Introduction to Rotary Encoders

Tyler Petty

CS 566 Laboratory 6 Report

1 Objective

The purpose of this lab is to learn how to detect a motor's position and rotations per minute RPM by detecting the changes in the PHA and PHB signals.

2 Apparatus

Experimentation used the Texas Instruments Tiva C Series EK-TM4C123GXL Launchpad, 300CPR Rotory Encoder attached to motor, custom pcb breakout board, Flexable Flat Cable FFC and additional jumper wires.

3 Method

For setup the rotory encoder and motor are connected to the custom pcb board with a flex cable. From the custom pcb, the rotory encoder phase pins are connected to the Tiva board along with the 3.3V and Ground pins. Table 1 describes the entire wiring while figure 1 shows the complete setup.

Using this setup, code was writen to detect the values of the PHA and PHB signals from the rotary encoder and, using a nested switch statement, detect the direction and magnitude of change in position by observing the difference between the previous PHA and PHB values and compare them to the current values. Velocity is calculated using a circular array of times, which is updated with the current time minus the previous time each time one of the phase signals change. The average of this array is then used in calculating the RPM of the motor.

The following code shows the interrupt function used to update the encoder position and time array when a change in PHA or PHB is detected.

```
static void
_interruptHandlerPortE(void)
{
    uint32_t mask = GPIOIntStatus(GPIO_PORTE_BASE, 1);
    if ((mask & PHA) || (mask & PHB))
    {
      uint8_t phVal, phaVal, phbVal;
      phVal = GPIOPinRead(GPIO_PORTE_BASE, (PHA|PHB));
      phaVal = (phVal & PHA) >= 1?1:0;
      phbVal = (phVal & PHB) >= 1?1:0;

      update_position(phaVal, phbVal);
      insert_time((UINT32_MAX - TimerValueGet(TIMER0_BASE, TIMER_A)));
    }
    GPIOIntClear(GPIO_PORTE_BASE, mask);
}
```

The interrupt then calls two functions, update_position() and insert_time() from timing_position.c. Update_position() is shown below and as described earlier, it consists of a nested switch statement in order to detect the direction of change of the motor. The magnitude of change is simply a summation of the changes in direction.

```
void update_position(uint8_t pha, uint8_t phb) {
    uint8_t position_val = (pha << 1) | phb;
    switch(previous_position) {
      case (0):
        switch(position_val) {
          case (0): break;
          case (1): ++position; break;
          case (2): --position; break;
          case (3): break;
        }
        break;
      case (1):
        switch(position_val) {
          case (0): --position; break;
          case (1): break;
          case (2): break;
          case (3): ++position; break;
        }
        break;
      case (2):
        switch(position_val) {
          case (0): ++position; break;
          case (1): break;
          case (2): break;
          case (3): --position; break;
```

```
        }
        break;
      case (3):
        switch(position_val) {
          case (0): break;
          case (1): --position; break;
          case (2): ++position; break;
          case (3): break;
        }
        break;
      default:
        #ifdef USB_SERIAL_OUTPUT
          UARTprintf("Error\n");
        #endif
    }
    previous_position = position_val;
    total_count += 1;
}

void insert_time(uint32_t time) {
  static uint32_t previous_time;
  if(previous_time < time)
    arr_time[pos] = time - previous_time;
  else
    arr_time[pos] = (time+0x7fffffff) - (previous_time - 0x7fffffff);
  pos = (pos + 1) % DATAPOINTS;
  previous_time = time;
}
```

In accordance to the lab requirements, a simple idle task was emplemented to print out the current position of the motor as well as the RPM. The RPM is calculated using two functions, avg_ticks_per_interrupt() and motor_get_rpm().

```
int avg_ticks_per_interrupt() {
    int curr_point = pos;
    int32_t time_diff = 0;
    uint32_t sum_time_diff = 0;
    int count = 0;
    for ( int i = 0; i < DATAPOINTS - 1; i++ ) {
        uint32_t timeA = arr_time[curr_point];
        uint32_t timeB = arr_time[(curr_point+1)%DATAPOINTS];

        time_diff = timeB - timeA;
        sum_time_diff += time_diff;
        UARTprintf("Sum: %d, TA: %d, TB: %d\n", sum_time_diff, timeA, timeB);
    }

    return (sum_time_diff/DATAPOINTS) ;
}

int32_t motor_get_rpm(void)
{
  return (72000/avg_ticks_per_interrupt());
}
```

4 Data

Tiva Board	Custom PCB
PC4	PHA
PC5	PHB
+3.3V	V3.3
GND	GND

Table 1: Pin layout

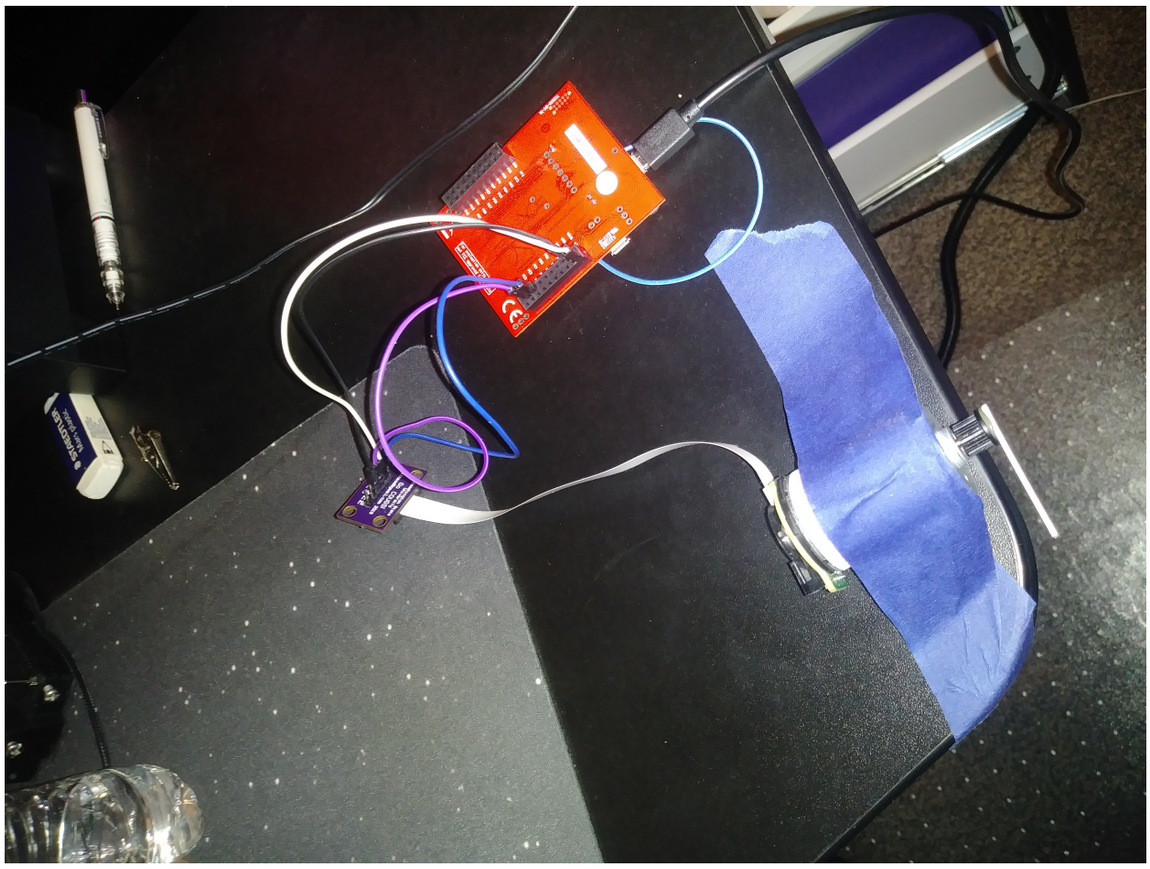

Figure 1: Breadboard Layout

5 Results and Analysis

The following is a sample of the printout from the setup running.

Hello from main()
Position: 0 Average RPM: 0
Position: 2013 Average RPM: 3
Position: 8005 Average RPM: 10
Position: 15669 Average RPM: 15
Position: 21532 Average RPM: 7
Position: 25038 Average RPM: 6
Position: 25257 Average RPM: 0

As the sample shows, the program is able to monitor the motors current position and RPM. The average RPM is calculated from an average of 10 readings, spaced out 50ms.

5.1 Question

Q1: What are the potential errors that your RPM method may encounter?
With the current implementation, calculating RPMs is currently susceptible to errors caused by quick changes in velocity, such as bumping the motor. This would cause the phase signals to change, causing an interrupt that will update the velocity array with the time difference, causing a cascading effect on the calculations. An example of this would be if the velocity array was filled with 19 readings of 1000 clock ticks time differences between interrupts and then the motor is jarred causing the 20th entry to be 500. This would mean the average time difference would be 975 instead of 1000. The resulting RPMs would be 74 RPM instead of the expected 72.

6 Conclusion

As the copied printout in Results and Analysis shows, I was able to program the setup to detect changes in position and maintain a record of its current position as well as calculate an average RPM every 500ms.

7 Appendix

7.1 master.c Main program

```
/* Standard includes. */
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

/* Kernel includes. */
#include "FreeRTOS.h"
#include "semphr.h"
#include "task.h"

/* Hardware includes. */
#include "inc/tm4c123gh6pm.h"
#include "driverlib/sysctl.h"
#include "driverlib/timer.h"
#include "driverlib/gpio.h"
#include "inc/hw_memmap.h"

/* Optional includes for USB serial output */
#ifdef USB_SERIAL_OUTPUT
#include "driverlib/rom.h"
#include "utils/uartstdio.h"
#include "driverlib/pin_map.h"
#include "driverlib/uart.h"
#endif

/*local includes*/
#include "assert.h"

// Included for GPIO_O_LOCK and others
#include "inc/hw_gpio.h" //for macro declaration of GPIO_O_LOCK and GPIO_O_CR
#include <inc/hw_types.h>

#include "timers.h"
#include "timing_position.h"


#define SPEED_TEST 0


#define LED_R (1<<1)
#define LED_G (1<<3)
#define LED_B (1<<2)
#define SW1   (1<<4)
#define SW2   (1<<0)
#define PHA (1<<2)
#define PHB (1<<3)

#define LED_ON(x) (GPIO_PORTF_DATA_R |= (x))
#define LED_OFF(x) (GPIO_PORTF_DATA_R &= ~(x))
#define LED(led,on) ((on)?LED_ON(led):LED_OFF(led))
#define pdTICKSTOMS( xTicks ) ((xTicks * 1000 ) / configTICK_RATE_HZ )

// static SemaphoreHandle_t _semPhB = NULL;

uint32_t SystemCoreClock;
uint16_t RPMs[10];
uint8_t RPM_pointer = 0;


#ifdef USB_SERIAL_OUTPUT

//*****************************************************************************
//
//: Configure the UART and its pins.  This must be called before UARTprintf().
//
//*****************************************************************************
```

```
static void
_configureUART(void)
{
    //
    // Enable UART0
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    //
    // Configure GPIO Pins for UART mode.
    //
    GPIOPinConfigure(GPIO_PA0_U0RX);
    GPIOPinConfigure(GPIO_PA1_U0TX);
    GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

    //
    // Use the internal 16MHz oscillator as the UART clock source.
    //
    UARTClockSourceSet(UART0_BASE, UART_CLOCK_PIOSC);

    //
    // Initialize the UART for console I/O.
    //
    UARTStdioConfig(0, 115200, 16000000);
}

#endif


static void
_interruptHandlerPortE(void)
{
    uint32_t mask = GPIOIntStatus(GPIO_PORTE_BASE, 1);
    if ((mask & PHA) || (mask & PHB))
    {
      uint8_t phVal, phaVal, phbVal;
      phVal = GPIOPinRead(GPIO_PORTE_BASE, (PHA|PHB));
      phaVal = (phVal & PHA) >= 1?1:0;
      phbVal = (phVal & PHB) >= 1?1:0;

      update_position(phaVal, phbVal);
      insert_time((UINT32_MAX - TimerValueGet(TIMER0_BASE, TIMER_A)));
    }
    GPIOIntClear(GPIO_PORTE_BASE, mask);
}

static void
_setupHardware(void)
{
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);

    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, (LED_G|LED_R|LED_B));

    GPIO_PORTE_CR_R = PHA | PHB;
    GPIOPinTypeGPIOInput(GPIO_PORTE_BASE, (PHA|PHB));
    GPIOIntRegister(GPIO_PORTE_BASE, _interruptHandlerPortE);
    GPIOIntTypeSet(GPIO_PORTE_BASE, (PHA|PHB), GPIO_BOTH_EDGES);

    GPIOPadConfigSet(GPIO_PORTE_BASE, (PHA|PHB), GPIO_STRENGTH_2MA, GPIO_PIN_TYPE_STD_WPU);

    SystemCoreClock = 80000000;  // Required for FreeRTOS.

    SysCtlClockSet( SYSCTL_SYSDIV_2_5 |
                    SYSCTL_USE_PLL |
                    SYSCTL_XTAL_16MHZ |
                    SYSCTL_OSC_MAIN);
    ROM_SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
                       SYSCTL_XTAL_16MHZ);

}
```

```
static void
_setupTimer(void)
{
  SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
  spinDelayMs(10);
  TimerClockSourceSet(TIMER0_BASE, TIMER_CLOCK_SYSTEM);
  TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);
  TimerLoadSet(TIMER0_BASE, TIMER_A, UINT32_MAX);
  TimerEnable(TIMER0_BASE, TIMER_A);
}

static void
_heartbeat( void *notUsed )
{
   timing_init(GPIOPinRead(GPIO_PORTE_BASE, PHA)  & PHA, GPIOPinRead(GPIO_PORTE_BASE, PHB)  & PHB);

   uint32_t green500ms = 500; // 1 second for on off
   uint32_t ledOn = 1;


   while(true)
   {
     vTaskDelay(green500ms / portTICK_RATE_MS);
     LED(LED_G, ledOn);
     ledOn = !ledOn;
    }
}

void _rpm_update(void * notUsed)
{
    while(1) {
      RPMs[RPM_pointer] = motor_get_rpm();
      RPM_pointer = (RPM_pointer + 1) % 10;
      vTaskDelay(50 / portTICK_RATE_MS);
    }
}

uint16_t rpm_average(void)
{
  uint32_t sum = 0;
  for (int i = 0; i < 10; i++) {
    sum += RPMs[i];
  }
  return (sum/10);
}

void
_idle_print_out (void * notUsed)
{
  GPIOIntEnable(GPIO_PORTE_BASE, (PHA|PHB));
  while(1) {
    while(1) {
      #ifdef USB_SERIAL_OUTPUT
        UARTprintf("Position: %d Average RPM: %d\n", motor_get_position(), rpm_average());
      #endif
      vTaskDelay(500 / portTICK_RATE_MS);
    }
  }
}

int main( void )
{
    _setupHardware();
    #ifdef USB_SERIAL_OUTPUT
            void spinDelayMs(uint32_t ms);
            _configureUART();
      _setupTimer();
            spinDelayMs(1000);  // Allow UART to setup

            UARTprintf("\n\n\nHello from main()\n");
    #endif
```

```
    // while(1) {
    //   uint32_t a = TimerValueGet(TIMER0_BASE, TIMER_A);
    //   spinDelayMs(1000);
    //   uint32_t b = TimerValueGet(TIMER0_BASE, TIMER_A);
    //   UARTprintf("%i\n", a-b);
    //
    // }

    xTaskCreate(_rpm_update,
                "update_rpm",
                configMINIMAL_STACK_SIZE,
                NULL,
                tskIDLE_PRIORITY + 2,
                NULL );

    xTaskCreate(_heartbeat,
                "green",
                configMINIMAL_STACK_SIZE,
                NULL,
                tskIDLE_PRIORITY + 1,  // higher numbers are higher priority..
                NULL );

    xTaskCreate(_idle_print_out,
                "printout",
                configMINIMAL_STACK_SIZE,
                NULL,
                tskIDLE_PRIORITY,
                NULL );


    /* Start the tasks and timer running. */
    vTaskStartScheduler();

    assert(0); // we should never get here..

    return 0;
}
```

7.2 timing_position.c Timing and position calculations

```
#include <stdint.h>
#include <stdlib.h>
#include "timing_position.h"
#define TRUE 1
#define FALSE 0

uint32_t arr_time[DATAPOINTS];
uint8_t pos;
int32_t position = 0;
uint8_t previous_position = 0;
int total_count;

int motor_get_position(void)
{
  return position;
}

void timing_init(uint8_t pha, uint8_t phb) {
    pos = 0;
    total_count = 0;
    previous_position = (pha << 1) | phb;
    for (uint8_t i = 0; i < DATAPOINTS; i++) {
      arr_time[i] = 0;
    }
}

void insert_time(uint32_t time) {
  static uint32_t previous_time;
  if(previous_time <= time)
    arr_time[pos] = time - previous_time;
  else
    arr_time[pos] = (time+0x7fffffff) - (previous_time - 0x7fffffff);
```

```
  pos = (pos + 1) % DATAPOINTS;
  previous_time = time;
}

void update_position(uint8_t pha, uint8_t phb)
{
    uint8_t position_val = (pha << 1) | phb;
    switch(previous_position) {
      case (0):
        switch(position_val) {
          case (0): break;
          case (1): ++position; break;
          case (2): --position; break;
          case (3): break;
        }
        break;
      case (1):
        switch(position_val) {
          case (0): --position; break;
          case (1): break;
          case (2): break;
          case (3): ++position; break;
        }
        break;
      case (2):
        switch(position_val) {
          case (0): ++position; break;
          case (1): break;
          case (2): break;
          case (3): --position; break;
        }
        break;
      case (3):
        switch(position_val) {
          case (0): break;
          case (1): --position; break;
          case (2): ++position; break;
          case (3): break;
        }
    }
    previous_position = position_val;
}


int avg_ticks_per_interrupt() {
    int curr_point = pos;
    int32_t time_diff = 0;
    uint32_t sum_time_diff = 0;
    int count = 0;
    for ( int i = 0; i < DATAPOINTS - 1; i++ ) {
        sum_time_diff += arr_time[i];
    }

    return (sum_time_diff/DATAPOINTS) ;
}

int32_t motor_get_rpm(void)
{
  return (72000/avg_ticks_per_interrupt());
}
```