EECS 461, Winter 2023, Problem Set 2: SOLUTIONS[1]

1. In all sensor interfacing, it is necessary to minimize the response of the system to *noise* in the measurements. For example, in quadrature decoding noise can cause spurious pulses in the quadrature signals. These pulses can appear to be valid rising or falling edges of the signals, and can result in erroneous updates of the counter and incorrect position measurements.

   Noise immunity for the S32K144 FlexTimer module is provided by a noise filter that is essentially a logic block that monitors changes in an input pulse train and rejects those whose duration is shorter than a predefined value. The Quadrature Decoder Mode of the FlexTimer accepts two input pulse trains, and can implement a filter on each one (although we do not do so in Lab 2). In this way spurious pulses are prevented from causing erroneous updates of the encoder counter. The use of the filter in Quadrature Decoder Mode is illustrated in Figure 1.

   When the filter is being used there will be latency, or delay, the between the time that a valid pulse occurs and when it is processed. For Quadrature Decoder Mode the output of the filter is fed into the logic that increments or decrements the counter based on the relative changes in the two quadrature signals, phase A and phase B. Any delay caused by the filter will therefore cause a delay in the time that the CNT register, and thus the encoder counter, is updated.

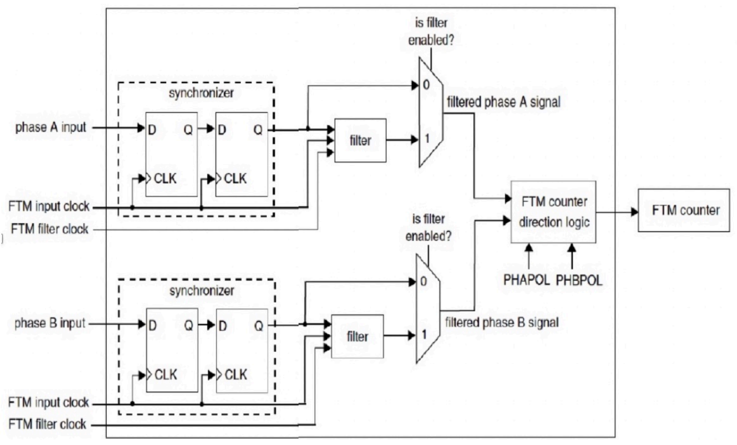

Figure 1: Filtering in the Quadrature Decoder Mode.

   The operation of the filter is described in Chapter 45 of the NXP S32K Reference Manual; however, *all the relevant information is summarized in the section on the FTM filter in the Powerpoint slides for Lab 2.*

   When a transition occurs on one of the quadrature signals (a rising or falling edge on Phase A or Phase B) a counter internal to the filter begins to increment until it reaches a specified value. If the signal remains unchanged at this time, the transition is considered and it is passed to the FTM counter direction logic and the16 bit counter register is updated. If the opposite transition is detected before the counter reaches the specified value, then the original transition is treated as noise and ignored, and the counter resets until another transition is detected.

   The parameters of the FlexTimer filter are controlled by bitfields in three registers:

   - Quadrature Decoder Control And Status (QDCTRL): PHAFLTREN and PHBFLTREN
   - Status And Control (SC): FLTPS[3:0]
   - Input Capture Filter Control (FILTER): CH0FVAL[3:0] (PhaseA) and CH1FVAL[3:0] (PhaseB)

[1]Revised November 5, 2022.

These parameters determine the duration of the noise pulses that will be rejected by the filter, and thus also the amount of delay, or *latency* that the filter introduces. The amount of delay depends on whether or not the filter is disabled (by setting CHnFVAL[3:0]=0) and also on the value of the FTM filter clock that is related to the input clock by a prescaler. For details see the slides for Lab 2.

(a). Suppose PHAFLTREN, PHBFLTREN, FLTPS[3:0], CH0FVAL[3:0] and CH1FVAL[3:0] are all 0 (this should be the way quadrature decode is set up for Lab 2). How much delay is introduced between the time an edge appears on the input and the CNT register is updated? Recall that the FlexTimer clock is 10MHz.

SOLUTION:
If the channel input filter is disabled (CHnFVAL[3:0] = 0) and and the filter clock is equal to the FTM clock (FLTPS[3:0] = 0), then the number of rising edges of the FTM input clock between an edge transition on the input channel and updating the CNT register is 3. This corresponds to a delay between 0.2μsec and 0.3μsec.

(b). Now suppose Phase A and Phase B filters are enabled, and FLTPS[3:0], CH0FVAL[3:0] and CH1FVAL[3:0] all equal 0b0001. Now how much latency is introduced?

SOLUTION:
FLTPS[3:0] = 0b0001 selects the clock prescaler used in the FTM filters to be divide by 2. Thus, the filter clock is 5MHz. Then the number of rising edges of the FTM input clock between an edge transition on the input channel and the update to the CNT register is 3 rising edges of FTM input clock + ($1 + 4\times$ CHnFVAL) rising edges of FTM filter clock, corresponding to a maximum delay of 1.3μsec.

(c). For the 1000 CPR encoder in the EECS 461 lab, each of the two quadrature channels will produce 2000 transitions (rising and falling edges) per wheel revolution. Suppose that the wheel is turning at a constant rate R revolutions/second. What is the amount of time Δt between successive edges (i.e., the length of a valid pulse)?

SOLUTION:
Denote the frequency of pulses by $f = 1/\Delta t$. Then $f = 2000$ edges/rev $\times R$ rev/second $= 2000R$ edges/second, and thus $\Delta t = 1/2000R$ second/edge.

(d). Using the values of FLTPS[3:0], CH0FVAL[3:0] and CH1FVAL[3:0] from part (1b), what is the largest integer value of R such that the filter does not cause a valid pulse to be ignored?

SOLUTION:
We need that the length of the pulse, Δt is greater than the maximum latency, which we computed to be 1.3μsec. Hence $1/2000R > 1.3\mu$sec or $R < 385$ rev/sec.

2. Consider the `C` code listed below. The variables `position1`, `position2`, `position3`, and `position4` are 32-bit signed integers that, if updated appropriately, will provide a running position count. The variables `CurrCount` and `PrevCount` are 16 bit unsigned integers that represent two successive reading of the counter used for quadrature decoding. Suppose the value of `CurrCount=0x0000`, and `PrevCount=0xFFFF`. This indicates that the position has increased by one count.

```
main()
{
int position1=0, position2=0, position3=0, position4=0;
unsigned short CurrCount, PrevCount;

CurrCount=0x0000;
PrevCount=0xFFFF;

position1=(CurrCount-PrevCount)+position1;

position2=((int)CurrCount-(int)PrevCount)+position2;

position3=(int)(CurrCount-PrevCount)+position3;

position4=(short)(CurrCount-PrevCount)+position4;
}
```

(a) What value will the first position variable take after it is updated? Will it have the correct value of +1? Explain your answer.

(b) Repeat part (2a) for the second position variable.

(c) Repeat part (2a) for the third position variable.

(d) Repeat part (2a) for the fourth position variable.

(You may wish to read the discussion of data type conversions on pp. 197-198 of [1].)

SOLUTION:

The `C` program listed below

```
void main()
{
  int position1=0, position2=0, position3=0, position4=0;
  unsigned short CurrCount, PrevCount;

  CurrCount=0x0000;
  PrevCount=0xFFFF;

  position1=(CurrCount-PrevCount)+position1;

  position2=((int)CurrCount-(int)PrevCount)+position2;

  position3=(int)(CurrCount-PrevCount)+position3;

  position4=(short)(CurrCount-PrevCount)+position4;

  printf("position1 = %i\n", position1);
  printf("position1 = %x\n\n", position1);

  printf("position2 = %i\n", position2);
  printf("position2 = %x\n\n", position2);

  printf("position3 = %i\n", position3);
  printf("position3 = %x\n\n", position3);

  printf("position4 = %i\n", position4);
  printf("position4 = %x\n\n", position4);
}
```

produced the following output:

```
position1 = -65535
position1 = ffff0001

position2 = -65535
position2 = ffff0001

position3 = -65535
position3 = ffff0001

position4 = 1
position4 = 1
```

and thus we see that only position4 updates correctly. We now see why only the fourth position is correct: To perform an arithmetic operation such as "-", each `unsigned short` is converted by *integral promotion* to an `int` [1]. Doing so yields

```
   CurrCount - PrevCount = 0x00000000 - 0x0000FFFF
                         = 0x00000000 + 0xFFFF0001
```

and thus

```
   position1 = 0xFFFF0001 + position1
             = 0xFFFF0001,
```

or `position1 = -65535` in decimal notation.

Similarly,

```
   (int)CurrCount - (int)PrevCount = 0xFFFF0001,
```

and

```
   (int)(CurrCount - PrevCount) = 0xFFFF0001,
```

and thus `position2` and `position3` will yield the same incorrect value as `position1`.

To evaluate `position4`, the `int CurrCount - PrevCount = 0xFFFF0001` must be converted to a (signed) `short`. Not all integer values can be represented in this way, and the C standard allows the result to be defined by the implementation. For our compiler, the leading bits are simply truncated, and thus

```
   short(CurrCount - PrevCount) = short(0xFFFF0001)
                                = 0x0001.
```

When a (signed) `short` is added to an `int`, it must be sign extended, and thus

```
   position4 = short(CurrCount - PrevCount) + position4
             = short(0xFFFF0001) + position4
             = 0x00000001,
```

which is the correct value.

3. In this problem, we will use Matlab, Simulink, and Stateflow to construct a simplified model of a quadrature decoding algorithm. The purpose of the problem is both to learn more about quadrature decoding as well as to learn how to set up and run a Stateflow model. Much embedded control software consists of event driven state transitions, and is best modelled using finite state machines. In fact, Stateflow was developed so that engineers could model the interaction between such software and a model of a dynamical system.

   The Matlab file "run_quad_decode.m", available on the Canvas website, is used to generate the two quadrature signals shown in Figure 2. These signals are then used as inputs to the Simulink model shown in Figure 3, written to model the "4X" mode of quadrature decoding as discussed in Lecture 3, and as implemented by the FlexTimer quadrature decode function on the NXP S32K144. Executing the file "run_quad_decode.m" will also perform quadrature decoding on the two signals in Figure 2, with the result shown in Figure 4.

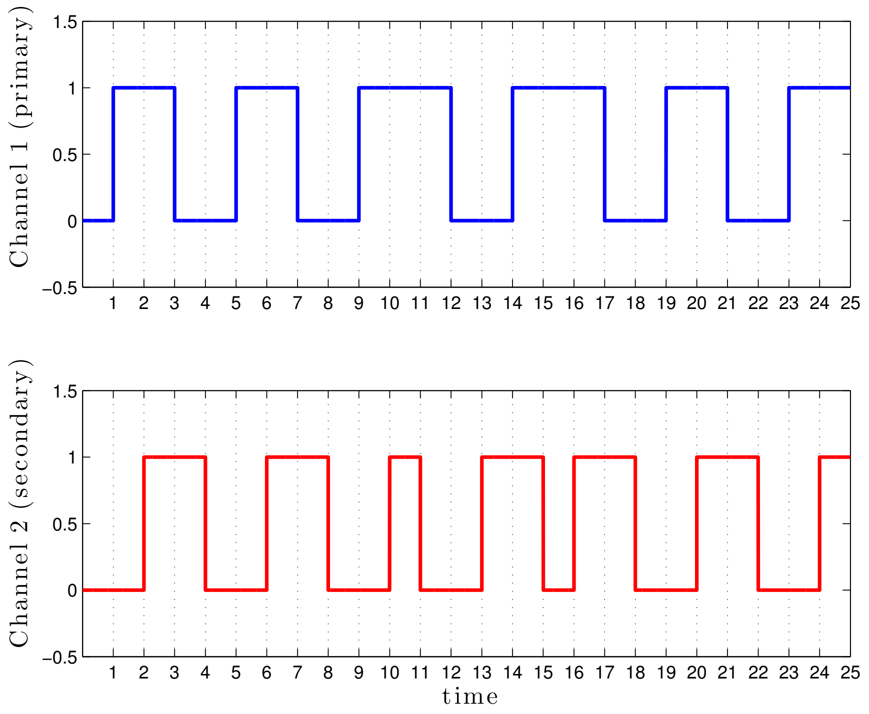

Figure 2: Quadrature Signals from Encoder

   Now inspect the incomplete Stateflow model of quadrature decoding shown in Figure 5. Your assignment is to complete this Stateflow model, so that when you execute "run_quad_decode.m" you will obtain the same plot as that shown in Figure 4. To do so, you must

   (i) using the Modeling/Event Input menu, add events to cause transitions between the states; call these events "primary" and "secondary", where "primary" and "secondary" refer to Inputs one and two from the workspace, respectively.
   (ii) at each transition, increment or decrement an output called "counter" appropriately
   (iii) configure the chart so that "primary" and "secondary" are events input from Simulink, triggered on both rising and falling edges.
   (iv) the output "counter" is an 8 bit unsigned integer; its initial value is `uint8(0)`.
   (v) you can verify that the chart is set correctly by using the Modeling/Model Explorer menu.

   Your completed model of quadrature decoding should yield the plot of counter value shown in Figure 4. Hand in the corresponding plot from your program to show that it achieves the same result, as well as your Stateflow chart.

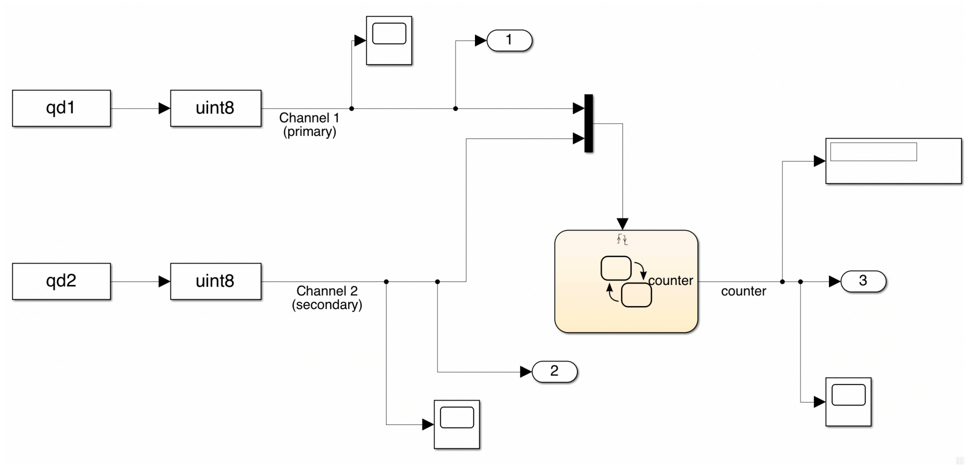

Figure 3: Simulink Model for 4X Mode Quadrature Decoding

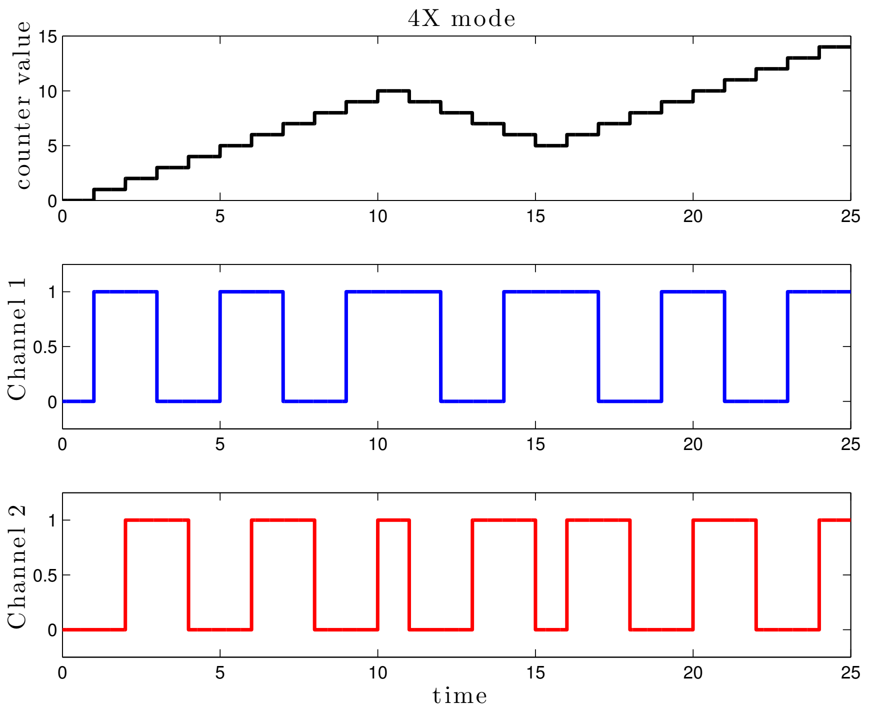

Figure 4: Counter Value vs Time for 4X Mode Quadrature Decoding

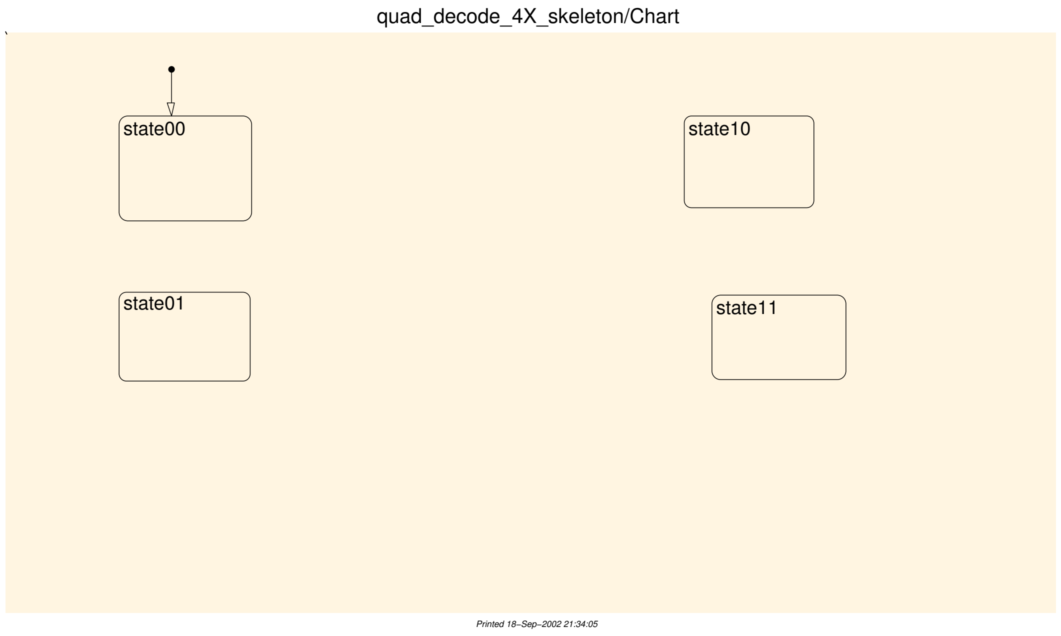

Figure 5: Incomplete Stateflow Chart for 4X Mode Quadrature Decoding

SOLUTION:

Your plot of counter value vs time should be identical to that in Figure 4. Your Stateflow chart should resemble that depicted in Figure 6.

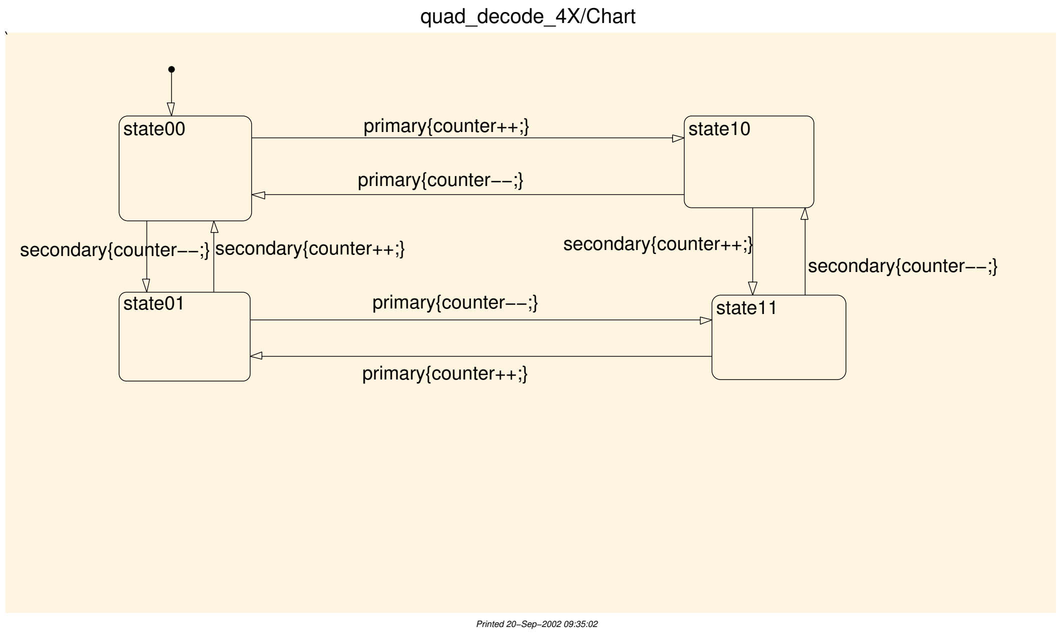

Figure 6: Stateflow Chart for 4X Mode Quadrature Decoding

References

[1] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, 2nd ed.* Prentice Hall, 1988.