ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institute for Dynamic Systems and Control
IDSC
Institut für Dynamische Systeme und Regelungstechnik

151-0593-00 **Embedded Control Systems** (Fall 2017) **Lab 2**

Topic: **Quadrature Decoding using the enhanced Time Processing Unit (eTPU)**

Pre-Lab due: 13. 09. 17 at 1 p.m. Post-Lab due: 13. 09. 17 at 5 p.m.

Name: .... Forename: .... Initials: ....

marianne.schmid@idsc.mavt.ethz.ch, 12. September 2017

1 Overview

The purpose of this lab is to learn about quadrature decoding and apply knowledge of sampling. To do so, we will use the Quadrature Decoding function of the MPC5553 enhanced Time Processor Unit (eTPU) as an angular position sensor.

1.1 Encoder

When developing a controller for a mechanical device with moving parts, it is often necessary to gather information about the motion of the device by using an encoder. The haptic device we will be using in the lab has an optical encoder for the purpose of obtaining position and velocity information. Optical encoders typically use optical gratings to produce two square-wave signals that are 90 degrees out of phase. This is an example of a quadrature-encoded signal. Figure 1 shows how the two signals vary as the shaft rotates in one direction.

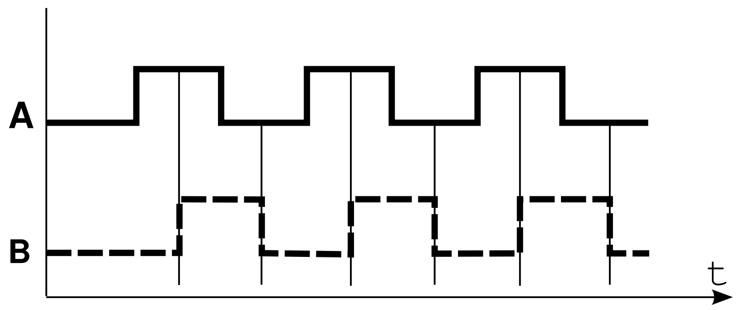

Figure 1: Output for a 2 channel optical encoder rotating in one direction

1.2 Time Processor Unit (eTPU)

The enhanced time processor unit is described in Chapter 18 of the MPC5553 Reference Manual and the Application Note AN2842 (available on the course website). The eTPU is a timing unit featured on the MPC5553 microcontroller that operates in parallel with the core (CPU). The eTPU does the following:

- Executes programs independently from the host core
- Detects and precisely records timing of input events
- Generates complex output waveforms
- Is controlled by the core without a requirement for real-time host processing

Software for specific eTPU functions may be written by the user, or, for common functions, obtained from Freescale. Freescale Application Note AN2842 describes the quadrature decode function. Read the application note, and look over Chapter 18 of the Reference Manual. You can find the necessary information concerning eTPU configuration registers in Section 18.4.

1.3 Quadrature Decoding eTPU Function

The quadrature decode (QD) function of the MPC5553 eTPU operates in one of three modes: slow, normal or fast. We will operate the eTPU in the slow quadrature decode mode. The quadrature decode function uses two eTPU channels to decode quadrature signals into a 24-bit counter. **Note:** In this lab we will use only 16 bits of the count register.

When in slow mode, the quadrature decode function is able to determine direction by looking at both the rising and falling edges of each signal. If the primary channel is leading the secondary channel, the 24-bit counter is incremented. If the primary channel is trailing the secondary channel, the counter is decremented. The function does this by comparing the current state of the two channels with the state before the most recent rising/falling edge. For example, when channel B goes from low to high and channel A remains high, the counter is incremented since B is trailing A. This creates 16 possible transitions: 4 increment the counter, 4 decrement the counter, and 8 are errors (see the pre-lab, below, for more information). When the function receives a sequence that is an error, the counter is unchanged.

You can also program the function to switch to normal and fast modes, based on the speed of the encoder wheel. Normal mode applies when the speed is greater than a defined slow-to-normal threshold, but less than a specified normal-to-fast threshold. In normal mode, the counter is updated by one for each valid transition on either channel as in slow mode, but the direction is not calculated (the last direction calculated in the slow mode is kept). When in fast mode, the function only monitors the signal on the primary channel, and updates the counter by 4 on each rising edge transition. **Note:** In this lab we will only work in slow mode, but it is important to understand the uses of the other two modes.

2 Design Specifications

2.1 Hardware

On the interface board, you will see a header bank labeled eTPUA [3..7] below the DIP switches and two Encoder sockets (Enc_1 and Enc_2) at the right hand side at the bottom. Encoder 1 is connected to eTPUA channel 0 and 1, Encoder 2 to eTPUA channel 3 and 4. Eight channels can interface with up to four quadrature encoders (the chip itself supports up to 32 channels per eTPU; 8 channels are made available on the interface board, and two channels are needed for each quadrature encoder). The pins for eTPUA channels 3 and 4 are connected to the encoder over a parallel ribbon cable. You may use the header bank to measure the voltage signals coming from the quadrature encoder channels.

The digital I/O interface from Lab 1 will be used again.

2.2 Software

fqd.h and fqd.c

You need to make a C file called *fqd.c* from the *fqd_template.c* file that contains four functions named `init_FQD`, `ReadFQD_pc`, `updateCounter` and `updateAngle`. The prototypes and details of these functions can be found in the *fqd.h* file. It is very important that you follow these prototypes and the comments in the *fqd_template.c* file closely. Regions in *fqd_template.c* which may need to be treated specially are marked with a `/* Fill in */` tag. Place *fqd.c* file in your `lib/` directory and the *fqd.h* file in your `include/` directory.

Makefile

To simplify the compilation process, a makefile has been provided to you. Note that for the makefile to work, you need to make sure that all of your files are in their proper directories, as specified in the lab handouts. The `gmake` command must be run from your `lab2` directory at the command shell.

Freescale header files used in this lab: *etpu_set.h, etpu_util.h, etpu_qd.h* and *etpu_ppa.h.*

2.3 Notes on Casting

In previous years students have sometimes observed a peculiar problem in which their `updateCounter` function failed to update the position correctly. The specific behavior observed was that the top 8 bits of the bar LEDs would alternate between being all on (0xFFXX) or all off (0x00XX). In the following exercise, we will see that this problem is due to the subtleties involved in typecasting correctly.

The following code does not work correctly. NEW_TOTAL and LAST_TOTAL are `int32_t`s and CURR_FQDPC and PREV_FQDPC are `uint16_t`s:

```
int32_t updateCounter()
{  /* setup */
   return (LAST_TOTAL + (CURR_FQDPC - PREV_FQDPC));
}
```

This code will work:

```
int32_t updateCounter()
{  /* setup */
   return (LAST_TOTAL + (int16_t)(CURR_FQDPC - PREV_FQDPC));
}
```

Note the difference between these two examples: they are identical except that the quantity (CURR_FQDPC - PREV_FQDPC), of type `uint16_t`, is cast to an `int16_t` before being added to LAST_TOTAL. The reason for doing so is that arithmetic on the MPC5553 is performed in 32 bit registers. When an unsigned 16 bit number is placed into a 32 bit register, the leading 16 bits of the 32 bit register are filled with zeros, a procedure termed integral promotion. After subtraction, these leading 16 bits may no longer be all equal to zero, with the effect that adding (CURR_FQDPC - PREV_FQDPC) to LAST TOTAL may yield an erroneous result. Casting (CURR_FQDPC - PREV_FQDPC) to a signed 16 bit number, on the other hand, truncates the leading 16 bits. When the resulting signed 16 bit integer is promoted to a 32 bit integer so that it may be added to LAST_TOTAL, the rules for sign extension in two's complement arithmetic guarantee that it will have the proper sign, and that the counter will be updated correctly. For more information about integral promotion and casting, see Section 2.7 of The C Programming Language, by Brian W. Kernighan and Dennis M. Ritchie. For information about sign extension, see the notes available on the course website.

To make this clearer, here is an example:

LAST_TOTAL = 0x00007FFF
CURR_FQDPC = 0xFFFF
PREV_FQDPC = 0x0000

Taking the clockwise direction to be positive, we see that the encoder has incremented one count in the negative (CCW) direction.

Subtracting CURR_FQDPC - PREV_FQDPC yields the 32 bit value 0x0000FFFF, which represents a large positive number, instead of -1. Casting (CURR_FQDPC - PREV_FQDPC) as an int16_t will truncate the leading 16 bits, yielding the signed 16 bit value 0xFFFF. When the latter value is converted to int32_t before addition to LAST_TOTAL, it will be sign extended to 0xFFFFFFFF, or -1, which is the correct answer.

Note that LAST_TOTAL and PREV_FQDPC will have to be `static` variables in order to maintain their values between calls to `updateCounter`. **Note also that PREV_FQDPC must have the same initial value as the position count register in *fqd.c* or there will be an initial offset in the count**.

3 Pre-Lab Assignment

Pre-lab questions must be done ***individually*** *and handed in at the start of the in-lab section. You must also,* ***with your partners****, design an initial version of your software before the in-lab section.*

1. Fill in the truth table in Table 1 for quadrature decoding. X_{i-1} and Y_{i-1} refer to the previous state while X_i and Y_i refer to the current state. The valid actions are increment, decrement, and error. A few have already been done for you.

2. The haptic wheel in the lab uses the encoder HEDL-5560 B13 from Avago. This encoder uses two channels with 1000 CPR (cycles per revolution) for each channel. What is the resolution, or smallest measurable angular motion, in degrees/encoder count, of the encoder when the QD function is operating in slow mode?

3. a. On Figure 2, draw lines where the quadrature decode function would update the counter and label each line with the value the counter would hold. Assume the function is operating in slow mode.

   b. Repeat Part (a) for Figure 3, but assume that the function is running in fast mode now.

4. What are the names of the threshold parameters that must be set to enable transitions among slow, normal, and fast modes? Suppose all these parameters are set to zero. In which mode will the QD function operate?

5. Read the discussion of noise immunity of Section 3.2 of the Freescale Application Note AN2842. What is the purpose of an acceptance window, and how does it help provide noise immunity? In which QD modes is an acceptance window used?

6. Some encoders have a third track, or channel, consisting of a single window. This may be used to align the wheel in a reference position, or to update a counter that keeps track of the integer number of wheel revolutions. These options are referred to as 'home' and 'index' channels in the Freescale Application Note 2842. Since we do not use these options with our encoder, it will be necessary to specify that they are not used when we initialize the eTPU. By consulting Section 4.1.1 of AN2842, find the variable used to determine whether an index or home channel is being used, and the value it must take when only quadrature decoding channels are to be used.

7. We shall use eTPU channels 3 and 4 to input the two quadrature signals produced by the encoder. The functionality of the pins to which these eTPU channels are connected is controlled by Pad Configuration Registers SIU PCR 117-118. By consulting Table 6.16 in Section 6.3.1.12 of the MPC5553 Manual, determine how these registers should be set.

8. Suppose that the int32 t variable TOTAL = 0x00000000, and that the uint16 t variables CURR_FQDPC = 0xFFFF and PREV_FQDPC = 0x0000, representing a turn of one encoder count in the negative (CCW) direction.

   a. With faulty casting TOTAL = TOTAL + (CURR_FQDPC - PREV_FQDPC) what will be the (incorrect) updated value of TOTAL? Will the value of TOTAL continue to be incorrect as the wheel is turned further in the CCW direction?

   b. Suppose that the wheel is now turned one encoder count in the positive (CW) direction, so that CURR_FQDPC = 0x0000 and PREV_FQDPC = 0xFFFF. What is the new value of TOTAL when the incorrect value from part (8a) is updated with faulty casting?

9. Complete *fqd.c* and hand in one copy of the code per group.

4 In-Lab Assignment

4.1 Reading the Counter

1. Write a simple C program called *lab2.c* that uses the `init_FQD` and `ReadFQD_pc` functions from *fqd.c*. Create a `uint16_t` variable, `counter`, in your program and store the values returned by `ReadFQD_pc` in this variable. You will be able to examine the count value of the decoder by looking at this variable when debugging.

2. Compile and debug your *lab2.c* program using the makefile by entering `gmake` at the command shell prompt. Make sure that all your files are in their respective directories.

3. Power up the evaluation and the interface board with the on/off switch at the MPC5553EVB and use the P&E Debugger to download the *lab2.elf* file to the processor. Connect the encoder to the Enc_2 socket on the interface board. For this lab, do not connect the power supply to the motor; the encoder works without external power.

4. In the debugger, right click in the Variables Window and select Add Variable, then type the variable name `counter` into the menu. Display the value in Decimal, and use the Default data type. Click OK and you should now see `counter` in the Variables Window.

5. Run the *lab2.c* program by clicking the High Level Go button in the debugger. Then, stop running *lab2.c* by clicking the 'Stop' button in the debugger, which is shown as a black square. Right-click the code window, choose 'Select Source Module' and find `lab2.c` in the list. Set a breakpoint just after the line where the counter is updated and click the High Level Go button again. The value of `counter` in the Variables Window should now change if the wheel of the haptic device is moved slightly between two runs. If not, correct the problem before proceeding.

6. Reset the target by clicking on the 'Reset' lightning bolt button in the debugger. You will see code run by in the Status Window and the Source Window is cleared. Download and run *lab2.elf* again and turn the wheel of the haptic device one full revolution. How much does the variable `counter` increment during one full revolution of the haptic wheel? Is this answer consistent with your answer to the Pre-Lab question?

4.2 Overflow and Underflow

7. Use your knowledge from Lab 1 to output the value of `counter` to the 16 LEDs. Recompile your code using the same command as above. Download and run your code.

8. With your new code running, turn the wheel until the LED counter shows 0x8000. This is the center position for the counter. Now, turn the wheel until the LED counter either underflows or overflows. How many revolutions of the wheel does it take to overflow a 16-bit register starting at this center value?

9. Show the running LED counter to the assistants and provide the solution to the above question.

4.3 Faulty Casting

10. Modify your code so that you use your `updateCounter` function to keep track of the position count. Output the middle 16 bits of your position counter to the LEDs. You can do this by using a bitwise shift. Verify that your `updateCounter` function works correctly.

11. Repeat step 9, but modify `updateCounter` to use the faulty casting method from the "Notes on Casting" section 2.3. Verify that there is in fact a problem with this code and show that problem to the assistants.

12. **Modify `updateCounter` again to use the correct casting method to proceed with the labs.**

4.4 Angle Calculation

13. Write a simple C program called *lab2angle.c* that outputs how much the haptic wheel turns in degrees. Besides the LED output (lower 16 bits), use the serial I/O functions to output the counts and the corresponding angle, using code similar to:

```
char buff[32];
sprintf(buff, "\fvalue = %i\r", value);
serial_puts(1,buff);
```

    Include the necessary header files and compile your program by entering `gmake lab2angle` at the command shell prompt. Open *TeraTerm* to check the serial output.

14. Show the running code to the assistants.

5 Post-Lab Assignment

1. Read the description of slow, normal, and fast decoding modes in Section 3.1 of application note AN2842.
   (a) What is the difference in the amount of computations performed by each of these three modes?
   (b) Based on this difference, which mode would you expect to execute more quickly?
   (c) Next, read the estimates of eTPU busy time found in Table 2 of AN2842. Why, in fact, does normal mode require more computation cycles than slow mode? You may wish to review the description of noise immunity in Section 3.2.
2. Suppose that the QD function in slow mode requires 150 eTPU clock cycles to process one rising or falling edge of a quadrature signal. Given that the eTPU clock is equal to the system clock, which is set to 120 MHz, what is the maximum rate at which the haptic wheel may turn, in revolutions/second, before the eTPU fails to process all edges?
3. Suppose that the haptic wheel is turning at the maximum rate allowed by the eTPU, as computed in question 2. What is the slowest rate at which the CPU can read the eTPU counter register without losing track of the direction of rotation? Calculate this rate for both the 16 bit counter that we use in the lab as well as for the 24 bit counter that is available in the hardware.

If you have comments that you feel would help improve the clarity or direction of this assignment, please include them following your post-lab solutions.

6 Table and Figures for Pre-Lab Questions

They will be collected at the beginning of the in-lab.

Previous State (X_{i-1},Y_{i-1})	Current State (X_i,Y_i)	Action Error, Increment, or Decrement
00	00	
00	01	Decrement
00	10	Increment
00	11	
01	00	
01	01	
01	10	
01	11	
10	00	
10	01	
10	10	
10	11	
11	00	
11	01	
11	10	
11	11	Error

Table 1: Encoder Transition Table for pre-lab problem 1.

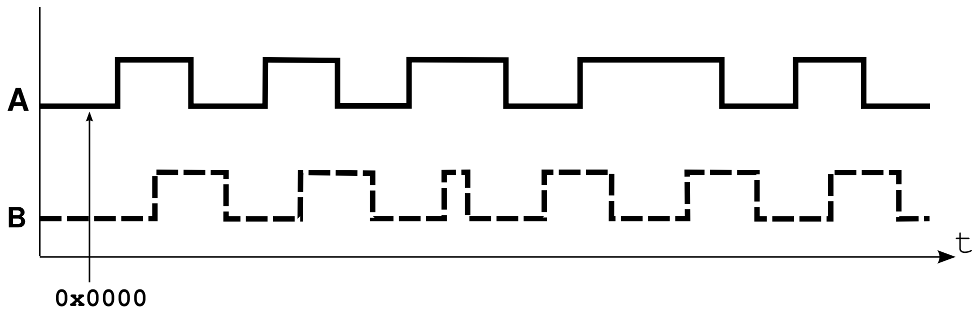

Figure 2: Sample output for pre-lab problem 3a. The 16-bit counter is initialized to 0x0000.

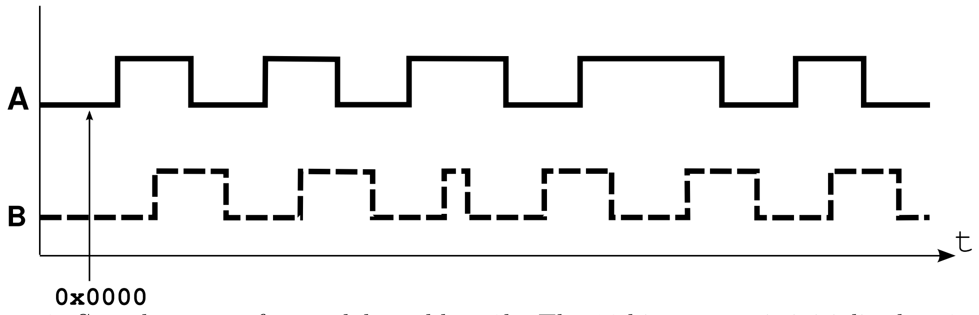

Figure 3: Sample output for pre-lab problem 3b. The 16-bit counter is initialized to 0x0000.