1. CPE 325: Laboratory Assignment #10
Analog – Digital Converter

Objectives: This tutorial will introduce the configuration and operation of the MSP430 12-bit analog to digital converter (ADC). In this lab you will use the on-board temperature sensor, and you will also learn how to configure the ADC12 for other external analog inputs.

- Configuration of the ADC12 peripheral
- Choosing reference voltages to maximize signal resolution
- Operation of the ADC12 peripheral
- Implementation of the on-board temperature sensor
- Configuration for other analog signal inputs

Note: this lab requires all the previous tutorials, especially tutorials on UART communication, timer A, and getting started with MSP430F IAR Embedded Workbench. Further, reading the user guide for TI Experimenter's Board is useful.

1.1. Analog to Digital Converters

Often times while creating or implementing systems one has the need to deal with analog signals. Though modern computing relies on binary states, the physical world operates through a variety of signals – chemical, physical, electromagnetic, and more. Sensors and transducers can often translate those phenomena into an analog voltage signal. Analog to digital converters allow us to interface our microcontroller with a variety of devices in order to store, analyze, and modify signals.

The MSP430 series have a variety of different analog to digital converters with varying features and conversion methods. In this lab we focus on the ADC12 converter. The ADC12 converter for the MSP430FG4618 has 16 configurable input channels that can each be routed to signal input pins, voltage signals, or an internal temperature sensor. The ADC12 has a configurable sampling timer that can be used to sample a configurable

1.1.1. ADC Resolution, Reference Voltages, and Signal Resolution

There are several key factors that should be regarded when configuring your ADC to most effectively read the analog signal. The first parameter you should understand is the device's vertical resolution factor. We will be using the 12-bit ADC12 peripheral that has a vertical resolution of 12 bits. That means that it can distinguish between 2^{12} (0 to 4095) input voltage levels. An A/D converter described as "n-bit" can distinguish between 0 and 2^n-1 voltage steps.

After acknowledging your ADC vertical resolution, the reference voltages need to be set. Setting the reference voltages dials in the minimum and maximum values read by the ADC. For instance, you could set your V_- to -5V and your V_+ to 10V. With that setup on the ADC12, the numerical sampled value 0 would correspond to a signal input of -5V, and a sampled value of 4095 would correspond to a 10V input.

It is very important to characterize the input signal you're expecting before you set up your ADC. If you expect a signal input between 0V and 3V, you should set your reference voltages to 0V

and 3V. If you set them to -5V and +5V, you would be wasting a large amount of your sample "bit depth," and your overall sample resolution would suffer because your sample input values would stay between 2048 and 3275. There would only be (3275-2048=1227) steps of resolution for your input signal rather than 4096 if you chose 0V and 3V as your reference voltages. Similarly, the ADC relies on a timer to periodically sample the incoming signals. You should choose a timer that doesn't sample too quickly or slowly and that can capture all important waveform features.

1.2. On-chip Temperature Sensor

The ADC12 has an internal temperature sensor that creates an analog voltage proportional to its temperature. The device can be configured and enabled using configuration registers on the ADC12 peripheral. The performance curve from the datasheet can be seen in figure 1.

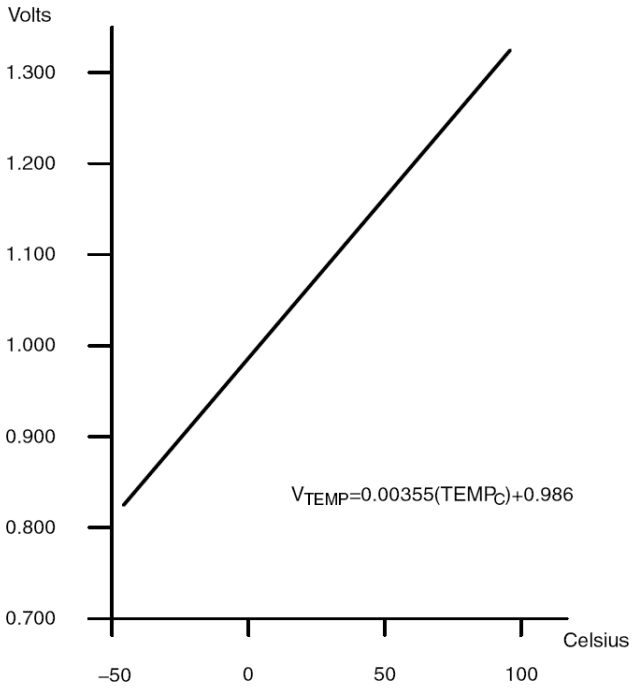


Figure 1. Internal temperature sensor voltage curve

Let us consider a C application shown in Figure 2 that samples the on-chip temperature sensor, converts the sampled voltage from the sensor to temperature in degrees Celsius and Fahrenheit, and sends the temperature information through a RS232 link to the Hyper-terminal application.

Analyze the program and test it on the TI Experimenter's Board. Answer the following questions.

What does the program do?
How the ADC12 module is configured?
How the USART0 module is configured?

```
1. /*****************************************************************************
2. //  TI Experimenter's Board Demo - Temperature Display in Hyperterminal
3. //
4. //  Filename: Temp_Display.c
5. //
6. //  Description:
7. //          This program samples the on-chip temperature sensor, converts the

8. //        sampled voltage from the sensor to temperature in degrees Celsius and
9. //        Fahrenheit, and sends the temperature sensor through a RS232 link
10.//          to the Hyper-terminal application.
11.//  Instructions: Set the following parameters in hyperterminal
12.//                                  Port :       COM1
13.//                                  Baud rate :  38400
14.//                                  Data bits:   8
15.//                                  Parity:             None
16.//                                  Stop bits:   1
17.//                                  Flow Control:None
18.//
19.//  Authors: A. Milenkovic, milenkovic@computer.org
20.//               Max Avula (ma0004@uah.edu)
21.//             UAH
22.//  March 2012
23.//  Built with IAR Embedded Workbench IDE Version: 6.3.11.2079
24.*****************************************************************************/
25.
26.#include  <msp430xG46x.h>
27.#include  <stdio.h>
28.
29.unsigned char thr_char;  /* hold char from UART RX*/
30.unsigned char rx_flag;   /* receiver rx status flag */
31.
32.char gm1[] = "Hello! I am an MSP430. Would you like to know my temperature? (Y|N)";
33.char gm2[] = "Bye, bye!";
34.char gm3[] = "Type in Y or N!";
35.
36.long int temp;
37.long int IntDegF;
38.long int IntDegC;
39.
40.char NewKey[25];
41.
42.
43.// UART Initializaion
44.void UART_Initialize(void);
45.//send char function
46.void UART_putchar(char c);
47.
48.void main(void)
49.{
50.  int i = 0;
51.
52.  WDTCTL = WDTPW + WDTHOLD;   // Stop watchdog timer
```

```
53.
54.  UART_Initialize();
55.
56.  ADC12CTL0 = SHT0_8 + REFON + ADC12ON;
57.  ADC12CTL1 = SHP;                      // enable sample timer
58.  ADC12MCTL0 = 0x01A;
59.  ADC12IE = 0x001;
60.
61.  rx_flag = 0;            // rx default state "empty"
62.  _EINT();                // enable global interrupts
63.
64.
65.Wait:
66.    //send a greeting message
67.    for(i = 0; i < 67; i++)
68.    {
69.           thr_char = gm1[i];
70.           UART_putchar(thr_char);
71.    }
72.     while(!(rx_flag&0x01)); // wait until receive the character from HyperTerminal
73.     rx_flag = 0;            // clear rx_flag
74.     UART_putchar(thr_char);
75.     UART_putchar('\n');    // newline
76.     UART_putchar('\r');    // carriage return
77.
78.    // character input validation
79.   if ((thr_char != 'y') && (thr_char != 'n') && (thr_char !='Y') && (thr_char !
   ='N'))
80.    {
81.           for(i = 0; i < 15; i++)
82.           {
83.                  thr_char = gm3[i];
84.                  UART_putchar(thr_char);
85.           }
86.           UART_putchar('\n');    // newline
87.           UART_putchar('\r');    // carriage return
88.           goto Wait;
89.           }
90.    if ((thr_char == 'y') || (thr_char == 'Y'))
91.    {
92.        ADC12CTL0 |= ENC + ADC12SC;         // Sampling and conversion start
93.       _BIS_SR(CPUOFF + GIE);              // LPM0 with interrupts enabled
94.
95.       //  oF = ((x/4096)*1500mV)-923mV)*1/1.97mV = x*761/4096 - 468
96.       //  IntDegF = (ADC12MEM0 - 2519)* 761/4096
97.
98.       IntDegF = (temp - 2519) * 761;
99.       IntDegF = IntDegF / 4096;
100.
101.         //  oC = ((x/4096)*1500mV)-986mV)*1/3.55mV = x*423/4096 - 278
102.         //  IntDegC = (ADC12MEM0 - 2692)* 423/4096
103.
104.         IntDegC = (temp - 2692) * 423;
```

```
105.         IntDegC = IntDegC / 4096;
106.
107.         //printing the temperature on hyperterminal
108.            sprintf(NewKey, "T(F)=%ld\tT(C)=%ld\n", IntDegF, IntDegC);
109.            for(i = 0; i < 25; i++) {
110.            thr_char = NewKey[i];
111.            UART_putchar(thr_char);
112.         }
113.         UART_putchar('\n');    // newline
114.         UART_putchar('\r');    // carriage return
115.         goto Wait;
116.      }
117.
118.       if ((thr_char == 'n') || (thr_char == 'N'))
119.          {
120.                 for(i = 0; i < 9; i++)
121.                 {
122.                        thr_char = gm2[i];
123.                        UART_putchar(thr_char);
124.                 }
125.                 UART_putchar('\n');    // newline
126.                 UART_putchar('\r');    // carriage return
127.      }
128.   } //end main
129.
130.   void UART_Initialize(void)
131.   {
132.
133.     P2SEL |= BIT4+BIT5;      // Set UC0TXD and UC0RXD to transmit and receive data
134.     UCA0CTL1 |= BIT0;        // software reset
135.     UCA0CTL0 = 0;                   // USCI_A0 control register
136.     UCA0CTL1 |= UCSSEL_2; // clock source SMCLK
137.     UCA0BR0=27;              // 1 MHz 38400
138.     UCA0BR1=0;               // 1 MHz 38400
139.     UCA0MCTL=0x94;           // Modulation
140.     UCA0CTL1 &= ~BIT0;       // software reset
141.     IE2 |=UCA0RXIE;       // Enable USCI_A0 RX interrupt
142.   }
143.   void UART_putchar(char c)
144.   {
145.     // wait for other character to transmit
146.     while (!(IFG2 & UCA0TXIFG));
147.     UCA0TXBUF = c;
148.   }
149.
150.   #pragma vector=USCIAB0RX_VECTOR
151.   __interrupt void USCIA0RX_ISR (void)
152.   {
153.     thr_char = UCA0RXBUF;
154.     rx_flag=0x01;          // signal main function receiving a char
155.     LPM0_EXIT;
156.   }
157.
```

```
158.
159.   #pragma vector=ADC12_VECTOR
160.   __interrupt void ADC12ISR (void)
161.   {
162.       temp = ADC12MEM0;                   // Move results, IFG is cleared
163.       _BIC_SR_IRQ(CPUOFF);                // Clear CPUOFF bit from 0(SR)
164.   }
```

Figure 2. C program that samples the on-board temperature sensor and send the data to HyperTerminal (Lab10_D1.c)

1.3. Example: Analog Thumbstick Configuration

The above program details configuration and use of the ADC12 for single channel use. However, many analog devices or systems would require multiple channel configuration. As an example, let's imagine an analog joystick as is used by controllers for most modern gaming consoles. So-called thumbsticks have X and Y axis voltage outputs depending on the vector of the push it receives as input. For this example, we'll use a thumbstick that has 0 to 3V output in the X and Y axes. No push on either axis results in a 1.5V output for both axes. In figure 3 below, note how a push at about 120° with around 80% power results in around 2.75V output for the Y axis and 0.8V output for the X axis.

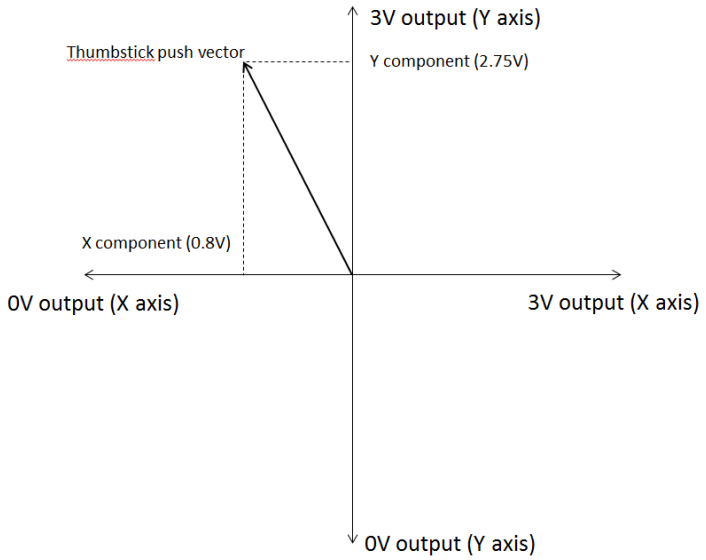
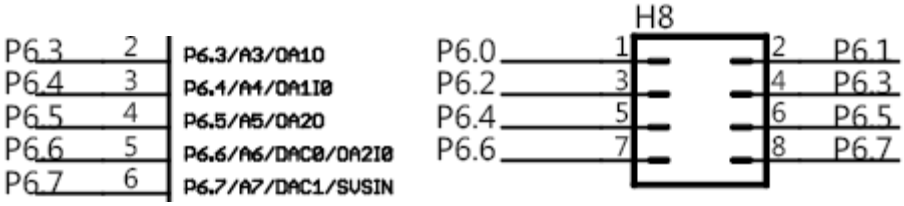


Figure 3: Performance data for hypothetical thumbstick

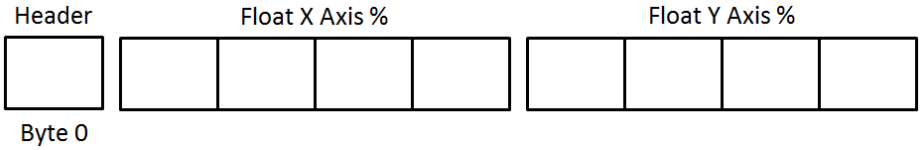
We want to test the thumbstick output using the UAH Serial App. To do this, we'll first hook the thumbstick outputs to our device. Let's say we'll use analog input A3 for the X axis and A7 for the Y axis.



Note that the A3 pin corresponds to pin 4 of the H8 header, and A7 corresponds to pin 8. That is where we'll connect our device. Because the outputs are from 0 to 3V, we need to set our reference voltages accordingly. We can use the board's ground and 3V supply as references. We'll want to have our output as Float datatypes. The output for each axis should be a percentage. In figure 3, for example, the converted Y axis output would be 91.67% and the X axis output would be 26.67%. Here's the formula you would use to convert the values (remember, the microcontroller is going to be receiving values from 0 to 4095 based on voltage values from 0V to 3V that we set as our references):

$$Input\ ADC\ Value \in steps \times \frac{3\,V}{4096\ steps} \times \frac{100}{3\,V} = \%Power$$

We could send our information in a variety of ways including a vector format, signed percentage, or even just ADC "steps." If we're using our unsigned percentage, our packet to send to the UAH serial app would like similar to the one below:



As discussed in a previous lab, a character pointer to non-character datatypes can be extremely helpful in sending byte-size chunks of information.
Below, figure 4 shows demo code that could be used to set up the ADC12 and UART and send the thumbstick information to the UAH serial app.

```
1  ////////////////////////////////////////////////////////////////////////////
2  //Program: Lab 10 demo 2
3  //Description: This code interfaces with an imaginary thumbstick that has
4  //    x and y axis output from 0 to 3 v.  The data is sent as a percentage
5  //    value to the UAH serial app
6  //By: Micah Harvey
7  //Date: July 10, 2013
8  //Baud rate: 38400
9  ////////////////////////////////////////////////////////////////////////////
10
11 #include <msp430xG46x.h>
12
13 void turn_off();
14 void sendData();
15 void UART_putchar(char);
16 void initializeUART();
17
18 volatile long int ADCXval, ADCYval;
19 volatile float Xper, Yper;
20
21 void main(void)
22 {
23   WDTCTL = WDTPW +WDTHOLD;              // Stop WDT
24
25   initializeUART();                    // Setup UART for RS-232
26
```

```
27   P6DIR &= ~0x88;       // Configure P6.3 and P6.8 as input pins
28   P6SEL |= 0x88; // Configure P6.3 and P6.8 as analog pins
29
30   //Set up timer to send ADC info to PC overy 100 ms
31   TACCR0 = 3277;                //3277 / 32768 Hz = 0.1s
32   TACTL = TASSEL_1 + MC_1;      //ACLK, up mode
33   TACCTL0 = CCIE;               //enabled interupt
34
35   //Set up ADC 12
36   ADC12CTL0 = ADC12ON + SHT0_6 + MSC; // configure ADC converter
37   ADC12CTL1 = SHP + CONSEQ_1;     // Use sample timer, single sequence
38   ADC12MCTL0 = INCH_3;            // ADC chan 0 is A1 pin - Stick X-axis
39   ADC12MCTL1 = INCH_7 + EOS;      // ADC chan 1 is A2 pin - Stick Y-axis
40                           //EOS - End of Sequence for Conversions
41   ADC12IE |= 0x80;                             // Enable ADC12IFG.8
42   for (int i = 0; i < 0x3600; i++);         // Delay for reference start-up
43   ADC12CTL0 |= ENC;                            // Enable conversions
44
45   _EINT();
46
47   while (1)
48   {
49     ADC12CTL0 |= ADC12SC;                   // Start conversions
50     __bis_SR_register(LPM0_bits + GIE);        // enter LPM0
51   }
52
53 }
54
55 ////////////////////////////////////////////////////////////////////////////
56 // User-defined function definitions
57 ////////////////////////////////////////////////////////////////////////////
58
59 void UART_putchar(char c)
60 {
61     while(!(IFG2 & UCA0TXIFG)); //Wait for previous character to be sent
62     UCA0TXBUF = c;              //Send byte to the buffer for transmitting
63 }
64
65 //Initializes UART control registers for transmitting data, as well as baud rate
66 void initializeUART(void)
67 {
68
69   P2SEL |= BIT4 + BIT5;          //Set up Rx and Tx bits
70   UCA0CTL0 = 0;                  //Set up default RS-232 protocol
71   UCA0CTL1 |= BIT0 + UCSSEL_2;   //Disable device, set clock
72   UCA0BR0 = 27;                  //1048576 Hz / 38400 = 27
73   UCA0BR1 = 0;
74   UCA0MCTL = 0x94;
75   UCA0CTL1 &= ~BIT0;             //Start UART device
76 }
77
78 void sendData(void)
79 {
```

```
80    Xper = (ADCXval*3/4095*100/3);        //calculate percentage outputs
81    Yper = (ADCYval*3/4095*100/3);
82
83    int i;
84
85    //Send packet via rs-232
86    UART_putchar(0x55);                   //send header
87
88    //Use character pointers to send one byte of float X and Y value at a time
89    char *xpointer=(char *)&Xper;
90    char *ypointer=(char *)&Yper;
91
92    //Send x percentage float one byte at a time
93    for(i=0; i<4; i++)
94    {
95      UART_putchar(xpointer[i]);
96    }
97
98    //Send y percentage float one byte at a time
99    for(i=0; i<4; i++)
100   {
101     UART_putchar(ypointer[i]);
102   }
103 }
104
105 ////////////////////////////////////////////////////////////////////////////
106 // Interrupt service routines
107 ////////////////////////////////////////////////////////////////////////////
108
109 #pragma vector=ADC12_VECTOR
110 __interrupt void ADC12ISR(void)
111 {
112   ADCXval = ADC12MEM0;                    // Move results, IFG is cleared
113   ADCYval = ADC12MEM1;
114   __bic_SR_register_on_exit(LPM0_bits);     // Exit LPM0
115 }
116
117 #pragma vector = TIMERA0_VECTOR
118 __interrupt void timerA_isr()
119 {
120   sendData();
121   __bic_SR_register_on_exit(LPM0_bits);     // Exit LPM0
122 }
```

1.4. References

In order to understand more about the ADC12 peripheral and its configuration, please reference the following materials:

- Davies Text, pages 407-438
- MSP430FG4618 User's Guide, Chapter 28, pages 787-814 (ADC12)

- MSP430FG4618 User's Guide, page 802 (Internal temperature sensor)

1.5. Assignment

Write a C program that will interface a 3-dimensional accelerometer ADXL335. Your program should sample x, y, and z axis 20 times per second, calculate acceleration in units of g (gravity of Earth), and send samples to a workstation running the UAH Serial App. The sample should start with the 0x55 header byte, then be followed by 3 32-bit float values for x, y, and z (in that order). You will need to access the datasheet for the ADXL335 accelerometer in order to properly connect it to your board and configure your ADC12. The datasheet is available on ANGEL.

Keep in mind that A2 on the board schematic DOES NOT WORK.