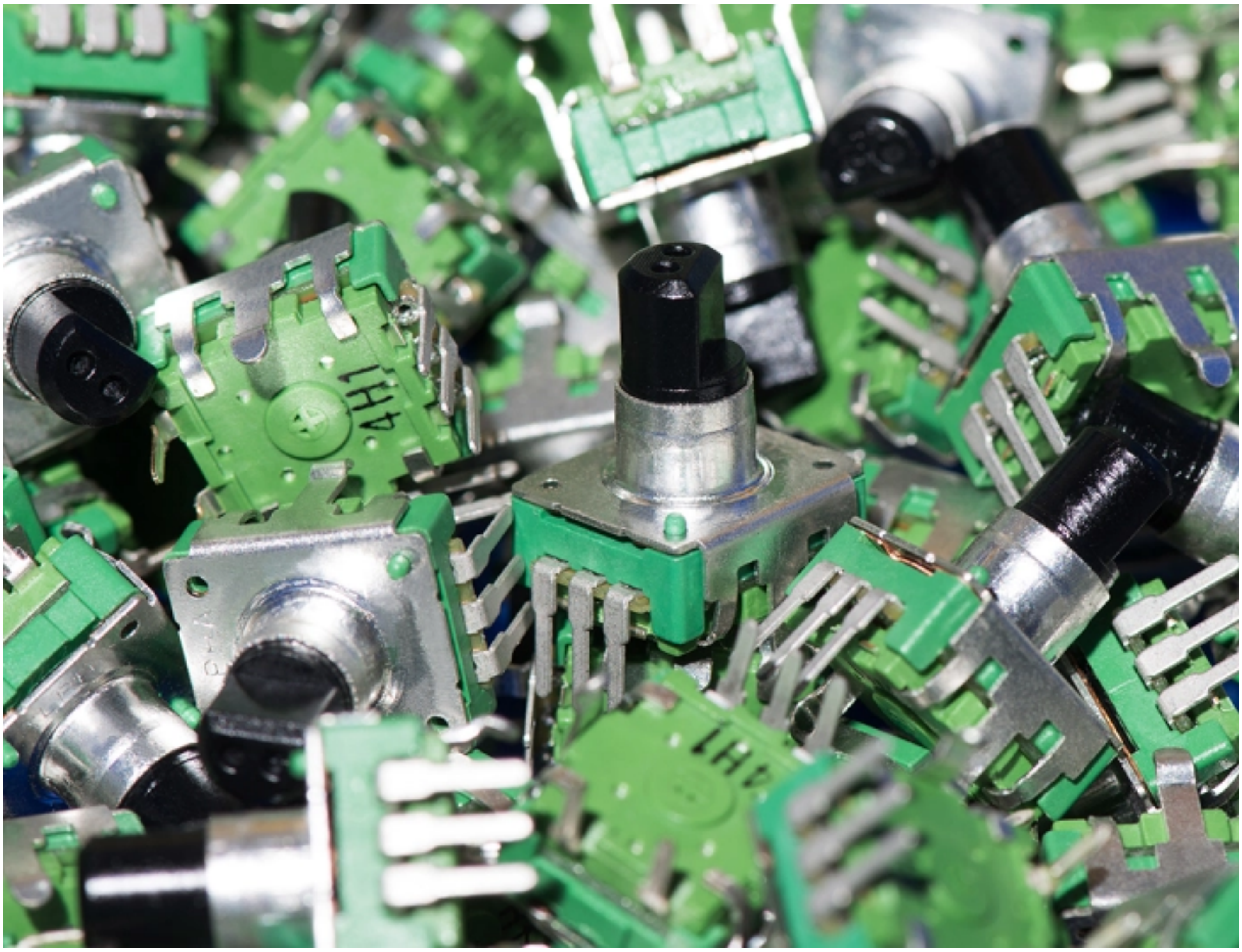
HACKAD

HOME BLOG HACKADAY.IO TINDIE HACKADAY PRIZE SU

A ROTARY ENCODER: HOW HARD CAN IT BE?

by: Al Williams

A



As you may have noticed, I've been working with an STM32 ARM CPU using Mbed. There was
Mbed was pretty simple, but a lot has changed since it has morphed into Mbed OS. Unfortunat
means that a lot of libraries and examples you can find don't work with the newer system.

I needed a rotary encoder — I pulled a cheap one out of one of those "49 boards for Arduino"
around. Not the finest encoder in the land, I'm sure, but it should do the job. Unfortunately, Mb
doesn't have a driver for an encoder and the first few third-party libraries I found either worked
or wouldn't compile with the latest Mbed. Of course, reading an encoder isn't a mysterious pro
hard can it be to write the code yourself? How hard, indeed. I thought I'd share my code and th
how I got there.

There are many ways you can read a rotary encoder. Some are probably better than my metho
these cheap mechanical encoders are terrible. If you were trying to do precision work, you sho
be looking at a different technology like an optical encoder. I mention this because it is nearly i
read one of these flawlessly.

So my goal was simple: I wanted something interrupt driven. Most of what I found required you periodically call some function or set up a timer interrupt. Then they built a state machine to tra encoder. That's fine, but it means you eat up a lot of processor just to check in on the encoder moving. The STM32 CPU can easily interrupt with a pin changes, so that's what I wanted.

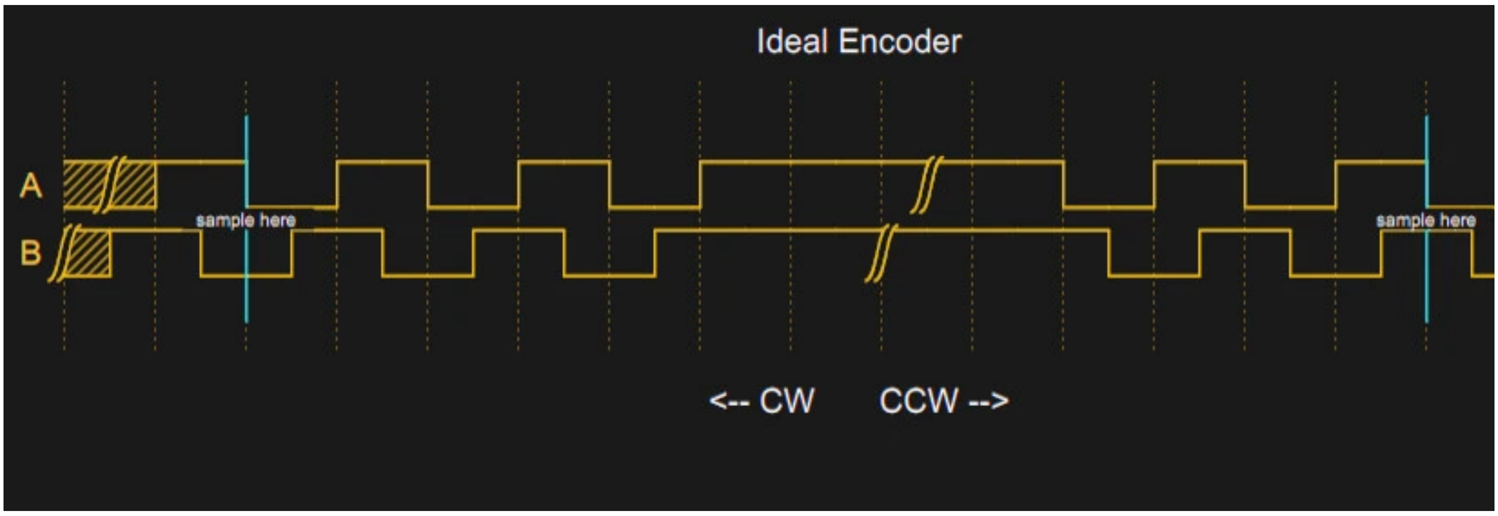
THE CATCH

The problem is, of course, that mechanical switches bounce. So you have to filter that bounce hardware or software. I really didn't want to put in any extra hardware more than a capacitor, sc software would have to handle it.

I also didn't want to use any more interrupts than absolutely necessary. The Mbed system mak handle interrupts, but there is a bit of latency. Actually, after it was all over, I measured the later isn't that bad — I'll talk about that a little later. Regardless, I had decided to try to use only a pai interrupts.

IN THEORY

In theory, reading an encoder is a piece of cake. There are two outputs, we'll call them A and E turn the knob, these outputs send out pulses. The mechanical arrangement inside is such that knob is turning in one direction, pulses from A are 90 degrees ahead of the pulses from B. If yc other way, the phase is reversed.



People usually think of the pulses as going positive, but most real encoders will have a contact and a pull up resistor, so actually, the outputs are often high when nothing is happening and th really low pulses. You can see that in the diagram when no one is turning the knob, there is a lo high signal.

Note on the left side of the diagram that the B signal drops before the A signal every time. If yc

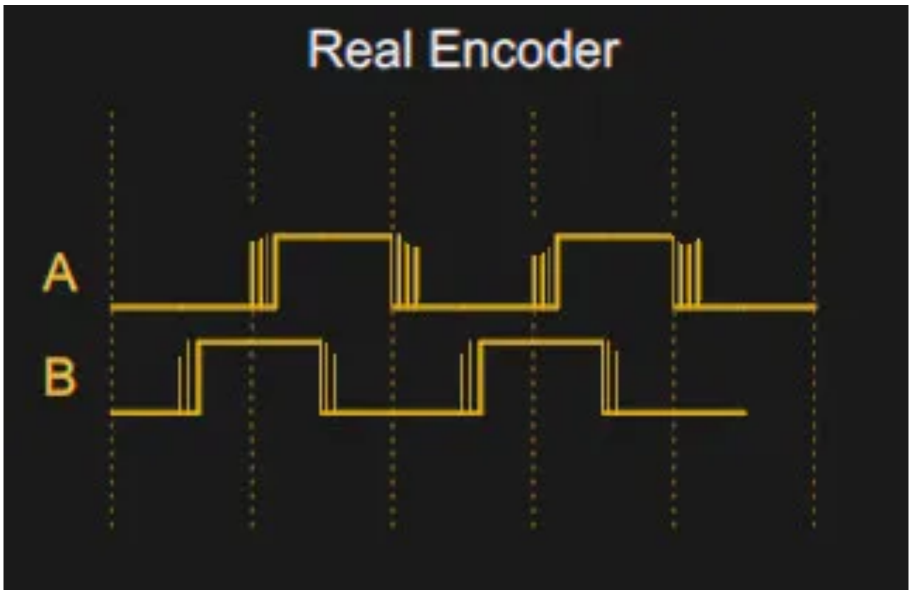
at the falling edge of A, you will always get a 0 in this case. The width of the pulses depends o
you turn, of course. When you turn the other way, you get the case on the right side of the diag
the A signal goes low first. If you sample at the same point, B is now 1.

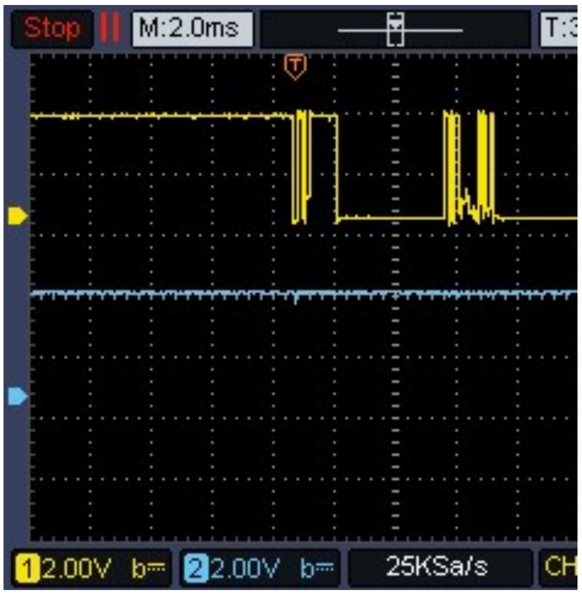
Note there is nothing magic about A, B, or the clockwise and counterclockwise labels. All it rea
"one way" and "the other way." If you don't like how the encoder is moving you can just swap /
swap it in software. I just picked those directions arbitrarily. Usually, channel A is supposed to "
clockwise direction, but it also depends on the edge you measure and how you connect every
software, you generally add one to a count for one direction and subtract for the other directio
idea of where you are over time.

There are many ways you can read this sort of input. If you are sampling it, it is pretty easy to b
machine from the two bits and process it that way. The output forms a gray code so you can th
bad states and bad state transitions. However, if you are sure about your input signal, it can be
than that. Just read B on one edge of A (or vice versa). You could verify the other edge if you w
more robustness.

IN PRACTICE

Unfortunately, real mechanical encoders don't look like the above diagram. They look more lik





This leads to a problem. If you are interrupting on both edges of input A (the upper trace on the
will get a series of pulses at both edges. Notice that B is in different states at each edge of A, s
an even number of pulses in total, your total count will be zero. If you are lucky, you might get a
number in the right direction. Or you might get the wrong direction. What a mess.

But on the sampling edge of A, B is rock solid. The lower trace on the scope looks like a straigl
because all the B transitions are off the screen at that scale. That's the secret to easily deboun

encoder. When A is changing, B is stable and vice versa. Since it is a gray code, that makes sen
the insight that makes a simple decoder possible.

THE PLAN

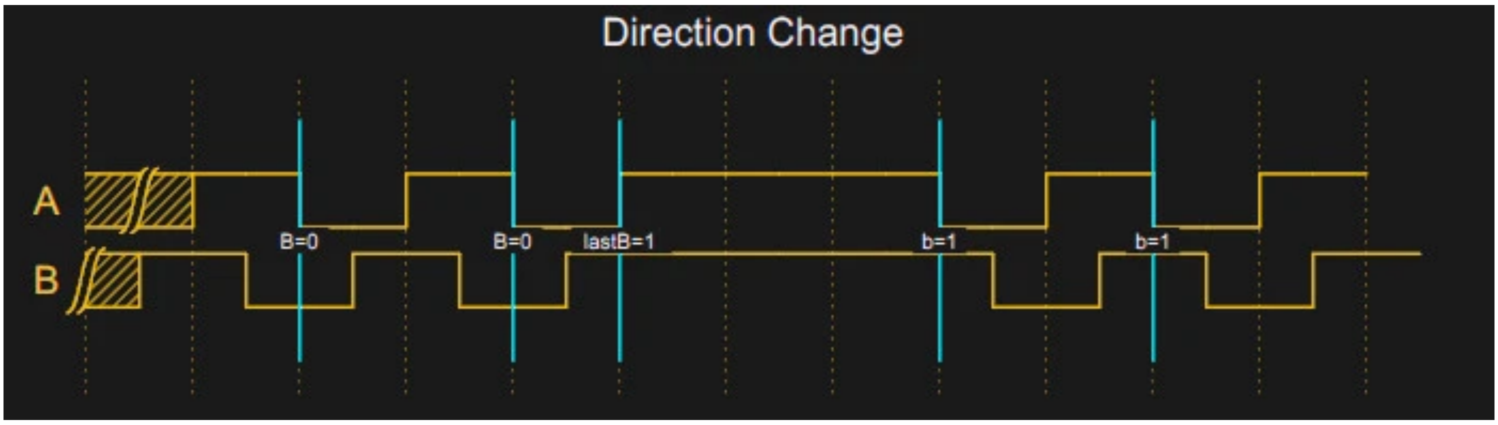
So the plan is to notice when A goes from high to low and then read B. Then ignore A until B c
you want to monitor B, of course, it has the same problem so you have to lock it to A which is s
change. In my case, I didn't want to use two more interrupts so I follow this logic:

1. When A falls, record the state of B and update the count. Then set a lock flag
2. If A falls again, if the lock flag is set or B has not changed, do nothing.
3. When A rises, if B has changed, record the state of B and clear the lock flag.

That means in the scope trace above, the first dip in the top trace causes us to read B. After tha
the transitions on the screen will have any effect because B has not changed. The rising edge
screen that occurs after B has had a noisy high to low transition will be the one that unlocks the

THE PROBLEM

There is a problem, though. The whole scheme relies on the idea that B will be different on a tr
edge for A compared to a falling edge. There is one case where B doesn't change but we still w
accept the A edge. That's when you change directions. If you monitored B, that would be easy
that's more code and two more interrupts. Instead, I decided that for a person twisting a knob,
twist in different directions very quickly, you won't even notice that one or two clicks of the enc
the wrong way. What you will notice is if you make a fine adjustment and then twist the other w
deliberately.



When you think you know the previous state of B and nothing has changed in a while (like a fe
milliseconds) the code will reset its idea of B to unknown so that the next B signal will be consi
no matter what.

I used the `Kernel::Clock::now` feature from Mbed. It isn't clear if you are supposed to cal
interrupt service routine (ISR), but I am and it seems to work without problems.

The only other issue is to make sure the count doesn't change in the middle of reading it. I disa
interrupts around the read just to make sure.

THE CODE

You can find the code on GitHub. If you made it through all the explanations, you should have r
following along.

```
void Encoder::isrRisingA()
{
   int b=BPin; // read B
   if (lock && lastB==b) return; // not time to unlock
// if lock=0 and _lastB==b these two lines do nothing
// but if lock is 1 and/or _lastB!=b then one of them does som
   lock=0;
   lastB=b;
   locktime=Kernel::Clock::now()+locktime0; // even if not loc
}

// The falling edge is where we do the count
// Note that if you pause a bit, the lock will expire because
// we have to monitor B also to know if a change in direction
// It is tempting to try to mutually lock/unlock the ISRs, but
// the edges are followed by a bunch of bounce edges while B i
// B will change while A is stable
// So unless you want to also watch B against A, you have to m
// compromise and this works well enough in practice
void Encoder::isrFallingA()
{
   int b;
   // clear lock if timedout and in either case forget lastB i
   if (locktime<Kernel::Clock::now())
     {
     lock=0;
     lastB=2; // impossible value so we must read this event
     }
   if (lock) return; // we are locked so done
   b=BPin; // read B
   if (b==lastB) return; // no change in B
   lock=1; // don't read the upcoming bounces
   locktime=Kernel::Clock::now()+locktime0; // set up timeout
   lastB=b; // remember where B is now
   accum+=(b?-1:1); // finally, do the count!
}
```

Setting up the interrupt is easy because of the `InterruptIn` class. This is like a `DigitalIn`
has a way to attach a function to the rising or falling edge. In this case, we use both.

LATENCY

I wondered how much time it took to process an interrupt on this setup, so that code is availab
`#define TEST_LATENCY 1`. You can see a video of my results, but TLDR: It took no more th
microseconds to get an interrupt and often about half of that.

Getting the encoder right was a little harder than I thought it would be, but mostly because I di
process more interrupts. It would be simple enough to modify the code to watch the B pin rela
pin and have a true understanding of the correct state of B. If you try that modification, here's a
by measuring the time between interrupts, you could also get an idea of how fast the encoder
which might be useful for some applications.

If you want a refresher on gray code and some of where it is useful, we've talked about it befor
sounds oddly familiar, I used an encoder on an old version of Mbed in 2017. In that case, I used
library that periodically polled the inputs on a timer interrupt. But like I say, there's always more
way to make stuff like this happen.

[Headline image: “Rotary Encoder” by SparkFunElectronics, CC BY 2.0. Awesome.]

Posted in ARM, Hackaday Columns, Microcontrollers
Tagged encoder, mbed, quadrature encoder, rotary encoder

← THIS LAPTOP GETS ALL THE PCIE DEVICES

10 THOUGHTS ON “A ROTARY ENCO HOW HARD CAN IT BE?”

jpa says:
April 20, 2022 at 10:08 am

The Gray code aspect of encoders means that bouncing doesn’t really matter, as long as you han reading correctly.

For interrupt driven approach, read the pin states at beginning of interrupt and compare it against state you stored at previous interrupt. Ignore the interrupt flags, they are important only for getting the interrupt. After you have processed the change, store the state you already read to be used n time. By only reading the state once per interrupt, no matter how many edges you have, the total will be correct – any bounces will just change it +- 1.

On STM32 though, you are best off using the encoder mode on the timers. That avoids any of the interrupt overhead and you can just read a 16-bit count out of the timer count register. It even has hardware filter for the inputs if it seems necessary.

Reply Rep

Artenz says:
April 20, 2022 at 10:10 am

I've always done this with a 1 kHz timer interrupt and a simple state machine. If ISR takes 1 usec, it results in 0.1% CPU load. State machine waits for 00->11 or 11->00 transitions, and then checks whi was the last one to change.

Reply Rep

Daid says:
April 20, 2022 at 10:39 am

Polling is the sane method indeed. It gives you a predictable system, easier to debug and reas about.

Reply Report

paulvdh says:
April 20, 2022 at 10:18 am

Putting RC circuits on these cheap mechanical encoders is a common way to tame them.
Don't just put (ceramic) capacitors on the outputs, as these can deliver very high peak currents (1(can destroy the switch contacts quickly.

Also:
most (all?) STM32 chips have hardware to keep track of the encoder count of quadrature encode hardware. You just program it once, and then you read the hardware timer register whenever you doing so. The hardware is built for high speed motor encoders, but it should work on a simple ma knob too.

Reply Rep

Mike says:
April 20, 2022 at 10:30 am

” Don’t just put (ceramic) capacitors on the outputs, as these can deliver very high peak curren (10A+) and can destroy the switch contacts quickly. ”

10A+, LOL…. Ya maybe if your sticking in a 100uf cap. A simple 200 Ohm on the common pin, a then a 0.1uf-0.2uf cap on its outputs will give you a nice clean transition.

Reply Report

fiddlingjunky says:
April 20, 2022 at 10:45 am

From the STM32F411 datasheet:

“TIM2, TIM3, TIM4, TIM5 all have independent DMA request generation. They are capable of handling quadrature (incremental) encoder signals and the digital outputs from 1 to 4 hall-effec sensors.”

The reference manual also gives a tidy example of which registers to set. I’m a huge fan of rea datasheets and using appropriate hardware functions. Good catch.

Reply Report

fonz says:
April 20, 2022 at 10:46 am

yep, the STM32F411 I’m guessing he’s using can do at least 4 encoders (maybe 6) using timers

https://github.com/langwadt/nucleo_quad4

Reply Report

Electronic Eel says:
April 20, 2022 at 10:34 am

Contact chatter or bouncing does not go well with interrupts: a big number of interrupts will be tri in short order during bouncing. This often means the rest of the tasks the MCU has to handle can and their throughput is significantly reduced compared to times without contact bounce.

So my suggestion is to use interrupts only to detect an initial change of the inputs. Then disable tl interrupt for some time and change over to a time based solution, for example via virtual times tha RTOS offers. Only when the timer handler doesn't detect a change for some time you switch back input interrupts. This way the input interrupts can't overwhelm the rest of the system.

Reply Rep

Artenz says:
April 20, 2022 at 10:42 am

That's why I just poll the switches in a timer interrupt. It avoids all this extra complexity. If your system can't handle a periodic timer, it can't handle fast bouncing contacts.

Reply Report

mrehorst says:
April 20, 2022 at 10:47 am

I did some PIC programming in assembly language back before Arduino became a thing. The way handled reading encoders was an interrupt triggered subroutine with a debounce delay and then up table. The table returned an increment or decrement value. It required minimal code and was s fast and could easily keep up with and speed you could manually spin the encoder. A few more lii code could make it speed sensitive so the increment/decrement value varied with the speed of e rotation.

Reply Rep

Leave a Reply

Enter your comment here...

Please be kind and respectful to help make the comments section excellent. (Comment Policy)

This site uses Akismet to reduce spam. Learn how your comment data is processed.

HOME
BLOG
HACKADAY.IO
TINDIE
HACKADAY PRIZE
VIDEO
SUBMIT A TIP
ABOUT
CONTACT US

NEVER MISS A HACK