February 2, 2018

v1.0

Polling and Pin-Triggered Interrupt Service Routines for Quadrature Rotary Encoders

Routines for reading low-cost mechanical quadrature rotary encoders typically use the "lookup table" method for evaluating successive encoder switch states. That method works fairly well, but is not ideal. It is not always able to distinguish a valid change from bounce noise which produces the same pattern. As a result, it may report a change in direction when none is intended. It also produces a "tick" at every valid change of state, which produces multiple ticks per detent that must be divided down.

The routines presented here include both polling routines in which a timer is set to generate periodic interrupts to read the switch pins, as well as pin-triggered interrupt routines. Both are based on the assumption that the two switches will never be bouncing at the same time. So while one switch is changing state and bouncing, the other switch is always stable. That assumption may not be true for very low-quality encoders, or for any encoder if it is turned too rapidly. But routines based on the assumption work remarkably well in practice. No hardware switch debouncing is required, and the logic of the routines achieves debouncing without waiting for a switch to settle. The routines can be set to work with encoders having the same number of pulses as detents per revolution (referred to here as "Type-0"), as well as those having half as many pulses as detents ("Type-1"). In both cases, the routines produce one tick per detent.

This document presents the details of the routines - not formal code, but the logic steps needed to understand and implement them. Separately, source code in assembler and the resulting .HEX files are included for the Texas Instruments MSP430G2231 microprocessor.

Polling

At each detent, both switches are always at the same state - both always open (high) for Type-0 encoders, and either both open or both closed (high or low) for Type-1. The polling routine begins by defining a Target switch pattern equal to the opposite of the current detent state. So if the state at the current detent is now 11b, the Target is defined as 00b. At each subsequent poll, the previously-saved Current state is copied to Previous, and then a new Current state is read from the port. Then the routine tests to see if the new Current state is the same as the Target. If not, it exits the servicing routine, doing nothing.

On the first poll when the new Current matches the Target, a tick is recognized, and the direction is determined by (Current XOR Previous). But before exiting the routine, the old Target is inverted to its opposite state for subsequent polls. If bouncing occurs on the latest pin to reach the old Target, it will be ignored because the other pin will still be stable at the old Target state, and will therefore never match the new Target until that pin changes state sometime later.

The exception to this process is that for Type-0 encoders a tick is recognized only if the Current/Target matching occurs when the Target is 11b. That produces one tick per detent. But the Target inversion is still performed at both 00b and 11b for such encoders.

The essence of this algorithm is that a tick is recognized the first time the switches match the Target, the direction being determined by which pin was the last to reach that state. The intermediate state change by the leading pin is just another non-matching poll, and when the match finally occurs, all bouncing on the trailing pin is mitigated by immediately inverting the Target. The leading pin's stable state at that point, at the old Target state, makes it impossible for bouncing on the trailing pin to result in a match at the new Target state.

Since the encoder pins do not generate interrupts, their pullup resistors need be enabled only when reading the port. This saves power, particularly for Type-1 encoders when half the time, on average, they come to rest at a detent where both switches are closed.

Assume:

The encoder pins are connected as inputs to any two pins of a digital I/O port.
These port bits need not be adjacent, but both must be on the same port.
Pull-UP resistors can be enabled on those pins internally.

Setup in Main Body:

```
Enable internal pull-UP resistors on the two pins
Read and mask switch state bits, save as Current
Set the two pin resistors to pull-DOWN (no floating inputs)
Target = 00b
If Current = 00b then Target = 11b
Set up timer-generated interrupt at about 500 per second (2 mSec)
```

Interrupt Service Routine:

```
Switch the two pin resistors to pull-UP
Copy Current to Previous
Read and mask switch state bits, save to Current
Set resistors back to pull-DOWN

If Current != Target then Return

If (Target = 00b) AND (encoder = Type-0)  then goto Exit1  // tick only at 11b
Direction = (Current XOR Previous)  // 10b = CW, 01b = CCW, others invalid
If Direction value is valid, process one tick for that Direction - blink LED, etc.

Exit1:
  Invert Target
  Return
```

A Type-1 encoder has half as many transitions between detents as a Type-0. Therefore it should be posssible to poll at half the frequency and still get the same performance. And since polling permits the pullup resistors to be enabled only during the few microseconds when the port is read, Type-1 encoders may be the preferred choice for projects where polling will be used.

Pin-Triggered Interrupts

Instead of periodic polling using timer-generated interrupts, changes of state on the encoder pins themselves can be used to trigger the processing interrupts. The method described here uses the same Current/Previous/Target/Direction logic used in polling. Interrupts occur only when a switch changes state, but not on any bouncing that takes place. This means that far fewer interrupts are generated than in polling - a maximum of four per detent versus 500 per second for polling - and there are no interrupts at all when the encoder is idle. But the servicing routine is somewhat more complicated.

An interrupt is enabled on only one pin at a time, on a specific edge transition. When that interrupt occurs, the interrupt enable is switched to the opposite pin during the servicing routine. Based on the original assumption that the pins will not be bouncing at the same time, we are switching the interrupt to the switch that is already stable. The bouncing that continues to occur on the pin which generated the interrupt will cause no further interrupts because interrupts on that pin are no longer enabled.

There are two problems with this type of servicing that complicate things. The first is that a small but significant delay occurs between the triggering of the interrupt and the reading of the port. Time is required to push things to the stack and begin executing the servicing routine. During that time, the value on the interrupting pin may change from its triggering value because of bouncing. To mitigate that possibility, after reading the port we replace the read value of the interrupting pin with the value we know it must have had - based on the edge setting. (If the processor can be set to automatically latch the current port value when the interrupt is triggered, this step is not necessary.)

The other problem is that with interrupts enabled on only one pin at a time, with any reversal of direction the pin we expected to be the leading pin, and therefore provide the correct Previous value when the Target state is reached, will wind up being the trailing pin. The previous transition on the actual leading pin will have been lost because no interrupt was enabled on that pin when it changed state. So if we ever arrive at a Target state, and the Previous value at that time is also a Target state, we know we have missed an interrupt, and we must reconstruct the pin values at that missing interrupt to get the proper Previous value. That is accomplished by inverting the value of the non-triggering pin in Previous.

Since transitions on the pins trigger the interrupts, at least one pullup resistor will have to be enabled at all times, and the second enabled when reading the port. But the logic presented here just leaves both resistors enabled all the time.

Assume:

The encoder pins are connected as inputs to any two pins of a digital I/O port.
These port bits need not be adjacent, but both must be on the same port.
Pull-UP resistors can be enabled on those pins.
Each chosen pin can be set to trigger an interrupt on a designated rising or falling edge.

Setup in Main Body:

Enable pullUP resistors on the two encoder pins
Read and mask switch state bits, save as Current
Target = 00b, Edges on both pins = falling
If Current = 00b then Target = 11b, Edges on both pins = Rising
Enable edge-triggered interrupt on ONE pin (either one) per Edges setting

Interrupt Service Routine:

Copy Current to Previous
Read and mask switch state bits, save to Current

Clear the bit in Current that caused the interrupt
Replace with the Edges value for that bit: falling = 0, rising = 1

If Current != Target then goto Exit2 //not at Target yet

If (Target = 00b) AND (encoder = Type-0) then goto Exit1 // tick only at 11b
If Previous equals 00b or 11b, invert the non-triggering pin value in Previous
Direction = (Current XOR Previous) // 10b = CW, 01b = CCW, others invalid
If Direction value is valid, process one tick for that Direction - blink LED, etc.

Exit1:
Invert Target
Invert Edges

Exit2:
Switch interrupt to other pin
Clear flags
Return

A Type-0 encoder may be the best choice for this type of servicing because both of its switches are open at every detent. While it requires twice as many interrupts per detent as a Type-1 encoder, the absolute number of interrupts is very small: 4 versus 2. So the processor time devoted to encoder support is minimized.

Some processors have a limited number of single-edge-triggered I/O port interrupts. They may instead have interrupts of the "any change" type where any transition on a pin triggers an interrupt. It should still be possible to use this routine with such interrupts, but the coder would have to maintain a variable which imitates the Edges register assumed here.

Texas Instruments MSP430 Code

Included in this repository are examples demonstrating both methods as implemented for the Texas Instruments MSP430G2231 processor. The source code is written in assembler, and for each method .HEX files are provided for use with Type-0 and Type-1 encoders. In addition, for the polling method, .HEX files for both 488 and 244 interrupts per second are provided. The code assumes the processor is installed on the Launchpad, with the encoder pins connected to p1.4 and p1.5. The Launchpad green LED will flash (dimly) on a CW rotation, and the red LED on a CCW rotation. If everything is working, there should be no false direction flashes.

While the MSP430 code is not directly applicable to other processors. Examination of the well-documented cource code will provide an understanding of exactly how the routines can be implemented in logical terms. So far, the routines have not been tested in C, or in C-like Arduino code, but the servicing routines are short enough that any higher-level inefficiencies should pose no problem. The main challenge is that since both the encoder pins and the indicator LEDs share their port pins with other functions, almost all of the servicing code is bit-masked, so that only the relevant bits are read or changed in the port registers after masking. That's easy in assembler, but it may be more difficult in higher-level languages.

https://github.com/gbhug5a