Lab 8 Report
EEL4742C - 00446

Yousef Awad

October 2025

Contents

Introduction

This experiment introduces the functionality of the Analog-to-Digital Converter (ADC). The primary objective is to learn how to configure and use the ADC module to interface with an analog peripheral, specifically the 2D joystick on the Educational BoosterPack. The lab begins with an overview of the Successive Approximation Register (SAR) ADC's operation, including the importance of the sample-and-hold time (SHT). This theory is applied by calculating the minimum SHT required for the joystick and configuring the ADC12_B module's registers accordingly. Practical work involves programming the MCU to read the joystick's horizontal (X-axis) coordinate , and then expanding the configuration to read both the X- and Y-coordinates using a "sequence-of-channels". Finally, this knowledge is applied to develop a "Platform Balancing Control" application, which uses the joystick to adjust the height of four platform corners while monitoring for unsafe height differences.

8.1 Using the ADC SAR-Type

So, first we must calculate the SHT (Sample-and-Hold Time) with a 12-bit resolution ADC in the msp430fr6989. Now to do that, I had to look into the datasheet and found the following:

- $C_{internal} = 15pF$
- $R_{internal} = 10K\Omega$

And then with the given external resistance and capacitance of $10k\Omega$ and $1pF$, respectively, all plugged into the following function

$$t \geq (R_I + R_E) * (C_I + C_E) * ln(2^{n+1})$$

$$t \geq (10 * 10^3 + 10 * 10^3) * (15 * 10^{-12} + 1 * 10^{-12}) * ln(2^{12+1})$$

$$t \geq 2.8883 * 10^{-6}$$

Therefore, we get, when rounded up, $3\mu s$ to be the minimum Sample-and-Hold Time. Now, to find the SHT duration we simply take the highest value of the frequency, 5.4MHz and find the time per cycle, or $\frac{1}{5.4MHz}$ and use it as the divisor to the previous $3\mu s$ from before. With that, I got, $\approx$ 11 SHT Cycles, which would round up to 16 due to it needing to be a power of 2. As well... Thankfully, the returned values are valid looking and make sense!

```
#include <msp430fr6989.h>
#include <stdint.h>

#define redLED BIT0
#define FLAGS UCA1IFG
#define RXFLAG UCRXIFG
#define TXFLAG UCTXIFG
#define TXBUFFER UCA1TXBUF
```

```c
#define RXBUFFER UCA1RXBUF

// 9600 baud based on 1 MHz SMCLK w/16x oversampling.
// 8 bits, no parity, LSB first, 1 stop bit UART communication
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_2; // 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;
}

void Initialize_ADC(void)
{
  // Configure 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 ***************
  // ADC12SHT0x sets SHT cycles for results 0-7, 24-31
  // ADC12MSC sets multiple analog inputs
  ADC12CTL0 |=
      ADC12SHT0_2; // Sets SHT of 16 cycles (found in doc. slau367o table 34.4)
  //*************** ADC12CTL1 ***************
  // ADC12SHS sets read trigger
  // ADC12SHP sets SAMPCON use
  /// ADC12DIV sets clock divider
  // ADC12SSEL sets clock base
  // ADC12CONSEQx sets conversion sequence mode
  ADC12CTL1 |= ADC12SHS_0;  // 0 = ADC12SC bit
  ADC12CTL1 |= ADC12SHP;    // 1 = SAMPCON sourced from clock
  ADC12CTL1 |= ADC12DIV_0;  // 0 = /1
  ADC12CTL1 |= ADC12SSEL_0; // 0 = MODOSC
  // ADC12CTL1 |= ADC12CONSEQ_1;
  // values in doc. slau367o table 34.5
  //*************** ADC12CTL2 ***************
```

```c
  // ADC12RES sets bit resolution
  // ADC12DF sets data format
  ADC12CTL2 |= ADC12RES_2; // 2 = 12-bit
  ADC12CTL2 &= ~ADC12DF;   // 0 = unsigned binary
  //*************** ADC12MCTL0 ***************
  // ADC12VRSELx sets VR+ and VR- sources as well as buffering
  // ADC12INCHx sets analog input
  ADC12MCTL0 |= ADC12VRSEL_0; // 0 -> VR+ = AVCC and VR- = AVSS
  ADC12MCTL0 |= ADC12INCH_10; // 10 = A10 input
  //*************** ADC12MCTL1 ***************
  // set ENC bit at end of config
  ADC12CTL0 |= ADC12ENC;
}

void uart_write_char(unsigned char ch)
{
  while ((FLAGS & TXFLAG) == 0)
  {
    // Wait for any ongoing transmission to complete
  }
  // Copy the byte to the transmit buffer
  TXBUFFER = ch; // Tx flag goes to 0 and Tx begins!
  return;
}

void uart_write_12bit(uint16_t n)
{
  const char hex_digits[] = "0123456789ABCDEF"; // Digits used in hexadecimal
  uint8_t digit;                                // one hex digit = 4 bits
  int i;
  // print the 0x part of hex format
  uart_write_char('0');
  uart_write_char('x');
  // Extract and print hex digits from input
  // i = 8 because bits 12-15 will always be 0000
  for (i = 8; i >= 0; i = i - 4)
  {
    digit = (n >> i) & 0xF;
    uart_write_char(hex_digits[digit]);
  }
}

void main(void)
{
  WDTCTL = WDTPW | WDTHOLD;
  PM5CTL0 &= ~LOCKLPM5;

  P1DIR |= redLED;
  P1OUT |= redLED;

  Initialize_UART();
  Initialize_ADC();

  for (;;)
  {
```

```
    ADC12CTL0 |= ADC12SC; // Triggers ADC12BUSY while reading input
    while ((ADC12CTL1 & ADC12BUSY) != 0)
    {
      // Wait for flag to drop
    }
    ADC12CTL0 &= ~ADC12SC;
    uint16_t x_coord = ADC12MEM0; // ADC12MEM0 linked to A10, x-input
    uart_write_12bit(x_coord);    // Print x-coordinate to console
    uart_write_char('\n');        // newline
    P1OUT ^= redLED;              // toggle red LED
    _delay_cycles(500000);        // 0.5 second delay
  }
}
```

8.2 Reading the X- and Y- Coordinates of the Joystick

```c
#include <msp430fr6989.h>
#include <stdint.h>

#define redLED BIT0
#define FLAGS UCA1IFG
#define RXFLAG UCRXIFG
#define TXFLAG UCTXIFG
#define TXBUFFER UCA1TXBUF
#define RXBUFFER UCA1RXBUF

// 9600 baud based on 1 MHz SMCLK w/16x oversampling.
// 8 bits, no parity, LSB first, 1 stop bit UART communication
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_2; // 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;
}

void Initialize_ADC(void)
{
  // Configure 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 (P8DIR=x, P8SEL1=1,P8SEL0=)
  P8SEL1 |= BIT7;
  P8SEL0 |= BIT7;
  // Turn on the ADC module
  ADC12CTL0 |= ADC12ON;
  // Turn off ENC (Enable Conversion) bit while modifying the configuration
  ADC12CTL0 &= ~ADC12ENC;
  //*************** ADC12CTL0 ***************
  // ADC12SHT0x sets SHT cycles for results 0-7, 24-31
  // ADC12MSC sets multiple analog inputs
  ADC12CTL0 |= ADC12SHT0_2; // Sets SHT of 16 cycles (found in doc.
```

```
    slau367o table 34.4)
  ADC12CTL0 |= ADC12MSC; // 1= multiple inputs
  //*************** ADC12CTL1 ***************
  // ADC12SHS sets read trigger
  // ADC12SHP sets SAMPCON use
  /// ADC12DIV sets clock divider
  // ADC12SSEL sets clock base
  // ADC12CONSEQx sets conversion sequence mode
  ADC12CTL1 |= ADC12SHS_0;  // 0 = ADC12SC bit
  ADC12CTL1 |= ADC12SHP;    // 1 = SAMPCON sourced from clock
  ADC12CTL1 |= ADC12DIV_0;  // 0 = /1
  ADC12CTL1 |= ADC12SSEL_0; // 0 = MODOSC
  ADC12CTL1 |= ADC12CONSEQ_1;
  // values in doc. slau367o table 34.5
  //*************** ADC12CTL2 ***************
  // ADC12RES sets bit resolution
  // ADC12DF sets data format
  ADC12CTL2 |= ADC12RES_2; // 2 = 12-bit
  ADC12CTL2 &= ~ADC12DF;   // 0 = unsigned binary
  //*************** ADC12CTL3 ***************
  // ADC12CSTARTADDx sets first ADC12MEM register in conversion sequence
  ADC12CTL3 |= ADC12CSTARTADD_0;
  //*************** ADC12MCTL0 ***************
  // ADC12VRSELx sets VR+ and VR- sources as well as buffering
  // ADC12INCHx sets analog input
  ADC12MCTL0 |= ADC12VRSEL_0; // 0 -> VR+ = AVCC and VR- = AVSS
  ADC12MCTL0 |= ADC12INCH_10; // 10 = A10 input
  //*************** ADC12MCTL1 ***************
  // ADC12ENC sets final conversion channel
  ADC12MCTL1 |= ADC12VRSEL_0;
  ADC12MCTL1 |= ADC12INCH_4;
  ADC12MCTL1 |= ADC12EOS; // 1 = last converted input
  // set ENC bit at end of config
  ADC12CTL0 |= ADC12ENC;
}

void uart_write_char(unsigned char ch)
{
  while ((FLAGS & TXFLAG) == 0)
  {
    // Wait for any ongoing transmission to complete
  }
  // Copy the byte to the transmit buffer
  TXBUFFER = ch; // Tx flag goes to 0 and Tx begins!
  return;
}

void uart_write_12bit(uint16_t n)
{
  const char hex_digits[] = "0123456789ABCDEF"; // Digits used in hexadecimal
  uint8_t digit;                                // one hex digit = 4 bits
  int i;
  // print the 0x part of hex format
  uart_write_char('0');
```

```c
  uart_write_char('x');
  // Extract and print hex digits from input
  // i = 8 because bits 12-15 will always be 0000
  for (i = 8; i >= 0; i = i - 4)
  {
    digit = (n >> i) & 0xF;
    uart_write_char(hex_digits[digit]);
  }
}

void main(void)
{
  WDTCTL = WDTPW | WDTHOLD;
  PM5CTL0 &= ~LOCKLPM5;

  P1DIR |= redLED;
  P1OUT |= redLED;

  Initialize_UART();
  Initialize_ADC();

  for (;;)
  {
    ADC12CTL0 |= ADC12SC; // Triggers ADC12BUSY while reading input
    while ((ADC12CTL1 & ADC12BUSY) != 0)
    {
      // Wait for flag to drop
    }
    ADC12CTL0 &= ~ADC12SC;
    uint16_t x_coord = ADC12MEM0; // ADC12MEM0 linked to A10, x-input
    uint16_t y_coord = ADC12MEM1; // ADC12MEM1 linked to A4, y-input
    uart_write_12bit(x_coord);    // Print x-coordinate to console
    uart_write_char(' ');         // Readability space
    uart_write_12bit(y_coord);    // Print y-coordinate to console
    uart_write_char('\n');        // newline
    P1OUT ^= redLED;              // toggle red LED
    _delay_cycles(500000);        // 0.5 second delay
  }
}
```

Student Q&A

1

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

- It would take 14 clock cycles.

2

Given: *In this experiment, we set our reference voltages* $VR+ = AVCC$ *(Analog Vcc) and* $VR- = AVSS$ *(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.* It is the following:

- $AVCC = V_{cc} = 3.3V$
- $AVSS = V_{ss} = 0.0V$

Conclusion

This lab successfully demonstrated the configuration and application of the ADC12_B module for interfacing with the analog joystick. A foundational part of the lab involved calculating the minimum sample-and-hold time (SHT) based on SAR ADC theory and configuring the necessary control registers, such as 'ADC12CTL0' and 'ADC12CTL1', for the correct clock, resolution, and SHT cycles. The experiment progressed from reading a single analog channel for the X-axis to implementing the ADC's "sequence-of-channels" mode to capture both X and Y coordinates from one trigger. This cumulative knowledge was then applied to build a "Platform Balancing Control" application, which used the joystick input to manage a system and enforce safety constraints, providing comprehensive practical experience in managing and utilizing analog-to-digital conversions.