EEL4742C: Embedded Systems

Experiment Ten:

ADC

Ryan Koons 11/07/2020

Introduction: This lab is primarily focused in configuring, utilizing the ADC, as well as the joystick on the boosterpack. The ADC that is focused on is an Successive Approximation Register type; which function using an array of capacitors to charge up and based on the quantization, convert to a certain bit resolution. Prior to configuring the ADC, first hand calculations have to be done to determine the clock cycles necessary for the ADC application; this will be explained in detail in the next section. The ADC uses a configurable range of reference voltages and a pseudo-binary search tree algorithm, to determine the bits that represent a set proportion of the reference voltage. That is, the ADC can generate various voltage levels and then essentially operate like a comparator to perform a proper conversion. In the next section, the sample and hold time (SHT) will also be computed. This value needs to ensure there is enough time to charge all of the capacitors, without risking the power-performance of the ADC and charging the capacitors for an unnecessary amount of time. As far as clock signal goes, MODOSC will be used to operate the ADC; performance times will have to be calculated based on the frequency that MODOSC provides. For this lab, performance will be focused on (speed), but using a different clock divider on MODOSC can also conserve on power. Lastly, the joystick and ADC will have to be configured via GPIO pin diversion, and Configuration Registers, this will be covered in more detail in the next section. It is also worth noting that the ADC can convert 32 analog channels and store them in 32 memory registers simultaneously.

10.1 ADC-Horizontal Joystick: The focus of this program is to become familiar with the ADC, as well as the joystick on the booster pack. Since, the joystick is built around a potentiometer, an ADC is necessary to convert the signal and display it on the PC. This program requires configuring the ADC, and reading only the horizontal joystick pot value. Note: UART functions from previously labs will be used to easily print joystick results to the display.

Within initialization, delay loop counters, formatting parameters and strings are initialized. Also, the init_UART and init_ADC functions are called. For init_ADC, a template was revised in order to properly configure the necessary registers for the ADC to work in an expected manner. Prior to any configuration, first the designated GPIO pins were re-configured to work with the joystick. To do so, the boosterpack guide was consulted to find the pins on the header for the vertical and horizontal joystick. Then this also had to be determined on the launchpad. E.g. J3.26 (Vert) on the booster pack coordinated with P8.7/A4; this is therefore the appropriate analog channel number for this signal. According to the family users guide, P8SEL1, P8SEL0, P9SEL0, and P9SEL1 and all of there corresponding bits were all set to configure the ADC via GPIO pin diversion. Next the ADC had to be turned on by bit-masking with ADC12CTL0. Note: All of the bitmasks necessary for configuring the ADC registers can be found exactly in the MSP430 header file. The "fields" in the family user guide are not a valid alternative for bit-masking in this situation.

The first configuration register: ADC12CTLO is then used to select the number of cycles (SHT). This was calculated by hand, prior coding. First, the minimum SHT was calculated as t>=

(R_1 + R_s) * (C_1+C_ext)*ln(2^n+2). R1 and C1 were found (and max value was selected) in the microcontroller data sheet. Rs and C_ext were found in the joystick guide. N is the number of bits for resolution, the desired for this case was 12-bit resolution. The minimum sample and hold time was calculated to be 3.11 microseconds. Additionally, a divider of 1 will be used to yield the fastest possible ADC conversion; alternatively, a larger clock divider (8) could be used in order to maximize the performance of the embedded system. This value was then multiplied by the largest MODOSC frequency (5.4 Mhz) to ensure that the longest case was being accounted for. The result was 16.2 cycles, but the next number above was selected from the provided list. Therefore, the desired number of cycles for this operation was 32. The family user-guide showed that ADC12HT03 would accomplish that clock and number of cycles, and the corresponding bit mask “ADC12SHT0_3” was used for the operation.

The next register: ADC12CTL1 was used to select the trigger bit, the divider and the clock select. ADC12CTL1 |= ADC12SHS_0|ADC12SHP|ADC12DIV_0|ADC12SSEL_0; was used in order to select the ADC12SC bit as the trigger, a clock divider of 1 (for fastest results), and MODOSC for the clock select. ADC12CTL2 was then used t select the bit resolution and an unsigned result (which happened to be default). SO, ADC12RES_2 was found from the family users guide and implemented in the code. Lastly, ADC12MCTL0 was used to configure the specific channel for measurement (horizontal). The appropriate reference voltage was selected, as well as analog input channel 10 (for A10—horizontal), via bit-masking (again grabbed from the MSP430 Header file). The last step of the initialization is to turn on the ENC bit to enable conversion.

The rest of the code is an infinite loop that provides a brief delay between ADC results. Each loop iteration sets the trigger bit, waits for the ADC12BUSY flag to clear. It then writes ADC12MEM0 (result) to the PC via uart_write_uint16() function. Then some formatting, is used to move the cursor to the front of the next line, and the red LED is toggled. The code shows a value of about 2000 when the joystick is horizontally at the origin, then a value of about 2000 at the left, and a value of about 4000 when all the way at the right. These results are correct since the joystick is essentially a potentiometer; when all the way to left it is fairly close to 0 except for marginal resistance. The maximum resistance is found all the way to the right, and it is about in the middle around the origin.

Analysis: The values of Ri and Ci that were chosen from the data sheet (p 60) were 10 k-ohm and 15 pF respectively. The range for Ri was 0.5 ->10 or 1 to 10 (depending on operating voltage). And the range of Ci was 10 to 15 pF. The maximum values of each were selected since the maximum case would have the longest simple time, and the program should be able to account for the maximum amount of time. Also, the upper range (operating voltage) was selected since the board is operating at a higher supply voltage. # of cycles= (10+10)k_ohm *(15+1)pf *ln2^12+2   *5.4Mhz= 16.2 cycles. Thus, the higher value of 32 cycles is selected.

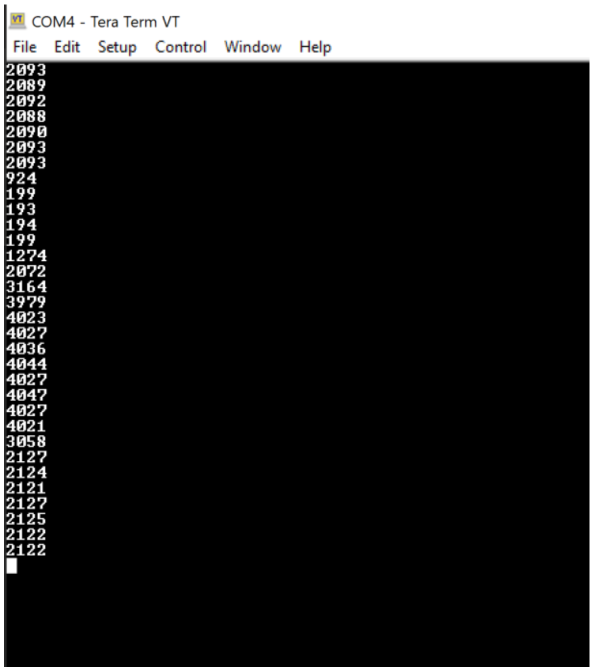
10.1 CCS Code:

```
//Lab 10.1: Configure ADC with horizontal Joystick
#include <msp430.h>
#define FLAGS UCA1IFG //Contains the Tx and Rx flags
#define RXFLAG UCRXIFG //Rx Flag
#define TXFLAG UCTXIFG //Tx Flag
#define TXBUFFER UCA1TXBUF// Tx Buffer
#define RXBUFFER UCA1RXBUF// Rx Buffer
#define redLED BIT0    //red LED at P1.0
#define greenLED BIT7  //green LED at P9.7
#define BUT1 BIT1     //Button 1 at P1.1
#define BUT2 BIT2     //BUtton 2 at P1.2

void init_adc(void); //function for initializing the ADc
void init_UART(void); //alternate UART Init. Function
void uart_write_char(unsigned char ch);
void uart_write_uint16(unsigned int n);


void main(void)
{
     unsigned int n=0; //incrementing 16-bit integer
     unsigned int i; //delay loop
    unsigned char nl='\n';//formatting parameters
    unsigned char cr='\r';
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
     PM5CTL0 &= ~LOCKLPM5;       //enable GPIO Pins

     //LED init.
     P1DIR |= redLED; //set as output
     P9DIR |= greenLED;
     P1OUT &= ~redLED;    //red OFF
     P9OUT &= ~greenLED; //green OFF
     init_UART(); //Initialize
     init_adc();



//infinite loop, set ADC12SC bit to start conversion
     //wait for ADC12BUSY bit to clear
     //
     while(1)
     {             //delay between numbers
                 for(i=0; i<65000; i++){}

                 //set ADC12SC bit
                 ADC12CTL0 |= ADC12SC; //set, start conversion for adc
                 //wait for flag to clear
```

```
                  while( (ADC12CTL1 & ADC12BUSY) != 0 ){} //wait here, use !=0, since
there could be other bits in bit field

                     uart_write_uint16(ADC12MEM0); //print number to screen
                     uart_write_char(nl); //Tx newline (down a line)
                     uart_write_char(cr); //Tx carriage return (leftmost column)
                     P1OUT ^= redLED; //toggle LEDS

                     //n++;
             }

}


void init_adc() //function to init. adc
{
// Divert the pins to analog functionality
// X-axis: A10/P9.2, for A10 (P9DIR=x, P9SEL1=1, P9SEL0=1)
    P9SEL1 |= BIT2;
    P9SEL0 |= BIT2;
//Y-axis: A4/P8.7, for A4(P8DIR=x P8SEL1/0=1)
     P8SEL0 |= BIT7;
     P8SEL1 |= BIT7;


// Turn on the ADC module
    ADC12CTL0 |= ADC12ON;

// Turn off ENC (Enable Conversion) bit while modifying the configuration
    ADC12CTL0 &= ~ADC12ENC;

//*************** ADC12CTL0 ***************/ //set number of cycles (fast mode)
    // Set ADC12SHT0 (select the number of cycles that you determined)
    ADC12CTL0 |= ADC12SHT0_3;  //32 cycles selected for bit field bits 8-111
    //*************** ADC12CTL1 ***************
    // Set ADC12SHS (select ADC12SC bit as the trigger)
    // Set ADC12SHP bit
    // Set ADC12DIV (select the divider you determined)
    // Set ADC12SSEL (select MODOSC)

    //ADC12SHS should be defaulted to use ADC, but just in case
    ADC12CTL1 |= ADC12SHS_0|ADC12SHP|ADC12DIV_0|ADC12SSEL_0;
//note: the last three value should all be defults, but they are stil set just in
case


    //*************** ADC12CTL2 ***************
    // Set ADC12RES (select 12-bit resolution)
    ADC12CTL2 |= ADC12RES_2; //select 12 bit resolution //should be default, and
unsigned binary
    //Omit Binary unsigned bitmask to have 0.
    // Set ADC12DF (select unsigned binary format)
    //this value is default
    //ADC12DF=0;  //default
    //*************** ADC12CTL3 ***************
    // Leave all fields at default values

    //*************** ADC12MCTL0 ***************

    // Set ADC12VRSEL (select VR+=AVCC, VR-=AVSS)
    //should be default
   // ADC12VRSEL=0;
    ADC12MCTL0 |= ADC12INCH_10| ADC12VRSEL_0;
    // Set ADC12INCH (select channel A10)

    // Turn on ENC (Enable Conversion) bit at the end of the configuration
    ADC12CTL0 |= ADC12ENC;
    return;
    }
```

```
//UART config
//4800 Baud, 8-bit data, LSB first, no parity, 1 stop bit, no flow CTL
//Initial clock: ACLK @ 32768 Hz, no oversampling
//UART config
//9600 Baud, 8-bit data, LSB first, no parity, 1 stop bit, no flow CTL
//Initial clock: SMCLK @ 1.000 MHz with Oversampling (16x)
void init_UART(void)
{
    //Divert GPIO pins for UART
        P3SEL1 &= ~(BIT4|BIT5);  //Using 3.4--UCA1TXD, 3.5--UCA1RXD
        P3SEL0 |= (BIT4|BIT5);

        //Use SMCLK, other settings default
        UCA1CTLW0 |= UCSSEL_2;

        //Configure clock dividers and modulators
        //UCBR=6, UCBRF=8, UCBRS=0x20, UCOS16=1 (over-sampling)
        UCA1BRW =6;
        //bit masking for UCBRS aand UCBRF done using 0-7 bit number
        UCA1MCTLW = UCBRS5|UCBRF3|UCOS16;


        //exit reset state, to begin Tx/Rx
        UCA1CTLW0 &= ~UCSWRST;
}


void uart_write_uint16(unsigned int n)
{
 int digit; //digit for parsing through
 if(n>=10000)
 {
     //modulo 10 ensures no digits past 16-bit int allowed (out of constraint)
     digit = (n/10000) % 10; //takes leftmost digit first
     uart_write_char('0' + digit); //pass correct ascii value for digit to computer
 } ///1000

 if(n>=1000)
  {
      digit = (n/1000) % 10; //takes next leftmost digit
      uart_write_char('0' + digit);
  }
//modulo is also used to hack off other digits of int that is being displayed
 if(n>=100)
  {
      digit = (n/100) % 10;//take next digit
      uart_write_char('0' + digit); //print
  }

 if(n>=10)
  {  //process repeats similarly
      digit = (n/10) % 10;
      uart_write_char('0' + digit);
  }
//ones place always executes
 digit = n%10;
 uart_write_char('0'+digit);

 return;

}

//TXFLAG: 1 When ready to transmit, 0 during transmission
void uart_write_char(unsigned char ch)
{
    //wait for any ongoing transmission  to complete
    while((FLAGS & TXFLAG) ==0 ){}

    //Write byte to the transmit buffer
    TXBUFFER= ch;
}
```

10.1 Output (Tera Term):



```
2093
2089
2092
2088
2090
2093
2093
924
199
193
194
199
1274
2072
3164
3979
4023
4027
4036
4044
4027
4047
4027
4021
3058
2127
2124
2121
2127
2125
2122
2122
```

10.2 Horizontal and Vertical Joystick-ADC: The goal of this program is to understand how to take measurement from simultaneous analog channels and in doing so, measure the vertical and horizontal joystick results.

The init_ADC() function is very similar to the previous section, but has to introduce even more configuration in order to add the ability to measure multiple channels at a time. Note: To see how to divert GPIO pin for joystick vertical measurement, see previous section. For this section, only the new lines of code in this function will be covered. In ADC12CTL0, the ADC12MSC bit is set to enable multiple sample and conversion. In ADC12CTL1, ADC12CONSEQ_1 is used to select sequence of channels (this is found in the family users guide). In ADC12CTL3, ADC12STARTADD_0 is bit-masked in order to designate where the first conversion is (ADC12MEM0). And then in ADC12MCTL1, analog input channel 4 is selected, the same reference voltage is used, and ADC12EOS is set to designate the last conversion (ADCMEM1). Similar to part one, then ADC12ENC is used to enable conversion (this designates the end of configuration).

The main function is similar to the previous section, but includes additional formatting in order to appropriately display the horizontal and vertical joystick measurements. In order to do so, the Uart_write string and Uart_write_uint16 functions are used. The result for the horizontal measurement is stored in ADC12MEM0, and ADCMEM1 stores the vertical result. As far as the measurement for the vertical and horizontal potentiometers. They follow the same trend. When a measurement is in the middle of its range it is around 2000. At its positive max it is around 4000, and the min for its range (negative axis) it is around 100. So (x,y) for left, up, right, and down should read about (100,2000), (2000,4000), (4000,2000), and (2000, 100), respectively.

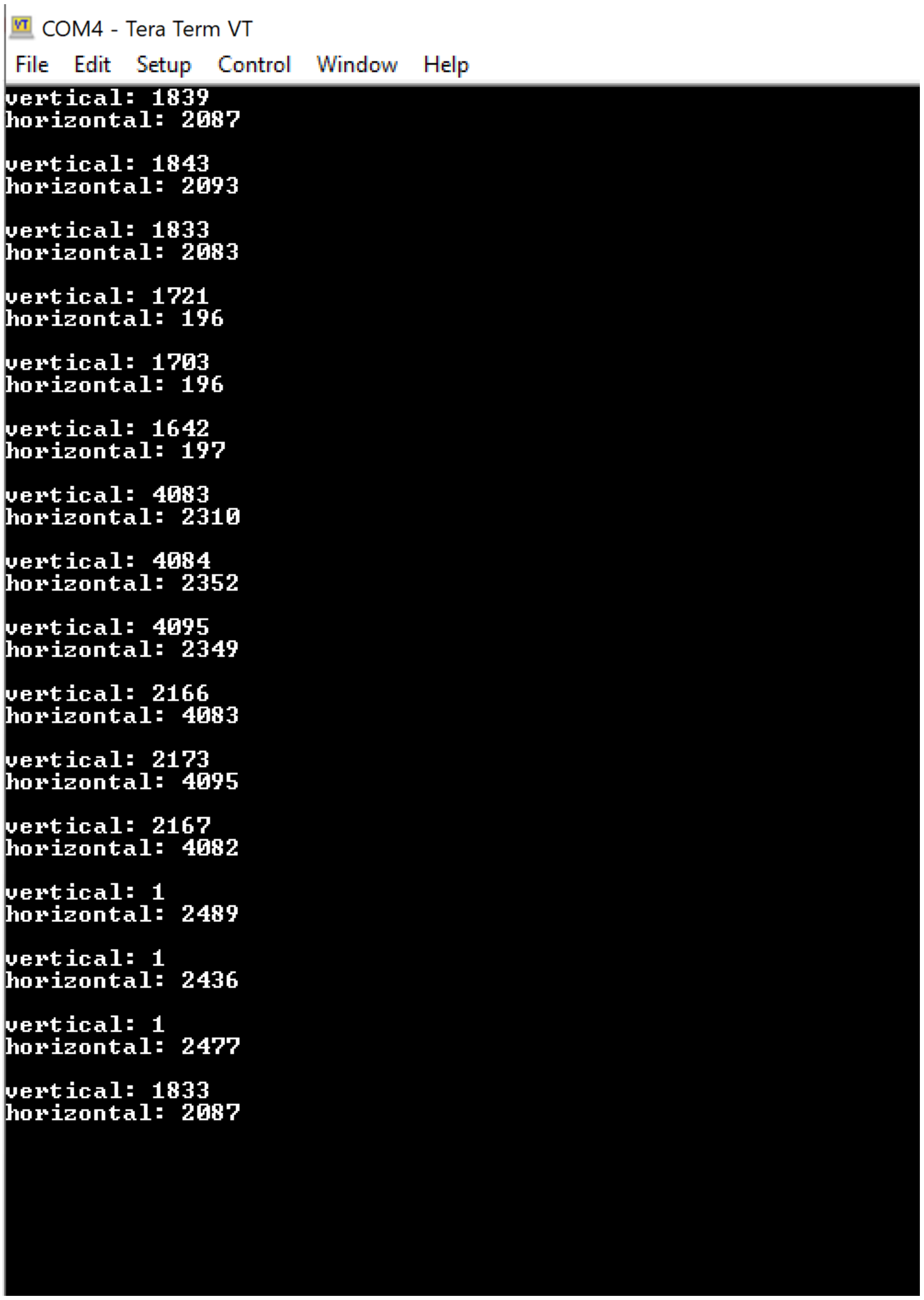
10.2 CCS Code:

```
//Lab 10.2: Vertical and Horizontal Joystick-ADC
#include <msp430.h>
#define FLAGS UCA1IFG //Contains the Tx and Rx flags
#define RXFLAG UCRXIFG //Rx Flag
#define TXFLAG UCTXIFG //Tx Flag
#define TXBUFFER UCA1TXBUF// Tx Buffer
#define RXBUFFER UCA1RXBUF// Rx Buffer
#define redLED BIT0    //red LED at P1.0
#define greenLED BIT7  //green LED at P9.7
#define BUT1 BIT1     //Button 1 at P1.1
#define BUT2 BIT2     //BUtton 2 at P1.2

void init_adc_2d(void); //function for initializing the ADc
void init_UART(void); //alternate UART Init. Function
void uart_write_char(unsigned char ch);
void uart_write_uint16(unsigned int n);
void uart_write_string(char * str);




void main(void)
{
     unsigned int n=0; //incrementing 16-bit integer
     unsigned int i; //delay loop
    unsigned char nl='\n';//formatting parameters
    unsigned char cr='\r';
    char vert[] ="vertical: ";
    char horiz[] ="horizontal: ";
    char newline[] = "\n";
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
     PM5CTL0 &= ~LOCKLPM5;       //enable GPIO Pins

     //LED init.
     P1DIR |= redLED; //set as output
     P9DIR |= greenLED;
     P1OUT &= ~redLED;    //red OFF
     P9OUT &= ~greenLED; //green OFF
     init_UART(); //Initialize
     init_adc_2d(); //Init ADC



//infinite loop, set ADC12SC bit to start conversion
     //wait for ADC12BUSY bit to clear
     //
     while(1)
     {             //delay between numbers
                for(i=0; i<65000; i++){}

                //set ADC12SC bit
```

```
                  ADC12CTL0 |= ADC12SC; //set, start conversion for adc
                  //wait for flag to clear
                  while( (ADC12CTL1 & ADC12BUSY) != 0 ){} //wait here, use !=0, since
there could be other bits in bit field
                     uart_write_string( vert); //print formatting string to screen
                     uart_write_uint16(ADC12MEM1); //print number to screen
                     uart_write_char(nl); //Tx newline (down a line)
                     uart_write_char(cr); //Tx carriage return (leftmost column)
                     uart_write_string( horiz); //print formatting string to screen
                     uart_write_uint16(ADC12MEM0); //print number to screen
                     uart_write_char(nl); //Tx newline (down a line)
                     uart_write_char(cr); //Tx carriage return (leftmost column)
                     uart_write_char(nl); //Tx newline (down a line)
                     P1OUT ^= redLED; //toggle LEDS

                     for(i=0; i<65000; i++){}//delay loop

                     //n++;
              }

}

void uart_write_string(char * str)
{
 int i=0;//parsing integer
 while( str[i] != '\0')//while not at end of string
 {
     uart_write_char(str[i]);//pass current char
     i++;
 }
 return;
}


void init_adc_2d() //function to init. adc
{
// Divert the pins to analog functionality
// X-axis: A10/P9.2, for A10 (P9DIR=x, P9SEL1=1, P9SEL0=1)
    P9SEL1 |= BIT2;
    P9SEL0 |= BIT2;
//Y-axis: A4/P8.7, for A4(P8DIR=x P8SEL1/0=1)
     P8SEL0 |= BIT7;
     P8SEL1 |= BIT7;


// Turn on the ADC module
    ADC12CTL0 |= ADC12ON;

// Turn off ENC (Enable Conversion) bit while modifying the configuration
    ADC12CTL0 &= ~ADC12ENC;

//*************** ADC12CTL0 ***************/ //set number of cycles (fast mode)
    // Set ADC12SHT0 (select the number of cycles that you determined)
    //set bit for ADC12MSC
    ADC12CTL0 |= ADC12SHT0_3 | ADC12MSC;  //32 cycles selected for bit field bits 8-
111



    //*************** ADC12CTL1 ***************
    // Set ADC12SHS (select ADC12SC bit as the trigger)
    // Set ADC12SHP bit
    // Set ADC12DIV (select the divider you determined)
    // Set ADC12SSEL (select MODOSC)
    //Set ADC12CONSEQ (select sequency of channels)

    //ADC12SHS should be defaulted to use ADC, but just in case
    ADC12CTL1 |= ADC12SHS_0|ADC12SHP|ADC12DIV_0|ADC12SSEL_0|ADC12CONSEQ_1;
//note: the last three value should all be defults, but they are stil set just in
case


    //*************** ADC12CTL2 ***************
    // Set ADC12RES (select 12-bit resolution)
    ADC12CTL2 |= ADC12RES_2; //select 12 bit resolution //should be default, and
unsigned binary
    //Omit Binary unsigned bitmask to have 0.
    // Set ADC12DF (select unsigned binary format)
    //this value is default
    //ADC12DF=0;  //default

    //*************** ADC12CTL3 ***************
    // Leave all fields at default values
    //set ADC12CSTARTADD to 0 (first conversion in ADC12MEM0)
    ADC12CTL3 |= ADC12CSTARTADD_0;

    //*************** ADC12MCTL0 & 1***************

    // Set ADC12VRSEL (select VR+=AVCC, VR-=AVSS)
    //should be default
   // ADC12VRSEL=0;
    ADC12MCTL0 |= ADC12INCH_10| ADC12VRSEL_0;
    ADC12MCTL1 |= ADC12INCH_4| ADC12VRSEL_0 |ADC12EOS;  //used for vertical result,
use analog channel 4
    //set ADC12EOS (last conversion in ADC12MEM1)

    // Set ADC12INCH (select channel A10)

    // Turn on ENC (Enable Conversion) bit at the end of the configuration
    ADC12CTL0 |= ADC12ENC;
    return;
    }
```

```
//UART config
//4800 Baud, 8-bit data, LSB first, no parity, 1 stop bit, no flow CTL
//Initial clock: ACLK @ 32768 Hz, no oversampling
//UART config
//9600 Baud, 8-bit data, LSB first, no parity, 1 stop bit, no flow CTL
//Initial clock: SMCLK @ 1.000 MHz with Oversampling (16x)
void init_UART(void)
{
    //Divert GPIO pins for UART
        P3SEL1 &= ~(BIT4|BIT5);  //Using 3.4--UCA1TXD, 3.5--UCA1RXD
        P3SEL0 |= (BIT4|BIT5);

        //Use SMCLK, other settings default
        UCA1CTLW0 |= UCSSEL_2;

        //Configure clock dividers and modulators
        //UCBR=6, UCBRF=8, UCBRS=0x20, UCOS16=1 (over-sampling)
        UCA1BRW =6;
        //bit masking for UCBRS aand UCBRF done using 0-7 bit number
        UCA1MCTLW = UCBRS5|UCBRF3|UCOS16;


        //exit reset state, to begin Tx/Rx
        UCA1CTLW0 &= ~UCSWRST;
}


void uart_write_uint16(unsigned int n)
{
 int digit; //digit for parsing through

 if(n>=10000)
 {
     //modulo 10 ensures no digits past 16-bit int allowed (out of constraint)
     digit = (n/10000) % 10; //takes leftmost digit first
     uart_write_char('0' + digit); //pass correct ascii value for digit to computer
 } ///1000

 if(n>=1000)
  {
      digit = (n/1000) % 10; //takes next leftmost digit
      uart_write_char('0' + digit);
  }
//modulo is also used to hack off other digits of int that is being displayed
 if(n>=100)
  {
      digit = (n/100) % 10;//take next digit
      uart_write_char('0' + digit); //print
  }

 if(n>=10)
  {  //process repeats similarly
      digit = (n/10) % 10;
      uart_write_char('0' + digit);
  }
//ones place always executes
 digit = n%10;
 uart_write_char('0'+digit);

 return;

}

//TXFLAG: 1 When ready to transmit, 0 during transmission
void uart_write_char(unsigned char ch)
{
    //wait for any ongoing transmission  to complete
    while((FLAGS & TXFLAG) ==0 ){}

    //Write byte to the transmit buffer
    TXBUFFER= ch;
}
```

10.2 Output (Tera Term):



```
vertical: 1839
horizontal: 2087

vertical: 1843
horizontal: 2093

vertical: 1833
horizontal: 2083

vertical: 1721
horizontal: 196

vertical: 1703
horizontal: 196

vertical: 1642
horizontal: 197

vertical: 4083
horizontal: 2310

vertical: 4084
horizontal: 2352

vertical: 4095
horizontal: 2349

vertical: 2166
horizontal: 4083

vertical: 2173
horizontal: 4095

vertical: 2167
horizontal: 4082

vertical: 1
horizontal: 2489

vertical: 1
horizontal: 2436

vertical: 1
horizontal: 2477

vertical: 1833
horizontal: 2087
```

.2

Student Q&A:

1) It takes 14 clock cycles for the ADC to convert a 12-bit result.
2) The total time would be 14+16.2 cycles =30.2 cycles or 5.59 microseconds.

3) AVCC Is applied the supply voltage Vcc, which is given as 3.3V. Likewise, the values of AVSS is applied Vss supply voltage, so it is 0 volts.
4) Ref_A can provide, 1.2V, 2.V, or 2.5V.

Conclusion: This lab was very helpful with understanding how to configure and utilize both the ADC and the joystick. The family User guide is the best source for figuring out how to configuring the configuration registers ADC12CTL0-3; however, it is important to remember that further analysis is necessary to determine number of cycles prior to configuring the ADC. The bitmasks for configuration can be explained in the family user guide under 'field'; but, the actual bitmask names should be grabbed from the MSP430 header file. This is an absolute imperative step, because the code will not compile or run if the fields from the family user guide are used instead of the bitmasked defines from the header file. For this lab the ADC was designed to be fast, but by dividing the clock more, the user can achieve a more power-efficient embedded system. For this lab the ADC is mainly reading the two potentiometer's currents (horizontal and vertical), but the ADC can obviously used to read other analog input channels usually other electrical currents. The ADC then also allows this analog input to be properly displayed on the LCD or the PC.