2019 6:54 AM

Seems to be a useful solution to resolve encoder mystery.....will definitely try this on Atmega16....Thanks

Reply

Jay August 18, 2019 3:26 PM

Can anyone help me figure out why i am getting 4x too many pulses?

Thank you for this great tutorial! It really opened my eyes to how the quadrature encoding pattern works and some clever ways to "use binary" to red the inputs (previously I'd have done some nested IF statements akin to "if this before than that, advance by 1, etc." This solution is much more elegant (and presumably faster.)

Though I believe I wrote my script nearly identical to this tutorial, I seem to be getting exactly 4x too many pulses out of my encoders (I have tried 2 different ones - a 100 p/rev. and a 360 p/rev. - I am seeing exactly 400 p/rev on the first and exactly 1440 p/rev on the second.

My leading theory is the ue of "CHANGE" as the interrupt signal. I THINK I am triggering the interrupt each time the encoder pulse goes low and then high again on each phase.

I have tried other settings (FALLING, RISING) and turned off and back on again the INPUT_PULLUP on inputs 2 and 3, to no avail. The encoder won't even provide pulses if turn off "PULLUP" and I get minimal, seemingly random pulses if I try "FALLING"

Encoder documentation claims the the pulses go "low."

Any ideas what may be going on?

Thank you very much,
Jay

Reply

Replies

Jay August 18, 2019 11:48 PM

After I wrote this post, I did some more analyzing of my issue and came to a conclusion - just in case someone stumbles on this post, I am posting in case someone finds it helpful.

I was correct about the interrupt acting on "CHANGE" causing the issue, but it was slightly more detailed than that. I was reading that quadrature encoders can run in several "modes": x1, x2, x4 and that each of these "modes" results in a multiplication of the number of Pulses Per Revolution because of how the pulses are read.

x1 only reads single pulses as the phase either rises or falls.
x2 reads only the rising or falling edge of the pulse
x4 reads both the rising and falling edge of each phase's pulse.

This tutorial has the encoder working in "x4 mode." This is not necessarily an issue, it effectively multiples your encoder's precision by 4x (so my 100 PPR encoder was actually delivering 400 ppr) which may be advantageous in some situations - I just didn't understand why it was happening.

Based on how we are reading the encoder and looking over the logic table, I figured how to get this to an "X2" but not an X1 (I am not sure if you can achieve "X1" reading this way, but maybe I am just not there yet. So I embarked on converting my coder to X2 which would result in 200 PPR instead of 400 PPR.

I re-wrote the "logic table" of pulses and looked at which pulses would be detected if I was only looking for the FALLING edge of the pulses and came up with a total of 4. (In one direction: 01, 00 and in the other direction, 00, 10).

As the tutorial above instructs, you place the old value next to the new value to generate a binary number indicating the "position" within our lookp_table array and we find that one direction results in position 1 and 4 and the reverse direction results in position 8 and 2.

The reason I was getting erratic behavior when I changed my detection to "FALLING" instead of "CHANGE" was that the "binary output" of the encoder was pointing to the wrong values in the lookup_table. (I further realized it wasn't erratic, it was just that the original arrangement of 1's and -1's in the lookup_table, each sequential pulse happened to point to a +1 and then a -1, cancelling each other out.

What resulted was that I needed to create a new lookup_table array that looked like this:

static int8_tlookup_table = {0,1,-1,0,1,0,0,0,-1,0,0,0,0,0,0,0};

I don't know if I am 100% correct on this, but I think this is the correct logic table for an "X2 mode" quadrature encoder setup.

Reply

CoreyG September 22, 2020 7:36 AM

I know this post is old (Relative to your own perspective) but the information is not. I just wanted to say "Thank You" for this explanation and example!

Reply

CoreyG September 22, 2020 7:36 AM

This comment has been removed by the author.

Reply

Lorenzo Leandro October 14, 2020 5:20 AM

Thank you so much for the thorough explanation!
I am controlling a high-res encoder SCANCON SCH50B with 12500ppr with an Arduino Uno.
I wonder how do I know if I am losing interrupts (e.g. if the motor is spinning too fast). I need to be absolutely certain that the code/electronics is fast enough to not be overrun by the hardware-specific application.

Thanks!

Reply

Replies

Lorenzo Leandro October 15, 2020 4:01 AM

I am also wondering what is the most efficient way to save the data from arduino to CSV file. If I do it in the loop then I have many points saved. If I do it in the interrupt routine I risk slowing that down. Any advice?

Thanks a lot

Reply

Unknown February 17, 2022 1:16 PM

Hello,
It Works with this simple code where SIN, COS are A, B They can be obtained from Analog Encoder and ADC(12-Bit):

```
if ((COS != SIN_1) && (SIN == COS_1)) QC++;
if ((COS == SIN_1) && (SIN != COS_1)) QC--;
if ((SIN == SIN_1) && (COS == COS_1)) QC = QC; //No Action
if ((SIN != SIN_1) && (COS != COS_1)) Serial.println("Invalid");
SIN_1 = SIN;
COS_1 = COS;
```

Reply

Mark March 10, 2022 5:13 PM

Thanks, banged in your code and it worked first time. You saved me some considerable time.

Reply

Enter Comment

Newer Post Home Older Post