CPE 325: Embedded Systems Laboratory
Laboratory #11 Tutorial
Analog-to-Digital Converter and Digital-to-Analog Converter

Aleksandar Milenković
Email: milenka@uah.edu
Web: http://www.ece.uah.edu/~milenka

Objective

This tutorial will introduce the configuration and operation of the MSP430 12-bit analog-to-digital converter (ADC12) and digital-to-analog converter (DAC12). Programs will demonstrate the use of ADC12 to interface an on-board temperature sensor as well as external analog inputs. In addition, a program will demonstrate how to generate an analog periodic signal of desired waveform. Specifically, you will learn how to:

Configure the ADC12 and DAC12 peripherals
Choose reference voltages to maximize signal resolution
Create waveform lookup table in MATLAB
Interface of an on-board temperature sensor
Interface external analog signal inputs

Notes

All previous tutorials are required for successful completion of this lab, especially the tutorials introducing the TI Experimenter's board, UART communication, and Timer_A.

Contents

1 Analog-to-Digital Converters 2
1.1 ADC Resolution, Reference Voltages, and Signal Resolution 2
1.2 On-Chip Temperature Sensor 2
1.3 Example: Analog Thumbstick Configuration 6
3 Digital-to-Analog Converters 12
3.1 Sinusoidal Wave Generator 13
4 References 15

1 Analog-to-Digital Converters

The world around us is analog. Sensors or transducers convert physical quantities such as, temperature, force, light, sound, and others, into electrical signals, typically voltage signals that we can measure. Analog-to-digital converters allow us to interface these analog signals and convert them into digital values that can further be stored, analyzed, or communicated.

The MSP430 family of microcontrollers has a variety of analog-to-digital converters with varying features and conversion methods. In this laboratory we focus on the ADC12 converter used in the MSP430FG4618. The ADC12 converter has 16 configurable input channels; 8 input channels are routed to corresponding analog input pins; remaining input channels are routed to internal voltages an on-chip temperature sensor.

1.1 ADC Resolution, Reference Voltages, and Signal Resolution

There are several key factors that should be regarded when configuring your ADC12 to most effectively read the analog signal. The first parameter you should understand is the device's voltage resolution, i.e., the smallest change of an input analog signal that causes a change in the digital output. We will be using the 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 10 V. With that setup on the ADC12, the numerical sampled value 0 would correspond to a signal input of -5 V, and a sampled value of 4095 would correspond to a 10 V input.

It is very important to characterize the input signal you are expecting before you set up your ADC. If you expect a signal input between 0 V and 3 V, you should set your reference voltages to 0 V and 3 V. 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 4095 if you choose 0 V and 3 V as your reference voltages.

An ADC typically relies on a timer to periodically generate a trigger to start sampling of the incoming signals. You should choose a timer period that triggers sampling frequently enough to recreate the original input signals (the minimum sampling frequency should be at least two times the frequency of the signal's largest harmonic).

1.2 On-Chip Temperature Sensor

The MSP430's ADC12 has an internal temperature sensor that creates an analog voltage proportional to its temperature. A transfer characteristic of the temperature sensor is shown in Figure 1. The output of the temperature sensor is connected to the input multiplexor channel 10 (INCHx=1010). When using the temperature sensor, the sample time (the time ADC12 is looking at the analog signal) must be greater than 30 μs. From the transfer characteristic, we get that the

temperature in degrees Celsius can be expressed as $degC = \frac{V_{sensor} - 986\ mV}{3.55\ mV}$, where V_{sensor} is the voltage from the temperature sensor. The ADC12 transfer characteristic gives the following equation: $ADCResult = 4095 \cdot \frac{V_{sensor}}{V_{REF}}$, or $V_{sensor} = V_{REF} \cdot \frac{ADCResult}{4095}$. By using the internal voltage generator V_{REF}=1,500 mV (1.5 V), we can derive temperature as follows: $degC = \frac{(ADCResult - 2692) \cdot 423}{4095}$. Make sure your calculations match the equation given. How would equation change if instead of using V_{REF}=1.5 V we use V_{REF}=2.5 V?

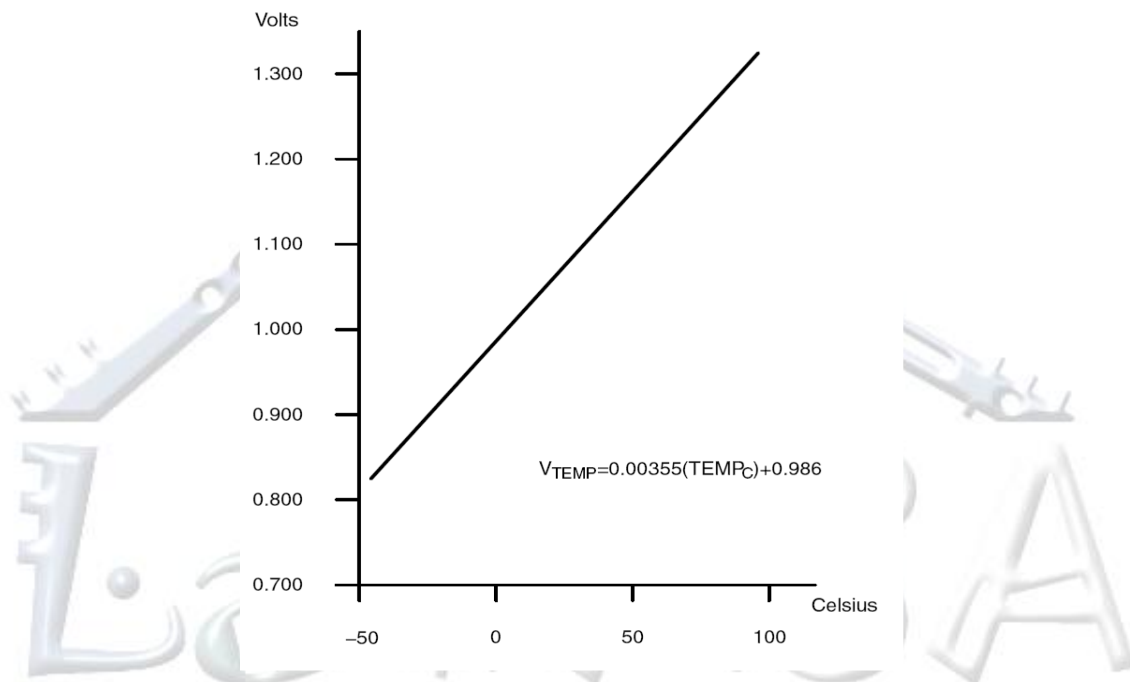


Figure 1. Internal Temperature Sensor Transfer Characteristic: V=f(T)

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

What does the program do?
What are configuration parameters of ADC12 (input channel, clock, reference voltage, sampling time, ...)?
What are configuration parameters of the USART0 module?
How does the temperature sensor work?

```c
/*------------------------------------------------------------------------------
 * File:        Lab11_D1.c (CPE 325 Lab11 Demo code)
 * Function:    Measuring the temperature (MPS430FG4618)
 * Description: This C program samples the on-chip temperature sensor and
 *              converts the sampled voltage from the sensor to temperature in
 *              degrees Celsius and Fahrenheit. The converted temperature is
 *              sent to HyperTerminal over the UART by using RS-232 cable.
 * Clocks:      ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = DCO = default (~1MHz)
 *              An external watch crystal between XIN & XOUT is required for ACLK
 * Instructions:Set the following parameters in HyperTerminal
 *                  Port :        COM1
 *                  Baud rate :   38400
 *                  Data bits:    8
 *                  Parity:       None
 *                  Stop bits:    1
 *                  Flow Control: None
 *
 *                         MSP430xG461x
 *                      -------------------
 *                  /|\|               XIN|-
 *                   | |                  | 32kHz
 *                   --|RST           XOUT|-
 *                     |                  |
 *                     |      P2.4/UCA0TXD|------------>
 *                     |                  | 38400 - 8N1
 *                     |      P2.5/UCA0RXD|<------------
 *                     |                  |
 * Input:       Character Y or y or N or n
 * Output:      Displays Temperature in Celsius and Fahrenheit in HyperTerminal
 * Author:      Aleksandar Milenkovic, milenkovic@computer.org
 *------------------------------------------------------------------------------*/

#include  <msp430xG46x.h>
#include  <stdio.h>

char ch;                    // Holds the received char from UART
unsigned char rx_flag;      // Status flag to indicate new char is received

char gm1[] = "Hello! I am an MSP430. Would you like to know my temperature? (Y|N)";
char gm2[] = "Bye, bye!";
char gm3[] = "Type in Y or N!";

long int temp;                       // Holds the output of ADC
long int IntDegF;                    // Temperature in degrees Fahrenheit
long int IntDegC;                    // Temperature in degrees Celsius

char NewTem[25];

void UART_setup(void) {
    P2SEL |= BIT4+BIT5;              // Set UC0TXD and UC0RXD a
    UCA0CTL1 |= BIT0;                // software reset
    UCA0CTL0 = 0;                    // USCI_A0 control register
    UCA0CTL1 |= UCSSEL_2;            // Clock source SMCLK - 1048576 Hz
    UCA0BR0 = 27;                    // Baud rate - 1048576 Hz / 38400
    UCA0BR1 = 0;
```

```
    UCA0MCTL = 0x94;                // Modulation
    UCA0CTL1 &= ~BIT0;              // Software reset
    IE2 |= UCA0RXIE;                // Enable USCI_A0 RX interrupt
}

void UART_putCharacter(char c) {
    while (!(IFG2 & UCA0TXIFG));    // Wait for TX to be ready
    UCA0TXBUF = c;                  // Put character into TX buffer
}

void sendMessage(char* msg, int len) {
    int i;
    for(i = 0; i < len; i++) {
        UART_putCharacter(msg[i]);
    }
    UART_putCharacter('\n');        // Newline
    UART_putCharacter('\r');        // Carriage return
}

void ADC_setup(void) {
    unsigned int i;
    ADC12CTL0 = SHT0_8 + REFON + ADC12ON; // (256*1/5MHz) > 30 us, 1.5 V
    ADC12CTL1 = SHP;                // Enable sample timer
    ADC12MCTL0 = INCH_10 + SREF_1;  // Channel 10, Vref+
    ADC12IE = ADC12IE + BIT0;       // Enable interrupt
    for (i = 0; i < 0x3600; i++);   // SW delay for reference start-up
}

void main(void) {
    WDTCTL = WDTPW | WDTHOLD;         // Stop watchdog timer
    UART_setup();                     // Setup USCI_A0 module in UART mode
    ADC_setup();                      // Setup ADC12

    rx_flag = 0;                      // RX default state "empty"
    _EINT();                          // Enable global interrupts
    while(1) {
        sendMessage(gm1, sizeof(gm1));// Send a greetings message

        while(!(rx_flag&0x01));       // Wait for input
        rx_flag = 0;                  // Clear rx_flag
        sendMessage(&ch, 1);          // Send received char

        // Character input validation
        if ((ch == 'y') || (ch == 'Y')) {
            ADC12CTL0 |= ENC + ADC12SC; // Sampling and conversion start
            _BIS_SR(CPUOFF + GIE);      // LPM0 with interrupts enabled
            //  oC = ((x/4095)*1500mV)-986mV)*1/3.55mV
            //  IntDegC = (ADCMEM0 - 2692)* 423/4095
            IntDegC = ((temp - 2692) * 423)/4095;
            IntDegF = IntDegC*(9/5) + 32;
            // Printing the temperature on HyperTerminal/Putty
            sprintf(NewTem, "T(F)=%ld\tT(C)=%ld\n", IntDegF, IntDegC);
            sendMessage(NewTem, sizeof(NewTem));
        }
        else if ((ch == 'n') || (ch == 'N')) {
```

```
            sendMessage(gm2, sizeof(gm2));
            break;                          // Get out
        }
        else {
            sendMessage(gm3, sizeof(gm3));
        }
    }                                       // End of while
    while(1);                               // Stay here forever
}

#pragma vector = USCIAB0RX_VECTOR
__interrupt void USCIA0RX_ISR (void) {
    ch = UCA0RXBUF;                     // Copy the received char
    rx_flag = 0x01;                     // Signal to main
    LPM0_EXIT;
}

#pragma vector = ADC12_VECTOR
__interrupt void ADC12ISR (void) {
    temp = ADC12MEM0;                   // Move results, IFG is cleared
    _BIC_SR_IRQ(CPUOFF);                // Clear CPUOFF bit from 0(SR)
}
```

Figure 2. C Program that Samples On-Chip Temperature Sensor

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 configurations. As an example, let us 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 will 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.

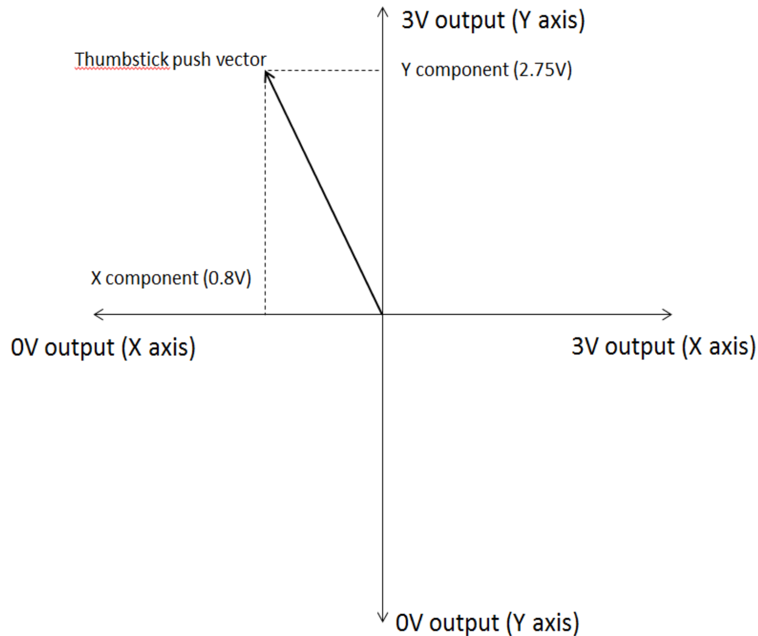


Figure 3. Performance data for hypothetical thumbstick

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

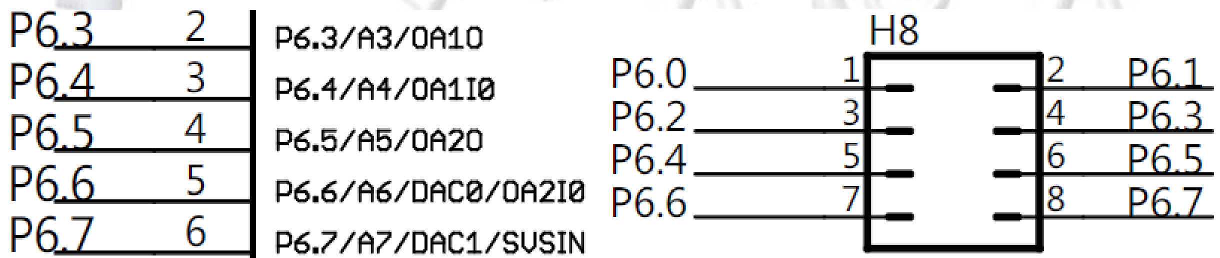


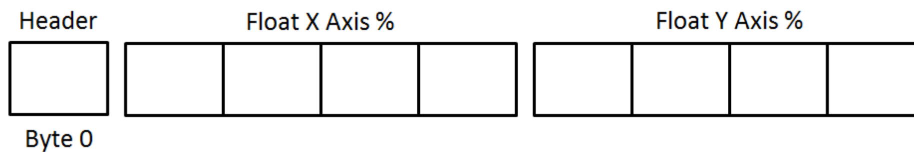
Figure 4. Pinouts and Header Connections for Analog Inputs

Note that the analog input A3 (port P6.3) corresponds to the pin 4 of the H8 header, and analog input A7 (port P6.7) corresponds to the pin 8 of the H8 header. That is where we will connect horizontal HORZ and vertical VERT wires of the thumbstick. 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 will want to have our output as the 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 is 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):

$$Input\ ADC\ Value\ in\ steps \times \frac{3V}{4095} \times \frac{100\%}{3V} = \%Power$$

We could send our information in a variety of ways including a vector format, signed percentage, or even just ADC "steps." If we are using the percentage calculated as shown above, our packet to send to the UAH serial app would look like the one below (1 header byte, 2 single precision floating-point numbers). Figure 5 shows how to configure UAH Serial App to accept two channels including single-precision floating-point numbers. Figure 6 shows signals representing the percentage of HORZ and VERT direction of the thumbstick (read line, CH0, represents HORZ and blue line, CH1, represents VERT) when it is moved along HORZ and VERT axes. The value 100 (100%) of the red line indicates that thumbstick is moved fully in the horizontal direction.



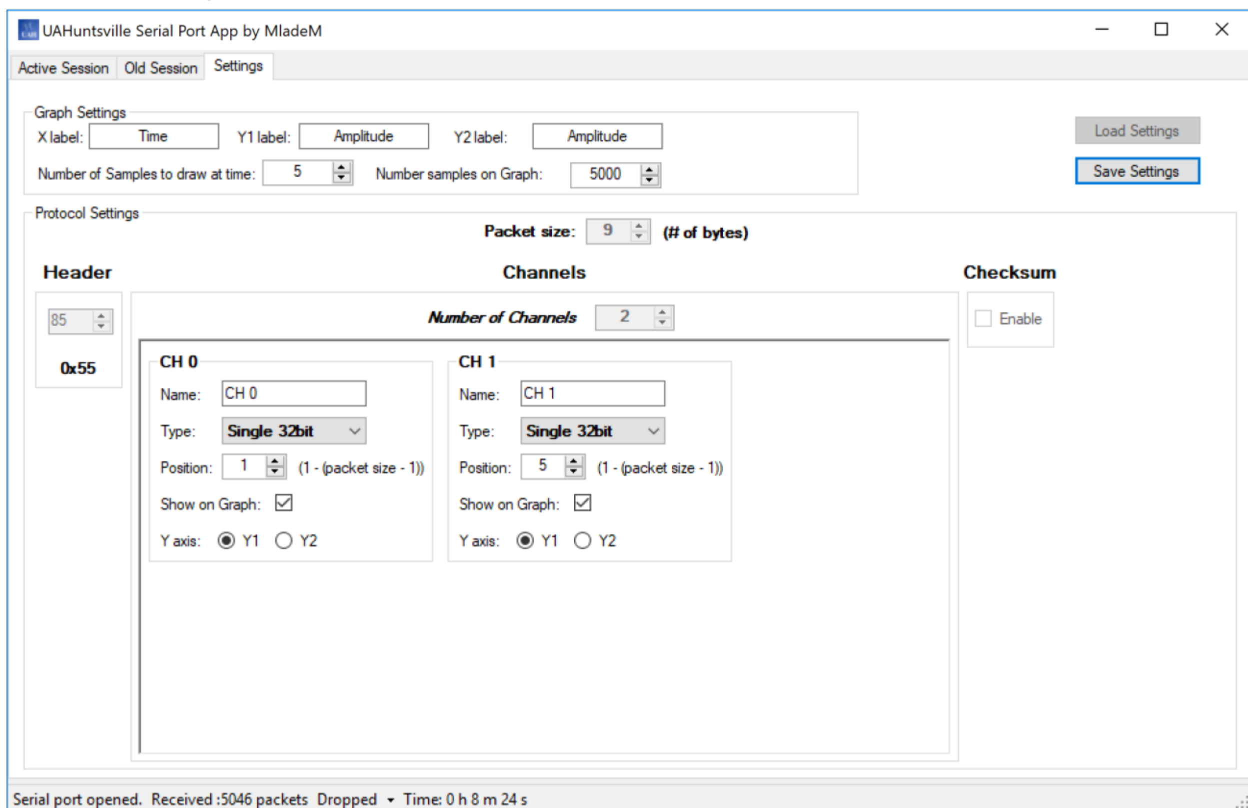


Figure 5. UAH Serial App Settings

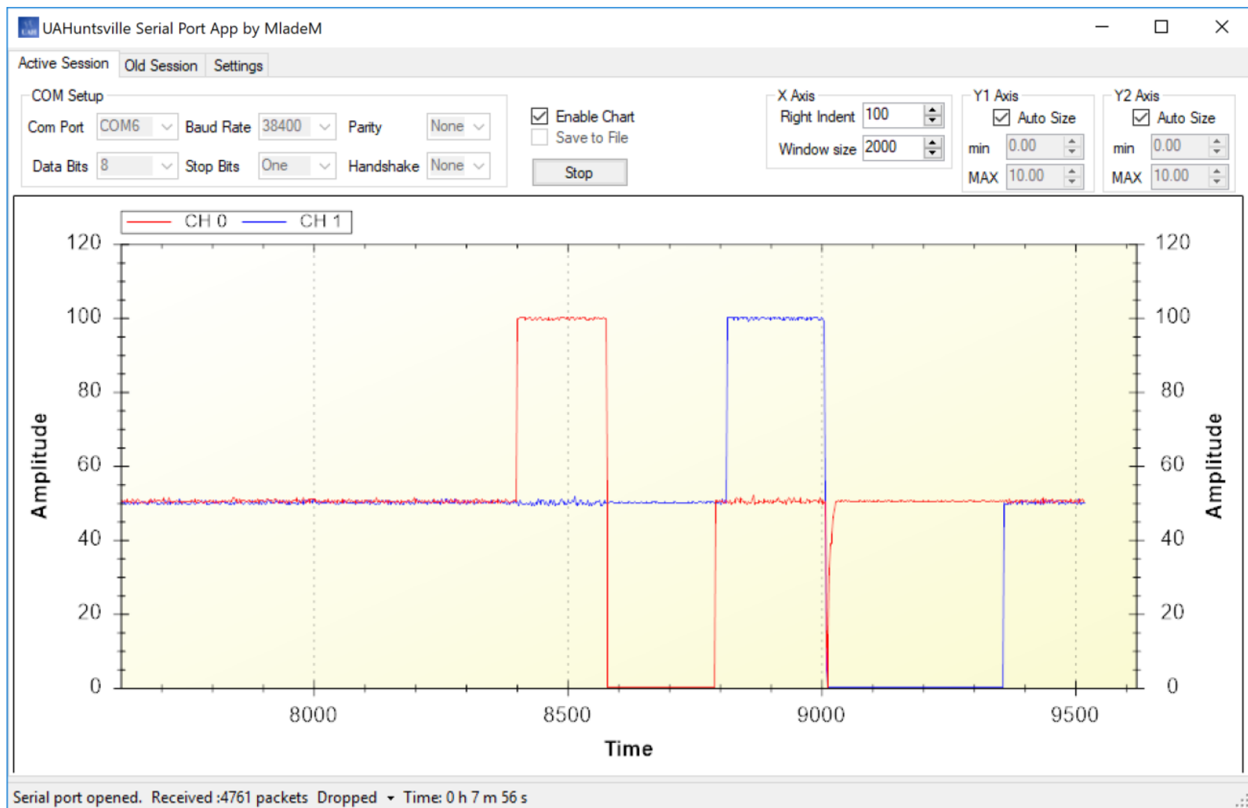


Figure 6. UAH Serial App Showing Percentage Signals from Thumbstick (CH0 – HORZ, CH1 – VERT)

```
/*------------------------------------------------------------------------------
 * File:        Lab11_D2.c (CPE 325 Lab11 Demo code)
 * Function:    Interfacing thumbstick (MPS430FG4618)
 * Description: This C program interfaces with a thumbstick sensor that has
 *              x (HORZ) and y (VERT) axis and outputs from 0 to 3V.
 *              The value of x and y axis
 *              is sent as the percentage of power to the UAH Serial App.
 * Clocks:      ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = DCO = default (~1MHz)
 *              An external watch crystal between XIN & XOUT is required for ACLK
 *
 *                         MSP430xG461x
 *                      -------------------
 *                  /|\|                XIN|-
 *                   | |                   | 32kHz
 *                   --|RST            XOUT|-
 *                     |                   |
 *                     |     P2.4/UCA0TXD|------------>
 *                     |                   | 38400 - 8N1
 *                     |     P2.5/UCA0RXD|<------------
 *                     |                   |
 * Input:       Connect thumbstick to the board
 * Output:      Displays % of power in UAH serial app
 * Author:      Micah Harvey
 *------------------------------------------------------------------------------*/
```

```c
#include <msp430xG46x.h>

volatile long int ADCXval, ADCYval;
volatile float Xper, Yper;

void TimerA_setup(void) {
    TACCR0 = 3277;                      // 3277 / 32768 Hz = 0.1s
    TACTL = TASSEL_1 + MC_1;            // ACLK, up mode
    TACCTL0 = CCIE;                     // Enabled interrupt
}

void ADC_setup(void) {
    int i =0;

    P6DIR &= ~BIT3 + ~BIT7;             // Configure P6.3 and P6.7 as input pins
    P6SEL |= BIT3 + BIT7;               // Configure P6.3 and P6.7 as analog pins
    ADC12CTL0 = ADC12ON + SHT0_6 + MSC; // configure ADC converter
    ADC12CTL1 = SHP + CONSEQ_1;         // Use sample timer, single sequence
    ADC12MCTL0 = INCH_3;                // ADC A3 pin - Stick X-axis
    ADC12MCTL1 = INCH_7 + EOS;          // ADC A7 pin - Stick Y-axis
                                        // EOS - End of Sequence for Conversions
    ADC12IE |= 0x02;                    // Enable ADC12IFG.1
    for (i = 0; i < 0x3600; i++);       // Delay for reference start-up
    ADC12CTL0 |= ENC;                   // Enable conversions
}

void UART_putCharacter(char c) {
    while(!(IFG2 & UCA0TXIFG));         // Wait for previous character to be sent
    UCA0TXBUF = c;                      // Send byte to the buffer for transmitting
}

void UART_setup(void) {
    P2SEL |= BIT4 + BIT5;               // Set up Rx and Tx bits
    UCA0CTL0 = 0;                       // Set up default RS-232 protocol
    UCA0CTL1 |= BIT0 + UCSSEL_2;        // Disable device, set clock
    UCA0BR0 = 27;                       // 1048576 Hz / 38400
    UCA0BR1 = 0;
    UCA0MCTL = 0x94;
    UCA0CTL1 &= ~BIT0;                  // Start UART device
}

void sendData(void) {
    int i;
    Xper = (ADCXval*3.0/4095*100/3);    // Calculate percentage outputs
    Yper = (ADCYval*3.0/4095*100/3);
    // Use character pointers to send one byte at a time
    char *xpointer=(char *)&Xper;
    char *ypointer=(char *)&Yper;

    UART_putCharacter(0x55);            // Send header
    for(i = 0; i < 4; i++) {            // Send x percentage - one byte at a time
        UART_putCharacter(xpointer[i]);
    }
    for(i = 0; i < 4; i++) {            // Send y percentage - one byte at a time
        UART_putCharacter(ypointer[i]);
```

```
    }
}

void main(void) {
    WDTCTL = WDTPW +WDTHOLD;                // Stop WDT
    TimerA_setup();                         // Setup timer to send ADC data
    ADC_setup();                            // Setup ADC
    UART_setup();                           // Setup UART for RS-232
    _EINT();

    while (1){
        ADC12CTL0 |= ADC12SC;                   // Start conversions
        __bis_SR_register(LPM0_bits + GIE); // Enter LPM0
    }
}

#pragma vector = ADC12_VECTOR
__interrupt void ADC12ISR(void) {
    ADCXval = ADC12MEM0;                      // Move results, IFG is cleared
    ADCYval = ADC12MEM1;
    __bic_SR_register_on_exit(LPM0_bits); // Exit LPM0
}

#pragma vector = TIMERA0_VECTOR
__interrupt void timerA_isr() {
    sendData();                               // Send data to serial app
    __bic_SR_register_on_exit(LPM0_bits); // Exit LPM0
}
```

Figure 7 shows demo code that could be used to set up the ADC12 and UART and send the thumbstick information to the UAH Serial App. Analyze the code and answer the following questions.

What does the program do?
What are configuration parameters of ADC12 (input channel, clock, reference voltage, sampling time, ...)?
How many samples per second is taken from ADC12?
How many samples per second per axis is sent to UAH Serial App?

```
1   /*------------------------------------------------------------------------------
2    * File:        Lab11_D2.c (CPE 325 Lab11 Demo code)
3    * Function:    Interfacing thumbstick (MPS430FG4618)
4    * Description: This C program interfaces with a thumbstick sensor that has
5    *              x (HORZ) and y (VERT) axis and outputs from 0 to 3V.
6    *              The value of x and y axis
7    *              is sent as the percentage of power to the UAH Serial App.
8    * Clocks:      ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = DCO = default (~1MHz)
9    *              An external watch crystal between XIN & XOUT is required for ACLK
10   *
11   *                        MSP430xG461x
12   *                     -------------------
13   *                  /|\|              XIN|-
```

```
14   *                    | |                 | 32kHz
15   *                  --|RST           XOUT|-
16   *                    |                   |
17   *                    |      P2.4/UCA0TXD|------------>
18   *                    |                   | 38400 - 8N1
19   *                    |      P2.5/UCA0RXD|<------------
20   *                    |                   |
21   * Input:       Connect thumbstick to the board
22   * Output:      Displays % of power in UAH serial app
23   * Author:      Micah Harvey
24   *------------------------------------------------------------------------------*/
25
26  #include <msp430xG46x.h>
27
28  volatile long int ADCXval, ADCYval;
29  volatile float Xper, Yper;
30
31  void TimerA_setup(void) {
32      TACCR0 = 3277;                      // 3277 / 32768 Hz = 0.1s
33      TACTL = TASSEL_1 + MC_1;            // ACLK, up mode
34      TACCTL0 = CCIE;                     // Enabled interrupt
35  }
36
37  void ADC_setup(void) {
38      int i =0;
39
40      P6DIR &= ~BIT3 + ~BIT7;             // Configure P6.3 and P6.7 as input pins
41      P6SEL |= BIT3 + BIT7;               // Configure P6.3 and P6.7 as analog pins
42      ADC12CTL0 = ADC12ON + SHT0_6 + MSC; // configure ADC converter
43      ADC12CTL1 = SHP + CONSEQ_1;         // Use sample timer, single sequence
44      ADC12MCTL0 = INCH_3;                // ADC A3 pin - Stick X-axis
45      ADC12MCTL1 = INCH_7 + EOS;          // ADC A7 pin - Stick Y-axis
46                                          // EOS - End of Sequence for Conversions
47      ADC12IE |= 0x02;                    // Enable ADC12IFG.1
48      for (i = 0; i < 0x3600; i++);       // Delay for reference start-up
49      ADC12CTL0 |= ENC;                   // Enable conversions
50  }
51
52  void UART_putCharacter(char c) {
53      while(!(IFG2 & UCA0TXIFG));         // Wait for previous character to be sent
54      UCA0TXBUF = c;                      // Send byte to the buffer for transmitting
55  }
56
57  void UART_setup(void) {
58      P2SEL |= BIT4 + BIT5;               // Set up Rx and Tx bits
59      UCA0CTL0 = 0;                       // Set up default RS-232 protocol
60      UCA0CTL1 |= BIT0 + UCSSEL_2;        // Disable device, set clock
61      UCA0BR0 = 27;                       // 1048576 Hz / 38400
62      UCA0BR1 = 0;
63      UCA0MCTL = 0x94;
64      UCA0CTL1 &= ~BIT0;                  // Start UART device
65  }
66
67  void sendData(void) {
68      int i;
```

```
69      Xper = (ADCXval*3.0/4095*100/3);    // Calculate percentage outputs
70      Yper = (ADCYval*3.0/4095*100/3);
71      // Use character pointers to send one byte at a time
72      char *xpointer=(char *)&Xper;
73      char *ypointer=(char *)&Yper;
74
75      UART_putCharacter(0x55);            // Send header
76      for(i = 0; i < 4; i++) {            // Send x percentage - one byte at a time
77          UART_putCharacter(xpointer[i]);
78      }
79      for(i = 0; i < 4; i++) {            // Send y percentage - one byte at a time
80          UART_putCharacter(ypointer[i]);
81      }
82  }
83
84  void main(void) {
85      WDTCTL = WDTPW +WDTHOLD;            // Stop WDT
86      TimerA_setup();                     // Setup timer to send ADC data
87      ADC_setup();                        // Setup ADC
88      UART_setup();                       // Setup UART for RS-232
89      _EINT();
90
91      while (1){
92          ADC12CTL0 |= ADC12SC;               // Start conversions
93          __bis_SR_register(LPM0_bits + GIE); // Enter LPM0
94      }
95  }
96
97  #pragma vector = ADC12_VECTOR
98  __interrupt void ADC12ISR(void) {
99      ADCXval = ADC12MEM0;                  // Move results, IFG is cleared
100     ADCYval = ADC12MEM1;
101     __bic_SR_register_on_exit(LPM0_bits); // Exit LPM0
102 }
103
104 #pragma vector = TIMERA0_VECTOR
105 __interrupt void timerA_isr() {
106     sendData();                           // Send data to serial app
107     __bic_SR_register_on_exit(LPM0_bits); // Exit LPM0
108 }
```

Figure 7. C Program that takes the x- and y- axis Samples from a Thumbstick

3 Digital-to-Analog Converters

So far we have explored analog-to-digital converters that convert an analog input to an integer value that can be stored and processed by the microcontroller. Conversely, digital-to-analog converters take a numerical value that provided by the user and output its corresponding analog voltage. The concepts of resolution and reference voltages are seen here again and should be understood to make the best use of the digital to analog converter.

The MSP430FG4618 has a digital-to-analog peripheral (DAC12) with two channels. Each channel has one control register and one data register. The control register determines the output pin, output level, reference voltage, 8- or 12-bit resolution, when the output latch triggers, the amplifier settings, and the interrupt settings. The data register holds the value to be converted to an analog voltage output.

The DAC12 operation is much simpler than the ADC12. There is no timer associated with the device, so one of the MSP's timers such as the Timer_A must be used. Likewise, there is no internal voltage reference, so it is easiest to configure the ADC12 voltage reference to be used. Once the DAC12 is configured and enabled, a timer interrupt can periodically call an ISR that writes a new value to the DAC12 data register. By default, when the data register is written, a new output voltage is generated.

As discussed in the previous section, it is important to optimize your signal's vertical (time) and horizontal (voltage) resolution for your requirements. Figure 8 below is an example of a D/A converter output waveform.

Notice how the output looks very blocky. With better resolution, the "blockiness" of the signal is minimized. The resolution is a function of how often the signal data is updated, the resolution of the converter, and the output waveform algorithm.

In this project, we will create a lookup table using MATLAB. The lookup table is a large array stored in memory that holds individual values that will be passed to the DAC12 in sequence to output a signal. While creating our waveform lookup table, it is important to keep in mind that we can increase our output resolution by increasing the number of samples used per period of our waveform. That also requires us to use a faster timer to refresh our DAC12 module to maintain the correct frequency.

MATLAB works by manipulating matrices. Below you will see an example of how to create and save a lookup table consisting of 256 values for a sine wave.

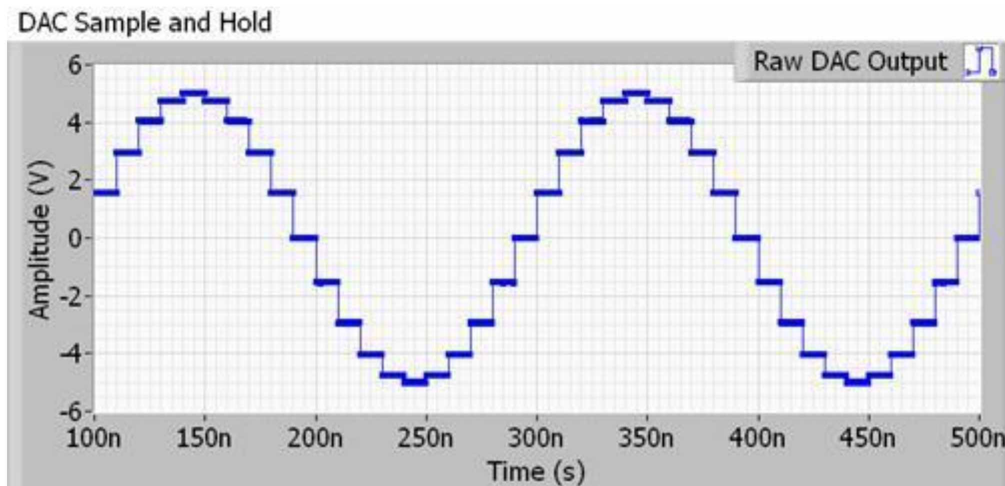


Figure 8. D/A Converter Output Waveform

3.1 Sinusoidal Wave Generator

Let us consider the following application. Your task is to develop a sinusoidal wave generator using the TI Experimenter's Board. The function you should implement is described as follows: y = 1.25*(1+sin(x)). The maximum voltage is consequently 2.5 V and the minimum voltage is 0 V. The frequency of the sine wave should be 10 Hz, that this one period should take 10 ms.

The signal is generated using the MSP430's DAC12 digital-to-analog converter. Periodically we will be sending a new digital value to the DAC12. These values are prepared in advance in a constant table. Let us first discuss how to create this lookup table. Our first step is to determine the number of samples we want to have and the number of bits to represent them. More samples, means a better quality of the generated sine wave. Let us assume that we want to have 256 samples of the function y in a lookup table, each sample with a value in the range of [0 ... 4095]. The sample with value 0 corresponds to 0 V, and samples with the value 0xFFF (4095) corresponds to 2.5 V. We can use MATLAB to generate the lookup table for x=0, x, 2x,..2*pi, where x=2*pi/256. Figure 9 shows the MATLAB program. We generate 256 samples (actually 257) in the range x=[0...2*pi] and calculate the integer values that correspond to each sample x, using the original function y=1.25*(1+sin(x)). The values are rounded to nearest integers and the samples are written back to a file. Note: the actual number of samples is 257, so we should remove the last one (the same as the first). We should also ensure that no samples are larger than 4095 (the largest number that can be represented by 12-bits). These constants will be used to initialize the lookup table visible to your program (here we create a header file named sine_lut_256.h).

```
x=(0:2*pi/256:2*pi);
y=1.25*(1+sin(x));
dac12=y*4095/2.5;
dac12r = round(dac12);
dlmwrite('sine_lut_256.h',dac12r, ',');
```

Figure 9 Matlab program to generate a 256-entry lookup table

The next step is to determine the trigger period. We want 256 samples to spread over 2*pi range of x. The required period of the sine wave is 10 Hz (0.1sec). That means that the trigger period is 0.1/256 sec. We will use a TimerA device to generate triggers. Assuming that the default clock frequency on SMCLK is used as the timer clock (1048576 Hz), we can determine the value that needs to be written to the TimerA counter (0.1/256)*FSMCLK = 410.

Figure 10 shows the complete program. We stop the watchdog time, initialize the ADC12 to give a reference voltage of 2.5 V, and initialize the timer to raise an interrupt every 0.1/256 sec. The TimerA ISR wakes the processor, we read the next sample from the table, and output it to the DAC12.

```
1   /*------------------------------------------------------------------------------
2    * File:        Lab11_D3.c (CPE 325 Lab11 Demo code)
3    * Function:    Sinusoidal wave with DAC (MPS430FG4618)
4    * Description: This C program reconstructs the sinusoidal wave (y=1.25(1+sin(x)))
5    *              from the samples using DAC and outputs at P6.6. WDT is used to
6    *              give an interrupt for every ~0.064ms to wake up the CPU and
7    *              feed the DAC with new value. Connect the oscilloscope to P6.6
8    *              to observe the signal. The interval used to read the samples
9    *              controls the frequency of output signal.
10   * Clocks:      ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = DCO = default (~1MHz)
11   *              An external watch crystal between XIN & XOUT is required for ACLK
12   *
13   *                         MSP430xG461x
14   *                      -------------------
15   *                   /|\|              XIN|-
16   *                    | |                 | 32kHz
17   *                    --|RST          XOUT|-
18   *                      |                 |
19   *                      |        DAC0/P6.6|--> sine (10Hz)
20   *                      |                 |
21   * Input:       None
22   * Output:      Sinusoidal wave with 10Hz frequency at P6.6
23   * Author:      Aleksandar Milenkovic, milenkovic@computer.org
24    *------------------------------------------------------------------------------*/
25  #include <msp430fg4618.h>
26  #include "sine_lut_256.h"  /*256 samples are stored in this table */
27
28  void TimerA_setup(void) {
29      TACTL = TASSEL_2 + MC_1;                // SMCLK, up mode
30      TACCR0 = 410;                           // Sets Timer Freq (1048576*0.1sec/256)
31      TACCTL0 = CCIE;                         // CCR0 interrupt enabled
32  }
33
```

```
34   void DAC_setup(void) {
35       ADC12CTL0 = REF2_5V + REFON;          // Turn on 2.5V internal ref volage
36       unsigned int i = 0;
37       for (i = 50000; i > 0; i--);          // Delay to allow Ref to settle
38       DAC12_0CTL = DAC12IR + DAC12AMP_5 + DAC12ENC;   //Sets DAC12
39   }
40
41   void main(void) {
42       WDTCTL = WDTPW + WDTHOLD;      // Stop WDT
43       TimerA_setup();                // Set timer to uniformly distribute the samples
44       DAC_setup();                   // Setup DAC
45       unsigned int i = 0;
46       while (1) {
47           __bis_SR_register(LPM0_bits + GIE); // Enter LPM0, interrupts enabled
48           DAC12_0DAT = LUT256[i];
49           i=(i+1)%256;
50       }
51   }
52
53   #pragma vector = TIMERA0_VECTOR
54   __interrupt void TA0_ISR(void) {
55       __bic_SR_register_on_exit(LPM0_bits);  // Exit LPMx, interrupts enabled
56   }
```

Figure 10. C Program to Generate Sine Wave Output Using DAC12

4 References

To understand more about the ADC12 peripheral and its configuration, please refer the following materials:

- Davies Text, pages 407-438 and pages 485-492
- MSP430FG4618 User's Guide, Chapter 28, pages 787-814 (ADC12)
- MSP430FG4618 User's Guide, page 802 (Internal temperature sensor)
- MSP430FG4618 User's Guide, Chapter 31, pages 869-886 (DAC12)