EEL 4742C: Embedded Systems

Christopher Badolato
11/25/2019
EEL4742-0011
Lab 10 ADC

Introduction

In this lab experiment we will be using the Analog to digital converter to take readings from the joystick located on the educational booster pack. The ADC converts an analog input signal to a binary number. First, we will read only from the x coordinated of the joystick. Then in part 2, we will configure the msp430 to ready both the x and y coordinates.

Part 10.1: Using the ADC SAR-Type

In this part of the experiment we learn to configure the Analog to Digital Converter (ADC) to convert the analog signal of the joystick to binary bits. (The specified number of bits is called the resolution.) The input signals will range between Vr- and Vr+
ADC produced result based on

Result = $N.\frac{Vin-Vr-}{Vr+ -Vr-}$

Where N is the full range value or 0 to 4000 in our case.
Moving the joystick to the right will print values closer to 4000.
Moving the joystick to the left will print values closer to 0.

We show below how we calculated the sample and hold time as well as the number of cycles needed to read the joystick.

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.

Ri = [1k to 10k ohm]
Ci = [10 to 15pF]

In experiment we used
Ri = 4k ohm
Ci = 15pF

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

Minimum sample and hold time is calculated using the equation below using the capacitance and resistance above with
Rs = 10k
Cs = 1pF
Ri = 4k
Ci = 15pF

$t >= (R_I + R_S) . (C_I + C_{pext}) . \ln(2_{n+2})$

t = .0000021737s or 2.17us
we wanted to be close to 3us

What divider of MODOSC did you use? How many cycles did you set the sample-and-hold time? Show your computation.

Using /1 for the MODOSC divider.
Our frequency varies from 4 to 5.4MHz we use the maximum frequency
.00000217s x 5.4MHz = 11.718 cycles
So we need 16 cycles

```
//Christopher Badolato
//11/25/2019
//Lab 10.1
//EEL 4742 0011
//ADC

#include <msp430.h>
#include <stdio.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 redLED BIT0 // Red LED at P1.0
#define greenLED BIT7 // Green LED at P9.7


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)
    ADC12CTL0|=ADC12ON|ADC12SHT1_10;
    //*************** ADC12CTL1 ***************
    // Set ADC12SHS (select ADC12SC bit as the trigger)
    // Set ADC12SHP bit
    // Set ADC12DIV (select the divider you determined)
    // Set ADC12SSEL (select MODOSC)
    ADC12CTL1= ADC12SHS_0|ADC12SHP|ADC12DIV_7|ADC12SSEL_0;
    //*************** ADC12CTL2 ***************
    // Set ADC12RES (select 12-bit resolution)
    // Set ADC12DF (select unsigned binary format)
```

```
    ADC12CTL2|= ADC12ENC;
    //*************** ADC12CTL3 ***************
    // Leave all fields at default values
    //*************** ADC12MCTL0 ***************
    // Set ADC12VRSEL (select VR+=AVCC, VR-=AVSS)
    // Set ADC12INCH (select channel A10)
    ADC12MCTL0|= ADC12INCH_10|ADC12VRSEL_0;
    // Turn on ENC (Enable Conversion) bit at the end of the configuration
    ADC12CTL0 |= ADC12ENC;
    return;
}


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;
}

// Configure UART to the popular configuration
// 9600 baud, 8-bit data, LSB first, no parity bits, 1 stop bit
// no flow control
// Initial clock: SMCLK @ 1.048 MHz with oversampling
void Initialize_UART(void){
        // Divert pins to UART functionality
    P3SEL1 &= ~(BIT4|BIT5);
    P3SEL0 |= (BIT4|BIT5);
        // Use SMCLK clock; leave other settings default
    UCA1CTLW0 |= UCSSEL_2;
        // Configure the clock dividers and modulators
        // UCBR=6, UCBRF=8, UCBRS=0x20, UCOS16=1 (oversampling)
    UCA1BRW = 6;
    UCA1MCTLW = UCBRS5|UCBRF3|UCOS16;
        // Exit the reset state (so transmission/reception can begin)
    UCA1CTLW0 &= ~UCSWRST;
}

void uart_write_char(unsigned char ch){
            // Wait for any ongoing transmission to complete
        while ((FLAGS & TXFLAG) == 0 ) {}
            // Write the byte to the transmit buffer
      TXBUFFER = ch;
}
```

```
    // The function returns the byte; if none received, returns NULL
unsigned char uart_read_char(void){
    unsigned char temp;
            // Return NULL if no byte received
    if( (FLAGS & RXFLAG) == 0)
        return NULL;
            // Otherwise, copy the received byte (clears the flag) and return it
    temp = RXBUFFER;
    return temp;
}

uart_write_uint16(unsigned int currentValue){
    int number;

    if(currentValue >= 10000){
        number = currentValue/10000;
        uart_write_char('0'+ number);
    }
    if(currentValue >= 1000){
        number = (currentValue/1000) % 10;
        uart_write_char('0'+ number);
    }
    if(currentValue >= 100){
        number = (currentValue/100) % 10;
        uart_write_char('0'+ number);
    }
    if(currentValue >= 10){
        number = (currentValue/10) % 10;
        uart_write_char('0'+ number);
    }
    number = currentValue % 10;
    uart_write_char('0' + number);
}

int main(void){
    WDTCTL = WDTPW | WDTHOLD; // Stop the Watchdog timer
    PM5CTL0 &= ~LOCKLPM5; // Enable the GPIO pins
    P1DIR |= redLED; // Direct pin as output
    P9DIR |= greenLED; // Direct pin as output
    P1OUT &= ~redLED; // Turn LED Off
    P9OUT &= ~greenLED; // Turn LED OFF
    int i;

    Initialize_UART();
    Initialize_ADC();


    while(1){
        P1OUT^=redLED;

        ADC12CTL0 |= ADC12SC;

        while ((ADC12CTL1 &ADC12BUSY)!=0);

        uart_write_uint16(ADC12MEM0);
```

```
        uart_write_char('\r');
        uart_write_char('\n');
        for(i = 0; i < (32768/2); i++){}

    }
}
```

Part 10.2: Reading the X- and Y- Coordinates of the Joystick

In this part of the experiment we will be reading the X and y coordinates from the joystick directly to a UART terminal on the CPU.

```
//Christopher Badolato
//11/25/2019
//Lab 10.2
//EEL 4742 0011
//ADC

#include <msp430.h>
#include <stdio.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 redLED BIT0 // Red LED at P1.0
#define greenLED BIT7 // Green LED at P9.7


void Initialize_ADC() {
//// Divert the vertical signal's pin to analog functionality
// X-axis: A10/P9.2, for A10 (P9DIR=x, P9SEL1=1, P9SEL0=1)
P9SEL1 |= BIT2;
P9SEL0 |= BIT2;
// Y- Axis: J3.26/ P8.7 FOR A4
P8SEL1 |=BIT7;
P8SEL0 |=BIT7;


// Turn on the ADC module
ADC12CTL0 |= ADC12ON;
// Turn off ENC (Enable Conversion) bit while modifying the configuration
ADC12CTL0 &= ~ADC12ENC;
//*************** ADC12CTL0 ***************
// Set the bit ADC12MSC (Multiple Sample and Conversion)

// Set ADC12SHT0 (select the number of cycles that you determined)
ADC12CTL0|=ADC12SHT0_2|ADC12MSC;
//*************** ADC12CTL1 ***************
// Set ADC12CONSEQ (select sequence-of-channels)
// Set ADC12SHS (select ADC12SC bit as the trigger)
// Set ADC12SHP bit
// Set ADC12DIV (select the divider you determined)
```

```
// Set ADC12SSEL (select MODOSC)
ADC12CTL1= ADC12SHS_0|ADC12SHP|ADC12DIV_7|ADC12SSEL_0;
ADC12CTL1|=ADC12CONSEQ_1;

//*************** ADC12CTL3 ***************
// Set ADC12CSTARTADD to 0 (first conversion in ADC12MEM0)
ADC12CTL3=ADC12CSTARTADD_0;
//*************** ADC12MCTL1 ***************
// Set ADC12VRSEL (select VR+=AVCC, VR-=AVSS)
// Set ADC12INCH (select the analog channel that you found)
// Set ADC12EOS (last conversion in ADC12MEM1)

//*************** ADC12CTL2 ***************
// Set ADC12RES (select 12-bit resolution)
// Set ADC12DF (select unsigned binary format)
ADC12CTL2|=ADC12RES_2;
//*************** ADC12CTL3 ***************
// Leave all fields at default values
//*************** ADC12MCTL0 ***************
// Set ADC12VRSEL (select VR+=AVCC, VR-=AVSS)
// Set ADC12INCH (select channel A10)
ADC12MCTL0|= ADC12INCH_10|ADC12VRSEL_0;
ADC12MCTL1|=ADC12INCH_4|ADC12VRSEL_0|ADC12EOS;
// Turn on ENC (Enable Conversion) bit at the end of the configuration
ADC12CTL0 |= ADC12ENC;
return;
}


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;
}

// Configure UART to the popular configuration
// 9600 baud, 8-bit data, LSB first, no parity bits, 1 stop bit
// no flow control
// Initial clock: SMCLK @ 1.048 MHz with oversampling
void Initialize_UART(void){
        // Divert pins to UART functionality
    P3SEL1 &= ~(BIT4|BIT5);
    P3SEL0 |= (BIT4|BIT5);
        // Use SMCLK clock; leave other settings default
```

```
    UCA1CTLW0 |= UCSSEL_2;
        // Configure the clock dividers and modulators
        // UCBR=6, UCBRF=8, UCBRS=0x20, UCOS16=1 (oversampling)
    UCA1BRW = 6;
    UCA1MCTLW = UCBRS5|UCBRF3|UCOS16;
        // Exit the reset state (so transmission/reception can begin)
    UCA1CTLW0 &= ~UCSWRST;
}

void uart_write_char(unsigned char ch){
            // Wait for any ongoing transmission to complete
        while ((FLAGS & TXFLAG) == 0 ) {}
            // Write the byte to the transmit buffer
      TXBUFFER = ch;
}

    // The function returns the byte; if none received, returns NULL
unsigned char uart_read_char(void){
    unsigned char temp;
            // Return NULL if no byte received
    if( (FLAGS & RXFLAG) == 0)
        return NULL;
            // Otherwise, copy the received byte (clears the flag) and return it
    temp = RXBUFFER;
    return temp;
}

uart_write_uint16(unsigned int currentValue){
    int number;

    if(currentValue >= 10000){
        number = currentValue/10000;
        uart_write_char('0'+ number);
    }
    if(currentValue >= 1000){
        number = (currentValue/1000) % 10;
        uart_write_char('0'+ number);
    }
    if(currentValue >= 100){
        number = (currentValue/100) % 10;
        uart_write_char('0'+ number);
    }
    if(currentValue >= 10){
        number = (currentValue/10) % 10;
        uart_write_char('0'+ number);
    }
    number = currentValue % 10;
    uart_write_char('0' + number);
}

int main(void){
    WDTCTL = WDTPW | WDTHOLD; // Stop the Watchdog timer
    PM5CTL0 &= ~LOCKLPM5; // Enable the GPIO pins
    P1DIR |= redLED; // Direct pin as output
    P9DIR |= greenLED; // Direct pin as output
```

```
    P1OUT &= ~redLED; // Turn LED Off
    P9OUT &= ~greenLED; // Turn LED OFF
    int i;

    Initialize_UART();
    Initialize_ADC();


    while(1){
        P1OUT^=redLED;

        ADC12CTL0 |= ADC12SC;

        while ((ADC12CTL1 &ADC12BUSY)!=0);

        uart_write_uint16(ADC12MEM0);
        uart_write_char(',');
        uart_write_uint16(ADC12MEM1);
        uart_write_char('\r');
        uart_write_char('\n');
        for(i = 0; i < (32768/2); i++){}
    }
}
```

Student Q&A

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

The 12-bit conversion in ADC12_B module always takes 14 cycles (not including sample and hold time)

2. The conversion time you found in the previous question does not include the sampleand-hold time. Find the total conversion time of your setup (sample-and-hold time and conversion time). Give the total cycles and the duration.

Rs = 10k
Cs = 1pF
Ri = 4k
Ci = 15pF
$t >= (R_I + R_S) . (C_I + C_{pext}) . \ln(2_{n+2})$

t = .0000021737s or 2.17us
.00000217s x 5.4MHz = 11.718 cycles

3. 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.

Vss has a nominal value of 0V
Vcc ranges from [1.8V to 3.6V]

4. It's possible for the ADC12 B module to use reference voltages that are generated by the module REF A (Reference A). What voltage levels does REF A provide? (look in the FR6xx Family User's Guide on p. 859 and p. 869).

Yes, it is possible using
1.2V, 2.0V, or 2.5V

Conclusion

To conclude in this lab, we configured our ADC analog digital converter to read input from the booster packs joystick and convert that signal into bits. We first find out result using just the x coordinate of the joystick. In the second lab we configure both of the joysticks to display with Uart.