Lab 8 – ADC

# Youssef Samwel

# [yo800238@ucf.edu](mailto:yo800238@ucf.edu)

# EEL4742C Embedded Systems

Prof. Dr. Zakhia Abichar - Section 00419

# 3/28/2024



# **Project Description**

In this lab, we will learn using the Analog-to-Digital Converter (ADC) of type Successive Approximation Register (SAR) with Charge Redistribution. We’ll use the ADC to interface the two-dimensional joystick that’s on the Educational BoosterPack.

# **2.0 Experiment Code**

#ifndef \_\_MAIN\_INCLUDE\_\_

#define \_\_MAIN\_INCLUDE\_\_

#include <msp430fr6989.h>

#include <stdint.h>

#include <stdbool.h>

#define FLAGS UCA1IFG      // Contains the transmit & receive flags

#define RXFLAG UCRXIFG     // Receive flag

#define TXFLAG UCTXIFG     // Transmit flag

#define TXBUFFER UCA1TXBUF // Transmit buffer

#define RXBUFFER UCA1RXBUF // Receive buffer

#define BUTTON1 BIT1

#define BUTTON2 BIT2

#define MAX(x, y) (((x) > (y)) ? (x) : (y))

#define MIN(x, y) (((x) < (y)) ? (x) : (y))

int i2c\_read\_word(unsigned char i2c\_addrs, unsigned char i2c\_reg, unsigned int \*data); //

int i2c\_write\_word(unsigned char i2c\_addrs, unsigned char i2c\_reg, unsigned int data); //

void Initialize\_I2C(void);

void Initialize\_UART(void);

void config\_ACLK\_to\_32KHz\_crystal();

void uart\_write\_uint16(unsigned int n);

void uart\_write\_string(const char \*str);

void uart\_write\_nstring(void \*pStr, uint16\_t size);

bool IsButton1Pressed()

{

    return (~P1IN & BUTTON1) ? true : false;

}

bool IsButton2Pressed()

{

    return (~P1IN & BUTTON2) ? true : false;

}

void terminal\_clear()

{

    uart\_write\_string("\033[2J");

}

void terminal\_reset\_cursor()

{

    uart\_write\_string("\033[1;1H");

}

// Read a word (2 bytes) from I2C (address, register)

int i2c\_read\_word(unsigned char i2c\_address, unsigned char i2c\_reg, unsigned int \*data)

{

    unsigned char byte1, byte2;

    // Initialize the bytes to make sure data is received every time

    byte1 = 111;

    byte2 = 111;

    //\*\*\*\*\*\*\*\*\*\* Write Frame #1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    UCB1I2CSA = i2c\_address; // Set I2C address

    UCB1IFG &= ~UCTXIFG0;

    UCB1CTLW0 |= UCTR;    // Master writes (R/W bit = Write)

    UCB1CTLW0 |= UCTXSTT; // Initiate the Start Signal

    while ((UCB1IFG & UCTXIFG0) == 0)

    {

    }

    UCB1TXBUF = i2c\_reg; // Byte = register address

    while ((UCB1CTLW0 & UCTXSTT) != 0)

    {

    }

    if ((UCB1IFG & UCNACKIFG) != 0)

        return -1;

    UCB1CTLW0 &= ~UCTR;   // Master reads (R/W bit = Read)

    UCB1CTLW0 |= UCTXSTT; // Initiate a repeated Start Signal

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    //\*\*\*\*\*\*\*\*\*\* Read Frame #1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    while ((UCB1IFG & UCRXIFG0) == 0)

    {

    }

    byte1 = UCB1RXBUF;

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    //\*\*\*\*\*\*\*\*\*\* Read Frame #2 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    while ((UCB1CTLW0 & UCTXSTT) != 0)

    {

    }

    UCB1CTLW0 |= UCTXSTP; // Setup the Stop Signal

    while ((UCB1IFG & UCRXIFG0) == 0)

    {

    }

    byte2 = UCB1RXBUF;

    while ((UCB1CTLW0 & UCTXSTP) != 0)

    {

    }

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    // Merge the two received bytes

    \*data = ((byte1 << 8) | (byte2 & 0xFF));

    return 0;

}

// Write a word (2 bytes) to I2C (address, register)

int i2c\_write\_word(unsigned char i2c\_address, unsigned char i2c\_reg,

                   unsigned int data)

{

    unsigned char byte1, byte2;

    byte1 = (data >> 8) & 0xFF; // MSByte

    byte2 = data & 0xFF;        // LSByte

    UCB1I2CSA = i2c\_address;    // Set I2C address

    UCB1CTLW0 |= UCTR;          // Master writes (R/W bit = Write)

    UCB1CTLW0 |= UCTXSTT;       // Initiate the Start Signal

    while ((UCB1IFG & UCTXIFG0) == 0)

    {

    }

    UCB1TXBUF = i2c\_reg; // Byte = register address

    while ((UCB1CTLW0 & UCTXSTT) != 0)

    {

    }

    while ((UCB1IFG & UCTXIFG0) == 0)

    {

    }

    //\*\*\*\*\*\*\*\*\*\* Write Byte #1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    UCB1TXBUF = byte1;

    while ((UCB1IFG & UCTXIFG0) == 0)

    {

    }

    //\*\*\*\*\*\*\*\*\*\* Write Byte #2 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    UCB1TXBUF = byte2;

    while ((UCB1IFG & UCTXIFG0) == 0)

    {

    }

    UCB1CTLW0 |= UCTXSTP;

    while ((UCB1CTLW0 & UCTXSTP) != 0)

    {

    }

    return 0;

}

void Initialize\_I2C(void)

{

    // Configure the MCU in Master mode

    // Configure pins to I2C functionality

    // (UCB1SDA same as P4.0) (UCB1SCL same as P4.1)

    // (P4SEL1=11, P4SEL0=00) (P4DIR=xx)

    P4SEL1 |= (BIT1 | BIT0);

    P4SEL0 &= ~(BIT1 | BIT0);

    // Enter reset state and set all fields in this register to zero

    UCB1CTLW0 = UCSWRST;

    // Fields that should be nonzero are changed below

    // (Master Mode: UCMST) (I2C mode: UCMODE\_3) (Synchronous mode: UCSYNC)

    // (UCSSEL 1:ACLK, 2,3:SMCLK)

    UCB1CTLW0 |= UCMST | UCMODE\_3 | UCSYNC | UCSSEL\_3;

    // Clock frequency: SMCLK/8 = 1 MHz/8 = 125 KHz

    UCB1BRW = 8;

    // Chip Data Sheet p. 53 (Should be 400 KHz max)

    // Exit the reset mode at the end of the configuration

    UCB1CTLW0 &= ~UCSWRST;

}

// Configure UART to the popular configuration

// 9600 baud, 8-bit data, LSB first, no parity bits, 1 stop bit

// no flow control, oversampling reception

// Clock: SMCLK @ 1 MHz (1,000,000 Hz)

void Initialize\_UART(void)

{

    // Configure pins to UART functionality

    P3SEL1 &= ~(BIT4 | BIT5);

    P3SEL0 |= (BIT4 | BIT5);

    // Main configuration register

    UCA1CTLW0 = UCSWRST; // Engage reset; change all the fields to zero

    // Most fields in this register, when set to zero, correspond to the

    // popular configuration

    UCA1CTLW0 |= UCSSEL\_\_SMCLK; // Set clock to SMCLK

    // Configure the clock dividers and modulators (and enable oversampling)

    UCA1BRW = 6; // divider

    // Modulators: UCBRF = 8 = 1000 --> UCBRF3 (bit #3)

    // UCBRS = 0x20 = 0010 0000 = UCBRS5 (bit #5)

    UCA1MCTLW = UCBRF3 | UCBRS5 | UCOS16;

    // Exit the reset state

    UCA1CTLW0 &= ~UCSWRST;

}

// Configures ACLK to 32 KHz crystal

void config\_ACLK\_to\_32KHz\_crystal()

{

    // By default, ACLK runs on LFMODCLK at 5MHz/128 = 39 KHz

    // Reroute pins to LFXIN/LFXOUT functionality

    PJSEL1 &= ~BIT4;

    PJSEL0 |= BIT4;

    // Wait until the oscillator fault flags remain cleared

    CSCTL0 = CSKEY; // Unlock CS registers

    do

    {

        CSCTL5 &= ~LFXTOFFG; // Local fault flag

        SFRIFG1 &= ~OFIFG;   // Global fault flag

    } while ((CSCTL5 & LFXTOFFG) != 0);

    CSCTL0\_H = 0; // Lock CS registers

    return;

}

void uart\_write\_uint16(unsigned int number)

{

    // Max number of digits in a uint16\_t is 5

    int divisor = 10000;

    while (divisor > 0)

    {

        int digit = number / divisor;

        if (digit > 0 || divisor == 1)

        {

            uart\_write\_char('0' + digit);

        }

        number %= divisor;

        divisor /= 10;

    }

}

void uart\_write\_string(const char \*str)

{

    uint16\_t length = strlen(str);

    uint16\_t i;

    for (i = 0; i < length; i++)

    {

        uart\_write\_char(str[i]);

    }

}

void uart\_write\_nstring(void \*pStr, uint16\_t size)

{

    uint8\_t \*str = (uint8\_t \*)pStr;

    uint16\_t i = 0;

    for (i = 0; i < size; i++)

    {

        uart\_write\_char(str[i]);

    }

}

void uart\_write\_char(unsigned char ch)

{

    // Wait for any ongoing transmission to complete

    while ((FLAGS & TXFLAG) == 0)

    {

    }

    // Copy the byte to the transmit buffer

    TXBUFFER = ch; // Tx flag goes to 0 and Tx begins!

    return;

}

// The function returns the byte; if none received, returns null character

uint8\_t uart\_read\_char(void)

{

    uint8\_t temp;

    // Return null character (ASCII=0) if no byte was received

    if ((FLAGS & RXFLAG) == 0)

        return 0;

    // Otherwise, copy the received byte (this clears the flag) and return it

    temp = RXBUFFER;

    return temp;

}

#endif

void Initialize\_ADC()

{

    // Divert the pins to analog functionality

    // X-axis: A10/P9.2, for A10 (P9DIR=x, P9SEL1=1, P9SEL0=1)

    P9SEL1 |= BIT2;

    P9SEL0 |= BIT2;

    // Turn on the ADC module

    ADC12CTL0 |= ADC12ON;

    // Turn off ENC (Enable Conversion) bit while modifying the configuration

    ADC12CTL0 &= ~ADC12ENC;

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ADC12CTL0 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    // Set ADC12SHT0 (select the number of cycles that you determined)

    // RI = 10 kOhm

    // CI = 15 pF

    // RE = 10 kOhm

    // CE = 1 pF

    // t >= (20 kOhm) \* (16 pF) \* ln(2^13)

    // t >= (3.2 10^-8) \* ln(2^13)

    // t >= 0.2883 \* 10^-6

    // t >= 0.2883 us

    // t >= (approx) .32 us

    // MODCLK: [4 4.8 5.4] MHz

    // clock cycles: (12 + 1 for bits) + (? conversion time)

    ADC12CTL0 |= ADC12SHT0\_2;

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ADC12CTL1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    // Set ADC12SHS (select ADC12SC bit as the trigger)

    // Set ADC12SHP bit

    // Set ADC12DIV (select the divider you determined)

    // Set ADC12SSEL (select MODOSC)

    ADC12CTL1 |= ADC12SC | ADC12SHP | ADC12DIV\_0 | ADC12SSEL\_0;

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ADC12CTL2 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    // Set ADC12RES (select 12-bit resolution)

    // Set ADC12DF (select unsigned binary format)

    ADC12CTL2 |= ADC12RES\_\_12BIT;

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ADC12CTL3 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    // Leave all fields at default values

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ADC12MCTL0 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    // Set ADC12VRSEL (select VR+=AVCC, VR-=AVSS)

    // Set ADC12INCH (select channel A10)

    // Turn on ENC (Enable Conversion) bit at the end of the configuration

    ADC12CTL3 |= ADC12INCH\_10 | ADC12VRSEL\_0;

    ADC12CTL0 |= ADC12ENC;

    return;

}

void Initialize\_ADC2()

{

    // 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

    P8SEL0 |= BIT7;

    P9SEL1 |= BIT7;

    // Turn on the ADC module

    ADC12CTL0 |= ADC12ON;

    // Turn off ENC (Enable Conversion) bit while modifying the configuration

    ADC12CTL0 &= ~ADC12ENC;

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ADC12CTL0 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    // Set ADC12SHT0 (select the number of cycles that you determined)

    ADC12CTL0 |= ADC12SHT0\_3 | ADC12SHT1\_3 | ADC12MSC;

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ADC12CTL1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    // Set ADC12SHS (select ADC12SC bit as the trigger)

    // Set ADC12SHP bit

    // Set ADC12DIV (select the divider you determined)

    // Set ADC12SSEL (select MODOSC)

    ADC12CTL1 |= ADC12SHP | ADC12CONSEQ\_1;

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ADC12CTL2 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    // Set ADC12RES (select 12-bit resolution)

    // Set ADC12DF (select unsigned binary format)

    ADC12CTL2 |= ADC12RES\_2;

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ADC12CTL3 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    //

    ADC12CTL3 &= ~ADC12CSTARTADD\_31;

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ADC12MCTL1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    // Set ADC12VRSEL (select VR+=AVCC, VR-=AVSS)

    // Set ADC12INCH (select channel A10)

    // EOS stuff

    ADC12MCTL1 |= ADC12INCH\_4 | ADC12EOS;

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ADC12MCTL0 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    // Set ADC12VRSEL (select VR+=AVCC, VR-=AVSS)

    // Set ADC12INCH (select channel A10)

    ADC12MCTL0 |= ADC12INCH\_10;

    // Turn on ENC (Enable Conversion) bit at the end of the configuration

    ADC12CTL0 |= ADC12ENC;

    return;

}

int main()

{

    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer

    PM5CTL0 &= ~LOCKLPM5;     // Enable the GPIO pins

    Initialize\_UART();

    Initialize\_ADC2();

    uint16\_t lastX = 0;

    uint16\_t lastY = 0;

    for (;;)

    {

        ADC12CTL0 |= ADC12SC;

        while ((ADC12CTL1 & ADC12BUSY) == ADC12BUSY)

        {

        }

        uint16\_t x = ADC12MEM0;

        uint16\_t y = ADC12MEM1;

        if (lastX == 0 || lastY == 0)

        {

            lastX = x;

            lastY = y;

            continue;

        }

        if (MAX(lastX, x) - MIN(lastX, x) > 100 ||

            MAX(lastY, y) - MIN(lastY, y) > 100)

        {

            uart\_write\_string("<");

            uart\_write\_uint16(x);

            uart\_write\_string(", ");

            uart\_write\_uint16(y);

            uart\_write\_string(">\n\r");

            \_delay\_cycles(5e3);

            lastX = x;

            lastY = y;

        }

    }

    return 0;

}

# **3.0 Student Q&A**

The code first initializes the ADC module for the X and Y axis. By enabling ADC on PORT9 and PORT8, 12 bit mode, and so on as describe the in lab manual. Afterwards, the code initializes the UART (using SMCLK). Finally, the code enters an infinite loop where it starts ADC conversions and outputs the results to the UART terminal.

What are the values of the ADC’s RI and CI? If these values have a range show the range. Did you use the lower or upper range of these values? Justify your choice.

I used the upper bound of these values because Vcc was higher than 2-Volts and for also because a higher capacitor and/or resistance require longer time to charge and therefore these values would satisfy the worst case.

What is the minimum sample-and-hold time? Show how you computed this duration

    // RI = 10 kOhm

    // CI = 15 pF

    // RE = 10 kOhm

    // CE = 1 pF

    // t >= (20 kOhm) \* (16 pF) \* ln(2^13)

    // t >= (3.2 10^-8) \* ln(2^13)

    // t >= 0.2883 \* 10^-6

    // t >= 0.2883 us

    // t >= (approx) .32 us

    // MODCLK: [4 4.8 5.4] MHz

1. How many cycles does it take the ADC to convert a 12-bit result? (look in the configuration register that contains ADC12RES).

14 clock cycles.

2. In this experiment, we set our reference voltages VR+ = AV CC (Analog Vcc) and VR− = AV SS (Analog Vss). What voltage values do these signals have? Look in the MCU data sheet (slas789c) in Table 5.3. Assume that Vcc=3.3V and Vss=0

# **4.0** **Conclusion**