Decode a quadrature encoder in software

Sid Levingston, Gentec-EO, Lake Oswego, OR; Edited by Martin Rowe and Fran Granville - January 20, 2011

Quadrature encoders work in many applications to determine displacement and direction of mechanical travel. They vary in design, but they all do the same thing: supply a set of square waves 90° out of phase. Figure 1 shows the typical output signals.

The encoder rotates clockwise when Channel A leads Channel B. If Channel B leads Channel A, the encoder is rotating counter clockwise. By counting the pulses and the direction of rotation, you can find the position of the encoder. Although ICs can decode quadrature encoders, you can easily and less expensively have the processor decode the signal. The signals from Channel A and Channel B go through a Schmitt trigger if necessary, but many encoders and processors include this trigger internally. The signals are then applied to two I/O pins on the processor that support edge-triggered interrupts. The code in the interrupt handler implements a standard decoder algorithm, but all algorithms typically follow these steps:

1. Set up a state table like the one in Table 1. The table wraps around from State 3 to State 0.
2. Initialize a counter.
3. Measure the current state of Channel A and Channel B. Find that state in the table and set a pointer to it.
4. Enable the interrupts.

In the interrupt handler, use the following steps:

1. Read the state of Channel A and Channel B.
2. If the state is the one preceding the pointer, decrement the counter.
3. If the state is the one following the pointer, increment the counter.
4. Set the pointer to the new state. 5. Clear the interrupt.

<table>
<thead>
<tr>
<th>Table 1: Logic States of Quadrature Encoder</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>State 0</td>
</tr>
<tr>
<td>State 1</td>
</tr>
<tr>
<td>State 2</td>
</tr>
<tr>
<td>State 3</td>
</tr>
</tbody>
</table>

Figure 1 A quadrature encoder produces pulses that are 90° out of phase.
This method requires that a state table exists, that the previous state remain, and that the handler determine on each interrupt which of four states exists and then make a decision based on two possible conditions. The handler accomplishes this task with a four-case switch, in which each case has two if conditions.

```
LISTING 1 QUADRATURE-ENCODER CODE
#define CHA BIT0
#define CHB BIT1

initPort()
{
   P1SEL = 0;
   P1DIR = 0;
   P1OUT = 0;
   P1IES = 0;  // rising edge.
   P1IE = CHA + CHB; // interrupt on CH A or CH B rising edge
}

#define vector = PORT1_VECTOR
__interrupt void port1_ISR (void)
{

   if(P1IFG & CHA) // who interrupted? If A then A is high
   {
      (P1IN & CHB) ? gStepCount++ : gStepCount++; // test B
      P1IFG &= ~CHA; // clear interrupt
   }

   if(P1IFG & CHB) // who interrupted? If B then B is high
   {
      (P1IN & CHA) ? gStepCount++ : gStepCount--; // test A
      P1IFG &= ~CHB; // clear interrupt
   }
}
```

Now, consider what happens in the real world. If the I/O pin generates an interrupt on a rising edge, then when the interrupt happens, that channel goes from low to high. Therefore, there’s no reason to read the state of the pin that interrupted. The other channel did not interrupt because the signals are 90° out of phase. So, to determine the current state, you need only to read the state of the pin that didn’t initiate the interrupt. The state of the unchanged low or high signal tells which way the encoder rotated. If it is low, then the interrupting pin is leading. If it is high, then the interrupting pin is trailing. You can use these facts to implement an efficient interrupt handler with no state table and no memory of a previous state.

The code in Listing 1 was tested on an MSP430F processor connected to an Encoder Products model 15T. The encoder monitored the position of a linear stage. The stage traveled 85 mm and could be tracked with resolution of 5 microns.

The define statements (highlighted in red) make the code more readable. The initPort() function (highlighted in blue) sets up the rising-edge interrupt on channels A and B. The final piece is the interrupt handler (highlighted in green). Note that it contains only six lines of code compared with the 20 or 30 lines it would take to implement the traditional method of decoding the channels.