# Using Interrupts for Accurate Time Measurements

In this article we'll look at how to use an interrupt from a digital pin on an Arduino board to make accurate time interval measurements.

Let's say you have a rotating shaft or cylinder and you'd like to know how fast it's going around. This might be part of a motor control program and we want the shaft RPM as part of the input into the control loop.

We can sense the rotation in many ways. The two I find easiest are: using a hall-effect switch with some pieces of magnet on the shaft, and using a light sensor with some reflective strips on the shaft. I think the hall-effect switch is the easiest as it is immune to ambient lighting. You can cut up a flexible fridge magnet (the plastic kind) and tape a couple of pieces to the shaft. I usually use two pieces to keep the balance of the shaft as it rotates. The Texas Instruments DRV5023 sensor is very easy to use with an Arduino board. It takes a 5V supply and ground on two pins and provides an output on the third that you can connect directly to the Arduino with the input pin set up to have the internal pullup resistor.

Once you have the hardware set up and connected to an Arduino digital input pin, how do you measure how fast the shaft is rotating?

The simplest way is to use calls to digitalRead to wait for the signal from the pin to go low indicating the magnet or reflective strip just went by, then timing how long it takes until the signal goes low a second time. We can use the *micros* function call to get an accurate time in microseconds. (Hang in there - we'll see the code in minute).

If we have two sensors on the shaft then we probably want to average every two readings to deal with the placement of the magnets on the shaft not being exact, but I've ignored that problem here to keep the example code shorter.

If your app does nothing else other than watch one digital input pin, it can measure the rotation interval reasonably accurately. The reality is usually different, however, because your app probably has other things to do and that likely means you won't be able to take the shaft interval measurements as often as you like. In addition, while you are waiting for the input pin to changes state, you're not doing all the other things your app needs to be doing.

The code you might put inside your loop function could look like this:

// Wait for the input to go high

while (digitalRead(FG\_INPUT\_PIN) == LOW);

// Wait for the input to go low

while (digitalRead(FG\_INPUT\_PIN) == HIGH);

// start the timer

uint32\_t start = micros();

// Wait for the input to go high

while (digitalRead(FG\_INPUT\_PIN) == LOW);

// Wait for the input to go low again

while (digitalRead(FG\_INPUT\_PIN) == HIGH);

// measure the elapsed time

uint32\_t interval = micros() - start;

If we run this code for a while and keep track of the measurements it takes, we will find that although it's reasonably good, other things going on in the program (like the Arduino's 1 millisecond timer interrupt or other interrupts) affect the accuracy. The example code shows how to measure the mean and variance of the data, and look at the time distribution of the measurements by putting them into bins. (Oh no, he's getting into statistics).

The example code uses a PWM generator in the processor to put out a squarewave with an interval of 1,024 milliseconds (a bit under 1 kHz). That output is then connected to the input pin we're sampling. I used that to provide an input for the following sets of measurements.

Running the app shows you something like this after a couple of hours:

Foreground: Mean: 1025.27 us, Variance: 118.24

0 0 46 228 4634967 205 54 0 0 0

That's a fairly normal looking distribution and perhaps the variance isn't too bad, but we can do better. If we are using only one or two samples to provide input to a closed loop motor control algorithm we'd like to be sure that the measurements are not just statistically good over a long period, but are good each time we take a measurement, otherwise we're going to have to take the measurement jitter into account in the algorithm.

OK, so that introduction took a long time, but let's look now at how easy it is to make the same measurement more accurately using an interrupt to control the time measurement.

I'm not going to describe the ins and outs of interrupts here other than to say that an interrupt can be handled by a special piece of code called an interrupt service routine (ISR) that will run every time our sensor pin changes state. If you want to know more, you can read a really detailed article about Arduino interrupts by Nick Gammon here: <http://gammon.com.au/interrupts>.

Before we look at the code, let's see the results:

Foreground: Mean: 1025.27 us, Variance: 118.24

0 0 46 228 4634967 205 54 0 0 0

Background: Mean: 1024.00 us, Variance: 6.32

0 2632 2613 2292 9391799 2292 2613 2632 0 0

This is the same set of samples we looked at just now but it shows both the data measured in the application's *loop* function (the foreground code), together with the data measured by the ISR (the background code). Notice that although there is still some variation in the samples, the variance is much lower for the ISR (background) case.

The code for computing the variance is in the example. The streaming method was taken from this Wikipedia article: <https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance>

Let's look now at how the code is set up to make the measurements in the ISR. I'm going to show just the important lines from the example, not all of it so we can focus on what's specific to the interrupt setup and handling.

It's important to know that the ATmega processors used in the Arduino boards have only certain pins that can be used to generate inputs, and not all of those pins are available on every board. The Arduino Uno family only supports this type of interrupt on pins 2 and 3. Read Nick Gammon's article, or the Arduino website for more information about the other pin interrupts that are available.

The pin we are going to sample is just set up as a normal input pin:

pinMode(BG\_INPUT\_PIN, INPUT);

For this example I chose to have two different input pins. One is for the foreground code and the other for ISR to monitor separately so it's easy to see how the two different code methods work. To run the app as I did, you just connect the PWM test signal that is generated on pin 6 to the inputs on pins 2 and 3.

To get our pin (pin 2) to generate an interrupt each time it goes low we tell the Arduino code to attach an ISR function (that we create) to it like this:

attachInterrupt(digitalPinToInterrupt(2), myISR, FALLING);

The *digitalPinToInterrupt* call translates our pin number (2 in this case) into the interrupt number that the ATmega processor uses. *myISR* is our handler code that we'll look at next, and FALLING means we want to trigger the interrupt when the input signal goes from high to low.

Now we can look at the code that handles the interrupt. All ISRs have this signature:

void someIsrName()

{

// your code

}

The ISR has no input arguments and returns nothing, which means that in order to see the results of our labor we'll need to store the results in some global variable that we can access in our normal foreground code. There are a few rules we need to follow when we do this. When we try to get the result later, we are going to read a variable that potentially can change right as we are reading it - that's the cool thing about interrupts - they happen right at the hardware event time (more or less). So any code that retrieves data from an ISR must temporarily halt interrupts while it fetches the data, then allow them again as soon as possible. The ONLY exception to this is a special case where the data is just one byte (8 bits) because reading one memory location can't be affected by an interrupt. The interrupt can change the data before we read it, or after, but not during. If we keep data in ints, longs, or structures of some kind as I do in the example code, then we must hold off interrupts as we read those variables or we risk getting half and old value and half a new value.

Here is the complete ISR code. :

// variable to keep track of the previous ISR time

uint32\_t g\_prev\_edge\_time = 0;

void myISR()

{

// get the time

uint32\_t edge\_time = micros();

// We toggle an output pin here that we can see it on the scope.

digitalWrite(ISR\_MONITOR\_PIN, HIGH);

if (g\_prev\_edge\_time != 0) {

// compute the elapsed time since the previous input edge

uint32\_t elapsed = edge\_time - g\_prev\_edge\_time;

// update the computation

g\_bgVar.update(elapsed);

}

g\_prev\_edge\_time = edge\_time;

// toggle the scope pin again just before we exit the ISR

digitalWrite(ISR\_MONITOR\_PIN, LOW);

}

This needs some explanation. There are two *digitalWrite* calls in here to pulse an output pin so that I can see when the ISR runs and how long it takes by looking at that signal on an oscilloscope. If you don't have a scope, or don't need that information, you can just delete those lines. By the way, it's perfectly OK to use *digitalRead* or *digitalWrite* calls inside an ISR.

We keep track of the last time the ISR ran in a global variable: *g\_prev\_edge\_time*. The ISR code doesn't try to compute anything the first time around as we don't have a valid interval to compute from. For each subsequent interrupt we compute how long it was since the previous one, and use that to update the statistics object.

The stats object class that computes the mean and variance was written from the Wikipedia article example code, so read that if you can't follow how the mean and standard deviation are computed as we get streamed data in the ISR.

In the functions used to retrieve data from the stats object like *getVariance* you'll see the use of the AVR macro ATOMIC\_BLOCK. This guarantees that the block of code that it's 'guarding' won't be interrupted. These blocks need to be as short as possible since they prevent all interrupts from happening.

So we see that using an interrupt to monitor an external even is quite easy if we follow a few rules. The interrupt technique results in more accurate results than doing it in the application's main foreground code.

Happy coding.