Pin Change Interrupt Timing

an exploration of latency and design

by: GreyGnome (aka, Mike Schwager)

Feb. 8, 2015 v. 1.0

Table of Contents

Introduction

I have a "dirty" switch: it bounces a lot when pressed. The bounces can occur very quickly, on the order of only microseconds apart. This switch is attached to an Arduino Duemilanove, based on the ATmega328p processor. The pin change interrupts on the ATmega chips act only on change of signal (either rising or falling), and they trigger when any pin on a port (== up to 8 pins) is triggered. If you want to determine which pin actually triggered, that would take software and that means a delay from the moment of signal change to determining which pin triggered the change. With a pin change interrupt enabled to trigger on the pin's port, how will a library designed for general use- which must use an algorithm to find the proper pin- react when this switch is pressed?

It bears repeating that for Pin Change Interrupts, the interrupt can take place when *any* pin on the port is interrupted. This presents a serious design challenge. If you have the luxury of knowing at the time of the signal which pin interrupted, you can design fast, custom code that will react to your simple situation. But remember that we are talking about a library: we don't know which pin may be doing the interrupting. So it must survey the pins to figure out which one(s) changed and had triggered the interrupt. Furthermore, there is an appreciable amount of time that it takes from the moment the triggering event happened to when we enter the interrupt subroutine (ISR) and go through the logic to figure out which pin did the triggering. ...How much time? That I aim to find out.

Why is this a big deal? Think of a bouncy switch: the interrupt triggers, the ISR starts up, and the first thing we need to do is query the port to see the state of its pins. Well, some time has elapsed since the triggering event and the query. In the course of that time, it's entirely possible- and I'm writing this because it's not only possible, but it can happen quite readily- that the state of the pin changes before we get a chance to sample it. So we get an interrupt but it looks like a false alarm! The ISR never calls the user's function because none of the user's interrupt pins appear to have changed.

The Delay

There is no complete solution to this problem, because of the nature of Pin Change Interrupts and the nature of a general purpose Interrupt SubRoutine (ISR). All you can do is mitigate the situation. I will attempt to do so by capturing the state of the port as early as possible in the ISR. The question is, how early is that?

No matter what, it will take some number of machine instructions from the time of the signal to actually executing the ISR: The current instruction is completed, and there is a jump to the ISR code which takes 3 clock cycles. So that's at least 4 clock cycles, which is about 250 nanoseconds on a 16MHz machine (4 * 62.5 ns). Best theoretical case, then, is that after our signal triggers an interrupt it will take us 250 ns to sample the pin's state. But there's much more than this.

A General-Purpose ISR Must Save Registers

An interrupt is by definition asynchronous- the computer never knows when it's going to happen. And the compiler writers are not able to know the myriad types of logic that will be necessary inside an interrupt. Thus they can never know which registers may be necessary for a user's ISR, and the implications of that are pretty significant. It means that the first thing the ISR should do is save the values of each and every register of the CPU. ...But isn't that expensive, time-wise? Why yes, yes it is. How so? Here is what the assembly looks like at the beginning of a compiler-generated ISR; note the proper preamble to the ISR pushes all registers (there are reasons registers r2-r17 are not saved for reasons that are not clear to me, but the assembly generated by the compiler for the ISR doesn't use them):

```
ISR(PORTC_VECT) {
     292:       1f 92           push    r1
     294:       0f 92           push    r0
     296:       0f b6           in      r0, 0x3f        ; 63
     298:       0f 92           push    r0
```



```
     29a:       11 24           eor     r1, r1
     29c:       2f 93           push    r18
     29e:       3f 93           push    r19
     2a0:       4f 93           push    r20
     2a2:       5f 93           push    r21
     2a4:       6f 93           push    r22
     2a6:       7f 93           push    r23
     2a8:       8f 93           push    r24
     2aa:       9f 93           push    r25
     2ac:       af 93           push    r26
     2ae:       bf 93           push    r27
     2b0:       cf 93           push    r28
     2b2:       df 93           push    r29
     2b4:       ef 93           push    r30
     2b6:       ff 93           push    r31
```

The ISR can make no assumptions: What was the state of the CPU when the interrupt took place? What registers are in use by a user's function when called from within the ISR? The compiler is pretty sophisticated, but to ask it to optimize so as to follow a trail of function calls so as to shorten an ISR preamble is beyond its capabilities.

The trouble is in the ISR's function call. In order to allow the library to accept a function call like

```
enableInterrupt(Pin, function, mode);
```

where the function's name may not be known at compile time, the ISR must be ready for any eventuality. To not do so means risking bugs by not preserving the main program's registers.

A Small ISR, for Comparison

So we need our massive preamble. What about an ISR that does not have such restrictions? What if our ISR was simple, and its needs were known to the compiler? To see, I create a small ISR in C:

```
volatile uint8_t pinstate;
ISR(PORTC_VECT) {
  pinstate=PINC;
}
```

This ISR simply copies the state of Port C to the volatile pinstate variable, which is a memory address accessible by the main program or any function in our sketch. Here is the assembly language that the compiler generates from this C:

```
volatile uint8_t pinstate;

ISR(PORTC_VECT) {
 212:   1f 92           push    r1
 214:   0f 92           push    r0
 216:   0f b6           in      r0, 0x3f        ; 63
 218:   0f 92           push    r0
 21a:   11 24           eor     r1, r1
 21c:   8f 93           push    r24
  pinstate=PINC;
 21e:   86 b1           in      r24, 0x06       ; 6
 220:   80 93 60 01     sts     0x0160, r24
}
 224:   8f 91           pop     r24
```



```
 226:   0f 90           pop     r0
 228:   0f be           out     0x3f, r0        ; 63
 22a:   0f 90           pop     r0
 22c:   1f 90           pop     r1
 22e:   18 95           reti
```

Note that it's much shorter, especially the preamble and the postamble. Much of the code is there because of ISR boilerplate actions:

```
      push    r1              Save whatever may be in r1
      push    r0              Save whatever may be in r0
      in      r0, 0x3f        Store SREG into r0
      push    r0              Save r0
      eor     r1, r1          Xor r1 with itself, which is a clever way to put '0'
                              in r1
```

How efficient is it? Well we don't need SREG, because the read of port C: `in r24, 0x06`, and the store of it in memory using the `sts 0x0160, r24` don't modify SREG. We don't need to zero out r1, because we never use r1. So all the `push`'ing and `pop`'ing are useless in this case, and serve merely to slow us down. Such is life with a compiler vs. hand crafted assembly code. It may be useful to create a hand-crafted ISR in this case.

Testing Speed

Entering the ISR

I attempt a test:

A switch is connected as follows: A pin is configured as an input port, and the pullup resistor is on. Pin Change Interrupt is enabled on the pin (which will trigger on any level change). So when I press the switch, the signal goes from high to low and the interrupt is triggered.

My ISR looks (in part) like this; this will turn on and off the Arduino Uno's pin 13 LED:

```
ISR(PORTC_VECT, ISR_NAKED) {
  uint8_t interruptMask;
  uint8_t ledon, ledoff;

  ledon=0b00100000; ledoff=0b0;

  PORTB=ledoff; // LOW
  PORTB=ledon;         // HIGH
  PORTB=ledoff; // LOW
  PORTB=ledon;         // HIGH
  PORTB=ledoff; // LOW
  (...)
}
```

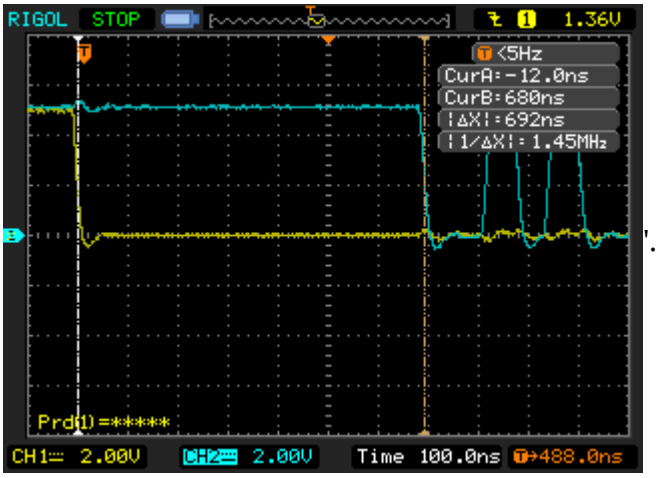
The generated assembly code looks like this:

```
00000292 <__vector_4>:
```



```
ledon=0b00100000; ledoff=0b0;

PORTB=ledoff; // LOW
   292:       15 b8           out     0x05, r1        ; 5
PORTB=ledon;          // HIGH
   294:       80 e2           ldi     r24, 0x20       ; 32
   296:       85 b9           out     0x05, r24       ; 5
PORTB=ledoff; // LOW
   298:       15 b8           out     0x05, r1        ; 5
PORTB=ledon;          // HIGH
   29a:       85 b9           out     0x05, r24       ; 5
PORTB=ledoff; // LOW
   29c:       15 b8           out     0x05, r1        ; 5
```

Notice a little optimization here: r1 is defined to always contain 0, so we don't even have to load a value from memory (0 is an important number!). This makes the first `out 0x05, r1` command very quick, and by using an oscilloscope we can see just how quickly the chip reacts after receiving the signal:



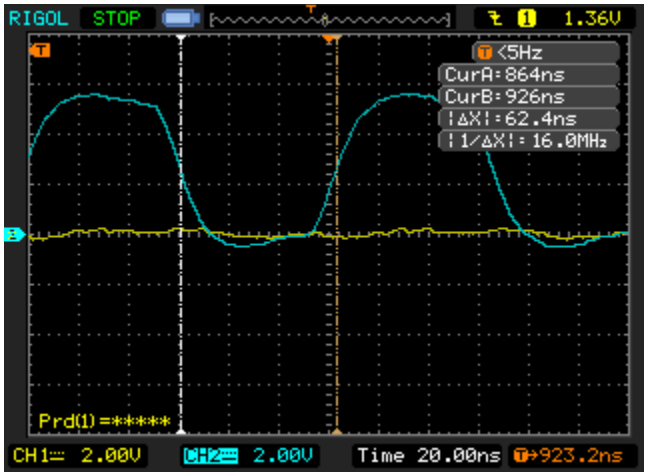
Look at the two vertical lines on the left and center-right in the above screenshot- those are the cursors. ...So we are measuring 700 nanoseconds from when my switch first closed its connection (yellow trace) to where pin 13 went LOW (blue trace). The cursor lines tell the tale.

How much of that 700 nanoseconds was taken up by the first command itself, the one that signals pin 13? That is:

```
292:       15 b8           out     0x05, r1        ; 5
```

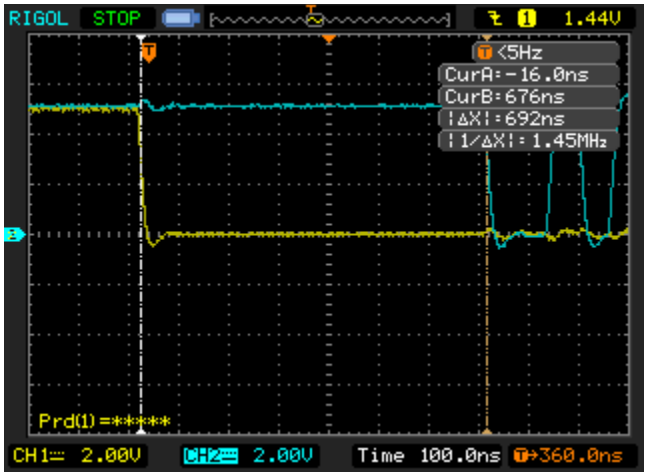
In the blue trace, you can see that after I go low I go high, then low, then high again. The second dip- the shorter dip- above, should tell the tale. And it says:

(note that here I have zoomed in for a closer, more accurate look)



...Lo and behold, the trough- the `PORTB=ledoff;` is 62.4ns wide, which is equivalent to a 16.0MHz frequency, which is a single clock cycle of the ATmega328p. Note that the data sheet states that the OUT assembly instruction takes 1 cycle. Yay, the stars align!

Now- going back to the first picture: How long did it take to enter the ISR? My experiment measures the time from the switch close to the Pin13 going LOW. We can subtract the time to execute that first command, and get to exactly how long it takes us to enter the ISR:



...That's 692ns measured, or approximately 11 cycles. Minus the 1 cycle that it took our command to run, it takes 10 cycles' time from interrupt to entering the ISR.

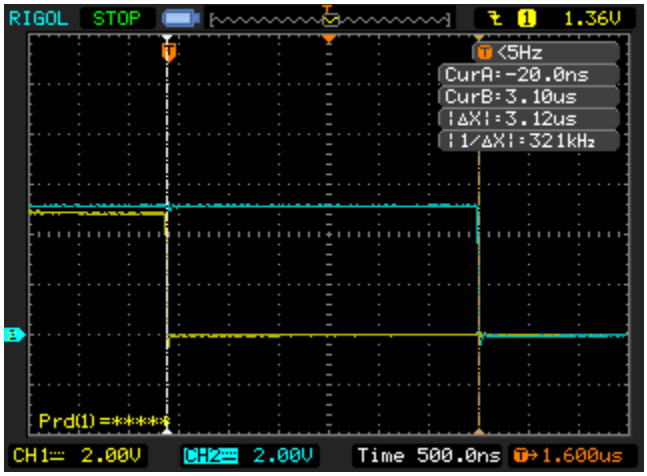
Thus, the best response to an interrupt that we could ever hope for is 630ns, more or less. That's at least 630 ns between activation and having a ghost of a chance to query the pins and store their value.

The Preamble

BUT! Can we just stuff an “in” command at the beginning of the ISR, in order to grab the port's pins' values? Remember, this is a general purpose ISR so the compiler inserts a preamble and a postamble. Here is what the assembly looks like when we don't use NAKED_ISR; note the proper preamble to the ISR (there are reasons registers r2-r17 are not saved; suffice it to say they are scratch and their values are not guaranteed):

```
ISR(PORTC_VECT) {
     292:       1f 92           push    r1
     294:       0f 92           push    r0
     296:       0f b6           in      r0, 0x3f        ; 63
     298:       0f 92           push    r0
     29a:       11 24           eor     r1, r1
     29c:       2f 93           push    r18
     29e:       3f 93           push    r19
     2a0:       4f 93           push    r20
     2a2:       5f 93           push    r21
     2a4:       6f 93           push    r22
     2a6:       7f 93           push    r23
     2a8:       8f 93           push    r24
     2aa:       9f 93           push    r25
     2ac:       af 93           push    r26
     2ae:       bf 93           push    r27
     2b0:       cf 93           push    r28
     2b2:       df 93           push    r29
     2b4:       ef 93           push    r30
     2b6:       ff 93           push    r31
  uint8_t ledon, ledoff;

  ledon=0b00100000; ledoff=0b0;

  PORTB=ledoff; // LOW
     2b8:       15 b8           out     0x05, r1        ; 5
```

Now let's measure the time from interrupt to shutting off the Pin 13 LED again in that `out 0x05, r1` command, just like before. It's given here:



...yikes! 3.12 microseconds! That's 50 times as long as 62.5 nanoseconds... which means, 50 machine cycles! Well, we do have 17 “push” instructions alone- at 2 cycles each, that's 32 * 62.5ns right there.

This means it would be 3.12 microseconds- at least- from a triggering event, to querying the pins of the PORT, to see which pin triggered. That's painful. Is there anything we can do?

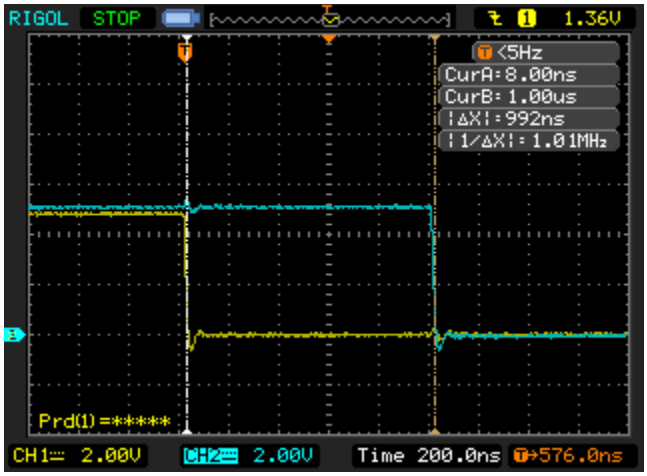
Query the Port by Hand

There is something: by hand-coding the preamble and post-amble, grab the contents of the port as soon as possible. Here's the code:

```
ISR(PORTC_VECT, ISR_NAKED) {
  uint8_t current;
  uint8_t ledon, ledoff;

  ledon=0b00100000; ledoff=0b0;

  asm volatile("\t"
  "push %0" "\t\n\t"
  "in %0,%1" "\t\n\t"
  : "=&r" (current)
  : "I" (_SFR_IO_ADDR(PINC))
  );
  PORTB=ledoff; // LOW

  // in r0, __SREG__ instruction saves SREG, then it's pushed onto the stack.
  asm volatile(
  "push r1" "\n\t"
  "push r0" "\n\t"
  "in r0, __SREG__" "\n\t" // 0x3f
  "push r0" "\n\t"
  "eor r1, r1" "\n\t"
  "push r18" "\n\t"
  "push r19" "\n\t"
  "push r20" "\n\t"
  "push r22" "\n\t"
```

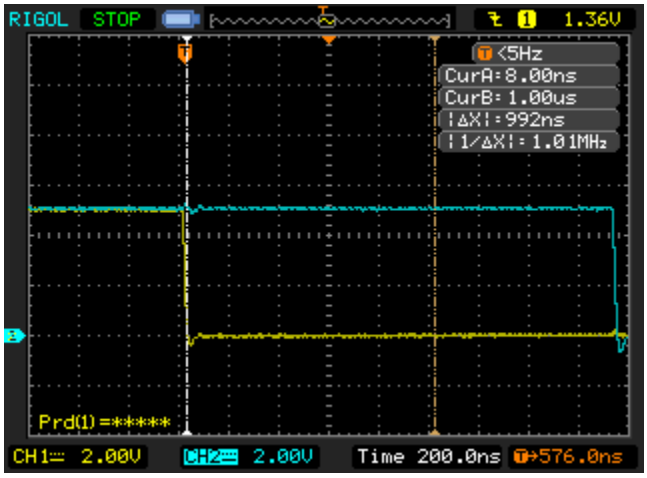
By saving our current PORT state into the variable “current” at the very start of the ISR, using assembly language, we are able to get it as quickly as practicable. Note, however, that because of an SR's special nature we can not be sure if the main code was using the register that will be used for “current” in the “%0” placeholder in the code: “`in %0,%1`”, above. Therefore, we must push it first... and take care to pop it last at the end of the ISR.

```
  "push %0" "\t\n\t"
  "in %0,%1" "\t\n\t"
```

Now that we have captured our PORT state as soon as possible, we ask: How well did we do? The screen says it took 992 ns... that's about 300 ns slower than a naked ISR preamble, or about 5 cycles. That's close to what we would expect from the assembly code: 2 cycles for the push, 1 for the in, and 1 for the out (to make the LED line go LOW). 4 instructions, or an expected 248ns slower than the routine with the `PORTB=ledoff;` command first.

...But we're still 62 ns off. What gives? According to my tests, over the course of many runs, there could be some jitter in our interrupt timing. As a matter of fact, there are some significantly longer periods than the one just shown. Take this run for example:



Here, the two cursors are in exactly the same positions as the previous screenshot.This time it took almost 800ns longer to respond to the interrupt.

This sort of thing happened infrequently, but over the course of a few runs one could find an outlier like this. My feeling is that sometimes the CPU may be in, for example, a timer interrupt and it is unable to service this interrupt until it's done. That's something to keep in mind.

Ultimately, the Pin Change Interrupts are a bit of an oddity. A single interrupt is called for any pin on a port that is triggered, and one needs software if one wants to determine which pin activated. Even then, there's no guarantee that the pin is in the same state that it was when the interrupt was triggered! Caveat

programmer- you must be aware of of the capabilities of the hardware that you're working with, and of your system's own performance needs.

That said, I have used a relatively slow Pin Change Interrupt library with great success (http://code.google.com/p/arduino-pinchangeint/). Most cheap switches, for example, are quite bouncy so it's necessary to account for a certain degree of slop in the real world. This is what engineers are paid to do: Make components and systems work in the real world.

And that, my friends, *is* what they call Rocket Science: It ain't easy.

A Real Life Example of a Bouncy Switch

Tracing the Interrupts' Behavior

How well is the ISR grabbing the proper pin state on a fast-changing signal? “Fast” is relative, but for an Arduino running at 16 MHz I would say that fast is anything in the microsecond range. The Arduino's clock runs at a period of 62.5ns which is one machine instruction. In the space of 1 microsecond 16 (1-cycle) machine instructions can run... that's not a whole lot.

In this test, the interrupt library does the following:

1. As soon as we've saved the current PORT state, send out a single pulse.
2. As soon as we've found the pin that on the port that we believe triggered the interrupt, send out a double pulse.

The interesting code parts are these:

```
ISR(PORTC_VECT, ISR_NAKED) {
  uint8_t current;
  uint8_t i;
  uint8_t interruptMask;
  uint8_t ledon, ledoff;

  ledon=0b00100000; ledoff=0b0;

  asm volatile("\t"
  "push %0" "\t\n\t"
  "in %0,%1" "\t\n\t"
  : "=&r" (current)
  : "I" (_SFR_IO_ADDR(PINC))
  );
  PORTB=ledoff; // LOW
  PORTB=ledon; // HIGH
```

Then:

```
    if (interruptMask & 0x01) {
```

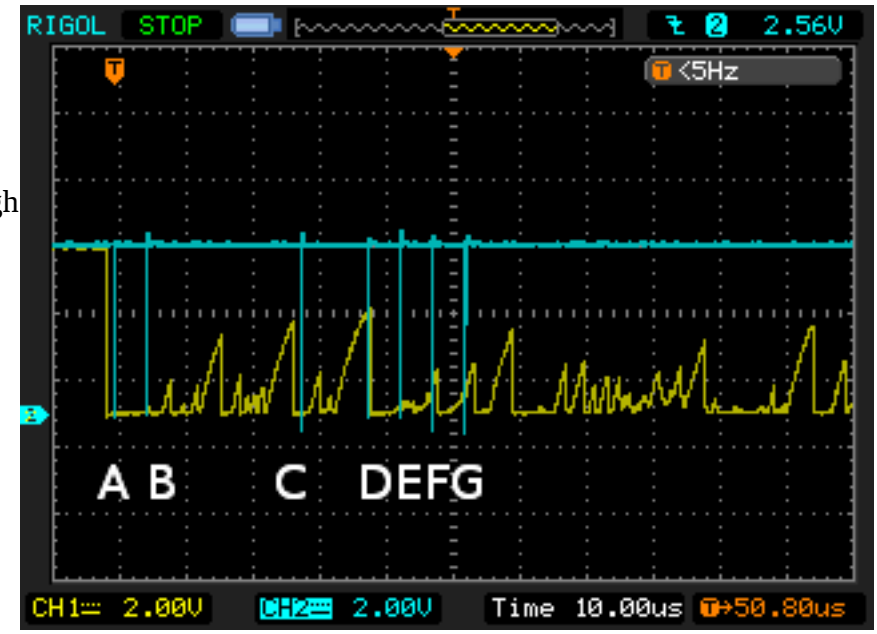


```
PORTB=ledoff; // LOW
PORTB=ledon; // HIGH
PORTB=ledoff; // LOW
PORTB=ledon; // HIGH
```

Now we can see when the interrupt triggers and when it thinks the pin state has changed. How does this line up against our bouncy switch? Are we properly reacting to all the activations?

Here is a real oscilloscope trace of a bouncy switch and of the Interrupt library reacting:

In this trace, the actions taken by the interrupt library are labelled A through G, in order to easily follow the sequence of events.



Note, again, that this trace shows that the real world is a dirty dirty place. Inside the processor we can be very deterministic, but in the world we have to be aware of what we are dealing with in order to design our systems properly. Here you can see the the bounces are coming microseconds apart; the X-axis time divisions are 10 microseconds. There are some significant pulses that appear less than 10 microseconds apart.

So let's evaluate the actions of the switch and the interrupt library, using that screenshot as our guide...

Pulses A and B

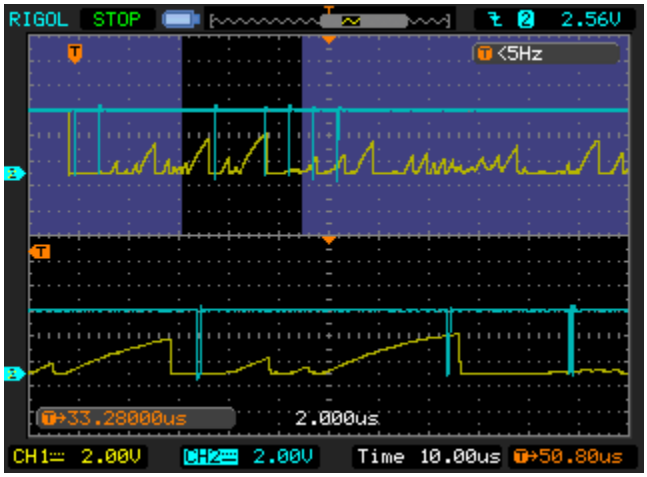
Zooming in on the two pulses, we see the library behave as expected: The switch (yellow trace) triggers the interrupt (blue trace) which reacts about 1us later. Then, about 5us later, the code has determined which pin activated so if you look closely at the bottom trace you can see that it sends a double-pulse (if you can't see it, you'll just have to trust me). Oll Korrect.

At this point, we have saved the current state of the pins into the portSnapshot variable: When the interrupt activates next, we'll need to know which pin changed *from this activation* of the interrupt.

The pin's state is saved at a LOW level.

Pulse C

This is interesting; Pulse C is on the left at the bottom zoomed-in view:



The interrupt is activated by the rising waveform on the left (note that the ATmega328's circuitry actually activated the interrupt somewhere inside this sawtooth). This would be a change in the pin's state, however, by the time the interrupt reacts, the pin is at a low level (refer to the leftmost pulse in the blue trace at the bottom of the above screenshot). In variable “current”, then, the pin's state is LOW. “current” and the portSnapshot variable would thus look the same, the ISR believes that no pin has activated, and there is no activation of the user's function; there is no double pulse.

Then, about 10 microseconds to the right of Pulse C, we find:

Pulse D and E

Pulse D is the leftmost pulse on the bottom trace in the screenshot. Here again we see an activation on a rising pulse, but this rise is appreciably longer than Pulse C. Somewhere during the rise, the ATmega328 recognizes the rising signal, the interrupt triggers, and the library captures the state of the port. This is important: because the ISR has time to recognize the HIGH level (note that the single pulse occurs while the sawtooth wave is still high, so “current” stores this HIGH level), there will be a change in the pin from what was previously saved in the “portSnapshot”. Again if you look closely you can see the double-pulse about 5 microseconds after the initial ISR pulse.

The pin change is recognized and the portSnapshot stores HIGH level.

Pulse F and G

Here we have an interrupt seemingly out of nowhere; about 5 microseconds after Pulse E comes Pulse F, a single pulse signifying entry of the ISR again. But there is no signal (yellow trace) to activate this ISR! ...Or isn't there?

Remember that the ISR at Pulse D acted on a HIGH transition. However, very soon after the ISR's first pulse we saw the signal transition rapidly to low. The ATmega328 recognized this transition, and tripped the PCIFx register (specifically, here, it's PCIF1): it sets a flag to let the system know there is a pending interrupt. We couldn't act on the interrupt until the previous interrupt exited. However, once it did, we enter the ISR again. This was about 5 microseconds in total after executing the user's function in the previous ISR (which happens to be this:

```
volatile uint16_t interruptCount=0;
void interruptFunction() {
  interruptCount++;
}
```

). All the register pop's at the end of the ISR, the return to the main program, executing a single instruction in the main program (this is guaranteed by the system), and the return to the ISR had taken 5 microseconds. Note that because the Pin Change Interrupts are high-level interrupts and there are no other interrupts set in this very simple system, we don't have to be concerned about another interrupt having used any of those 5 microseconds (see page 58 of the ATmega48A/48PA/88A/88PA/168A/168PA/328/328 datasheet).

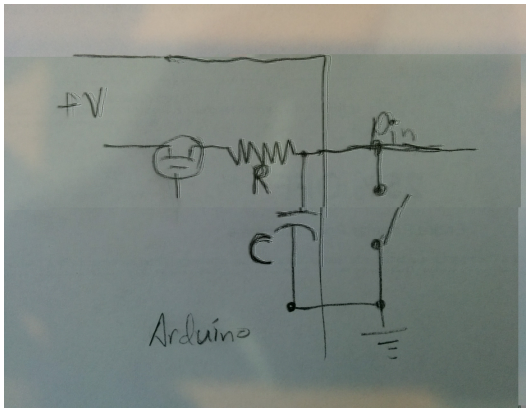
So, to recap: During the ISR that acted on Pulse D, we flagged another interrupt due to the transition from HIGH to LOW. “portSnapshot” would have had registered HIGH for this pin, and now in the ISR for Pulse E “current” registers LOW for this pin. Thus, we find the changed pin and execute the users' interrupt routine at Pulse G.

Implications

At the start of the Arduino sketch, prior to any switch activations, “current” shows a high value for our pin. We have seen at Pulse C that it's possible for us to miss a transition. Is it possible for us to miss a

switch activation?

In the current circuit, we have essentially an RC circuit to V+ but a direct connection to ground. This would explain the sawtooth nature of the waveforms in the switch bounces:



Because of the longer rise time of the signal when the switch is open, when the initial trigger takes place the signal goes low quickly, but it takes some time to go back to high when it bounces. So the "current" variable is likely to report LOW. So we should see the transition, and record it properly.

However there is no guarantee, and we have not even taken into consideration other interrupts that may be set. If a fast-changing signal on a pin registers a transition (on any Pin Change Interrupt pin) while an ISR is being executed, there is a good possibility that that pin's state will be misread. This makes Pin Change Interrupts a poor choice for fast-acting signals *unless fault tolerance is built in to the software*, and possibly even problematic in systems with a number of bouncy switches.

Again, however, multiple cheap bouncy rotary encoders have been used satisfactorily by the author with Pin Change Interrupts on the Arduino with an ATmega328p chip. The situation is not dire by any means, but again as always, Caveat Engineer; you must understand your system that is built on compromise- including, that is, the Pin Change Interrupts- and design accordingly.

EnableInterrupt and PinChangeInt Comparison

This report was written during development of the EnableInterrupt library. The author has maintained another Pin Change Interrupt library, called PinChangeInt https://code.google.com/p/arduino-pinchangeint/. This sort of speed analysis and optimization was not performed for that library. There the compiler and system's code were accepted and measured but not criticized.

So how does this library's speed compare? Is the Enable Interrupt library a significant performance upgrade? Let's compare the two.

Sketch

For this sketch, we're going to use a software interrupt. The idea is this:

1. Use Port C's 0-bit, which would be Pin A0 on the Arduino Uno, for the software interrupt.
2. Set A0 and Pin 13 high.
3. Print something to indicate start.
4. 2 seconds later, bring A0 low. This will interrupt the system and enter the ISR. The ISR is very simple; the function it calls is a simple variable increment.
5. However, it's a little tricky: We discover by trial-and-error that the interrupt code is not entered immediately. In the main loop, we toggle Pin 13 repeatedly. A little experimentation allows us to create exactly the series of instructions needed so that the pin LOW instruction is the last one run prior to entering the ISR. Thus we can see the LOW called jut before entering the ISR, then observe the HIGH transition performed as the first instruction after the ISR.
6. This will tell us within a couple of machine instructions' time (ie, 130 ns), the exact duration of the ISR.

Here's the loop() code:

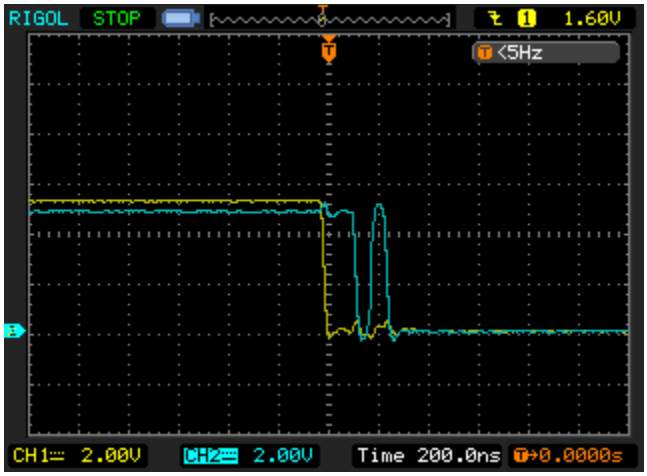
```
void loop() {
  uint8_t led_on, led_off;         // DEBUG
  led_on=0b00100000; led_off=0b0;

  //*led_port|=led_mask; // LED high
  PORTB=led_on;
  PORTC=0x01;
  Serial.println("--------------------------------------");
  delay(2000);   // Every 2 seconds,
  PORTC=0x00;    // software interrupt, port A0, triggers the ISR
  PORTB=led_on;
  PORTB=led_off;
  PORTB=led_on;
  PORTB=led_off; // at this point, we enter the ISR
  PORTB=led_on;  // this is after the ISR.
```

Here are the graphs showing the response of the system.

EnableInterrupt

Here we see the interrupt (yellow line) and the main loop operating, until the ISR takes over and prevents the Pin 13 toggle back to on:

```
  PORTC=0x00;    // software interrupt, port A0, triggers the ISR
  PORTB=led_on;  // We don't see this as the pin is already HIGH
  PORTB=led_off;
  PORTB=led_on;
  PORTB=led_off; // at this point, we enter the ISR
```

So it takes 4 machine instructions to enter the ISR.

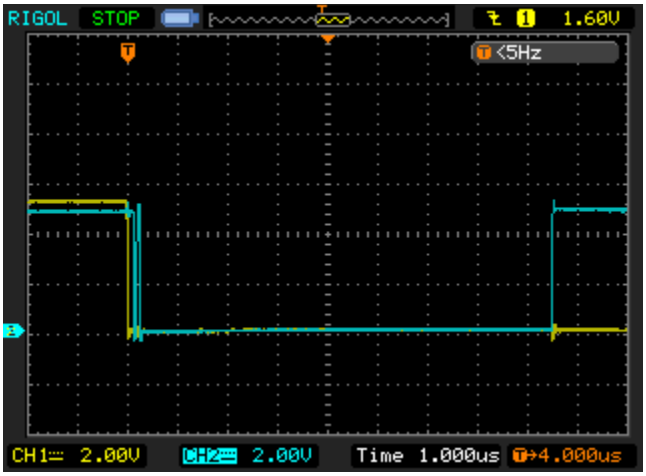
Following

```
PORTB=led_off; // at this point, we enter the ISR
```

in the main loop Pin 13 is brought high again and it stays high until after the next interrupt:

```
PORTB=led_on;  // this is after the ISR.
```

So we should be able to see exactly when the ISR exits and we can measure the duration of the entire ISR. Here is the graph:

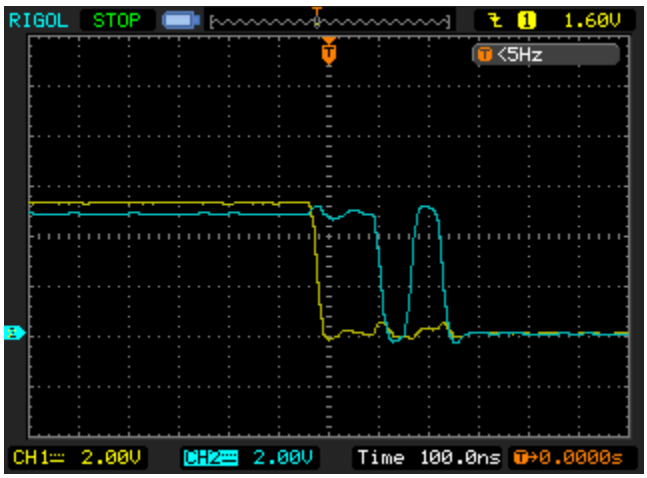


The ISR thus takes about 8 microseconds.

Compare the PinChangeInt library:

PinChangeInt

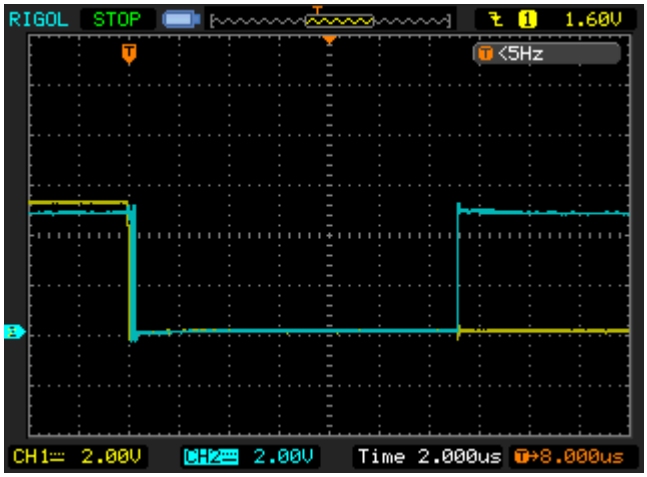
It takes about the same time to enter the ISR, as expected. There is no difference between entering any ISR, that is up to the system (note that the time/div is twice as long here as the previous graph):



How long does this ISR take? The graph tells the tale:

Here we can see that, between the execution of these two commands:

```
PORTB=led_off; // at this point, we enter the ISR
PORTB=led_on;  // this is after the ISR.
```



is about 13 microseconds elapsed time. All else being equal, the Enable Interrupt code is 5 microseconds faster.

License