CPE 325: Intro to Embedded Computer System

Lab 9

Analog to Digital Converter, Digital to Analog Converter

Submitted by: Hannah Kelley

Date of Experiment: November 3, 2025

Report Deadline: November 10, 2025

Demonstration Deadline: November 10, 2025

Theory

Topic 1: Accelerometers

Accelerometers are sensors used to measure the force of acceleration in one or more directions. They can detect changes in motion or orientation by measuring the combined affects of static acceleration (gravity) and dynamic acceleration (from movement or vibration). In this lab the ADXL335 accelerometer was used to measure acceleration in the x, y, and z directions. The magnitude of force in each direction is sent to the MSP430f5539 in the form of an proportional analog voltage and converted using the built-in Analog-to-Digital Converter (ADC).

Topic 2: ADC and DAC

Analog-to-Digital Converters (ADCs) and Digital-to-Analog Converters (DACs) are essential components for any computer interfacing between analog and digital systems. An ADC converts continuous analog signals, such as voltages from sensors like the ADXL335 used in this lab, into discrete digital values that microcontrollers and other computers and process. This allows physical data such as acceleration, temperature, or sound intensity, to be represented numerically. In contrast, a DAC performs the opposite function and converts digital values into analog signals. This is useful for generating analog waveforms or recreating sensor outputs.

Results & Observation

Program 1 and 2:

Program Description:

In this lab, a C program was written to interface the ADXL335 three-dimensional accelerometer with the MSP430F5539 microcontroller. The program sampled the accelerometer's x, y, and z outputs at a rate of 10 samples per second, converted the analog voltages to digital values using the onboard ADC, and calculated the acceleration in units of g (acceleration due to gravity). The results for each axis were transmitted to a workstation and displayed as three separate graphs in the UAH Serial App.

In the second part of the lab, the program was expanded to demonstrate a potential airbag crash sensor application. The total acceleration magnitude, $M = \sqrt{x^2 + y^2 + z^2}$, was computed using data from all three axes. When the measured magnitude exceeded a critical threshold of 3g, the red LED on the MSP430 board illuminated to indicate simulated airbag deployment. Otherwise, the LED remained off under normal conditions.

Program Output:

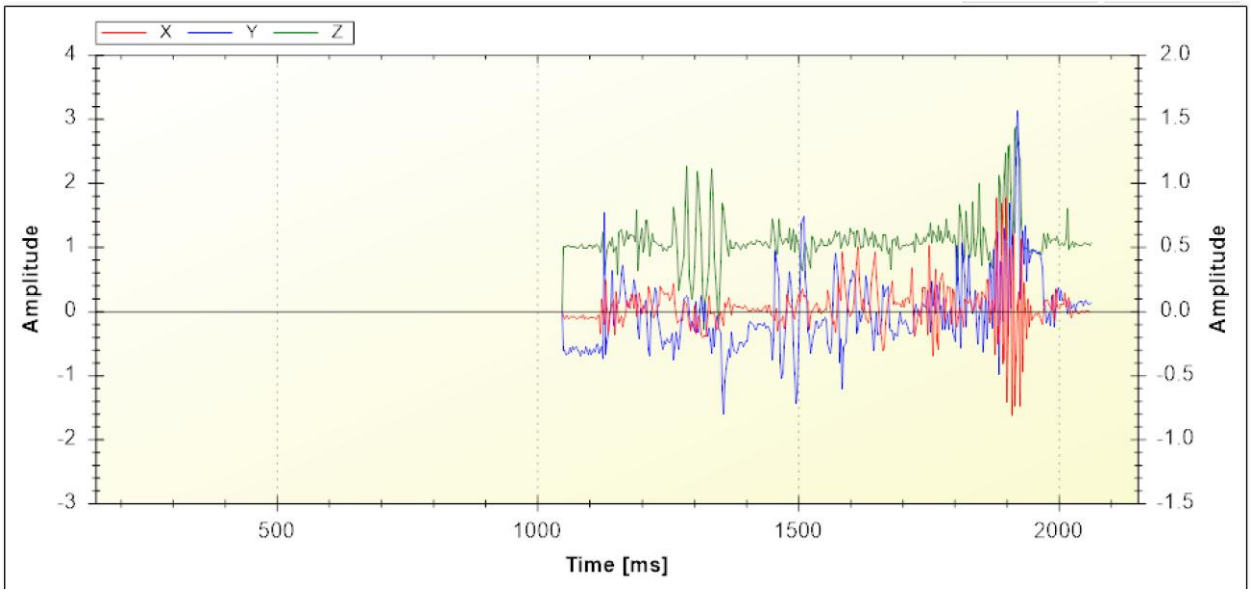

Figure 01: UAH Serial App Accelerometer Output

Report Questions:

1. **What formula was used for accelerometer interfacing?**

   The accelerometer used in this experiment, the ADXL335, produces an analog voltage output proportional to the acceleration experienced along each axis (X, Y, and Z). At rest, each axis outputs approximately 1.5 V, which corresponds to 0 g of acceleration. The sensitivity of the sensor is approximately 0.3 V per g, meaning that for every 0.3 V change from the zero-g level, the output represents a change of ±1 g.

   Since the MSP430 microcontroller measures voltage through a 12-bit analog-to-digital converter (ADC), the ADC produces a digital value between 0 and 4095 corresponding to input voltages from 0 V to 3.0 V. To convert these ADC readings into acceleration values in units of g, the following formula was used:

   $$g = \frac{\left(3 * \frac{n}{4095} - 1.5\right)}{0.3}$$

   This formula—defined as a macro for use in code—first converts the ADC output to a voltage, then subtracts the 1.5 V zero-g offset, and finally divides by the 0.3 V/g sensitivity to obtain acceleration in g. This conversion allows the MSP430 to interpret the accelerometer's analog signals as meaningful physical accelerations, which can then be displayed in the UAH Serial App for monitoring and analysis.

2. **What sampling rate was used for the accelerometer and why?**

   In this experiment, the accelerometer was sampled at a rate of 10 samples per second (10 Hz). The sampling rate was determined by configuring Timer_A on the MSP430 microcontroller to generate an interrupt every 0.1 seconds, corresponding to a 10 Hz update interval. Each time

the timer interrupt occurred, new ADC conversions were triggered, and the resulting acceleration data were transmitted to the UAH Serial App

A sampling rate of 10 Hz was chosen because it provides a suitable balance between data resolution and system performance for this application. The motion being measured by the ADXL335—such as tilt or slow movement of the sensor—is relatively low frequency and does not require high-speed sampling. Higher sampling rates would increase processor workload and data transmission demands without providing additional useful information for slow or moderate motion. Therefore, a 10 Hz sampling rate was sufficient to accurately capture gradual changes in acceleration while maintaining efficient use of processing time, power, and serial communication bandwidth.

Program Flowchart:

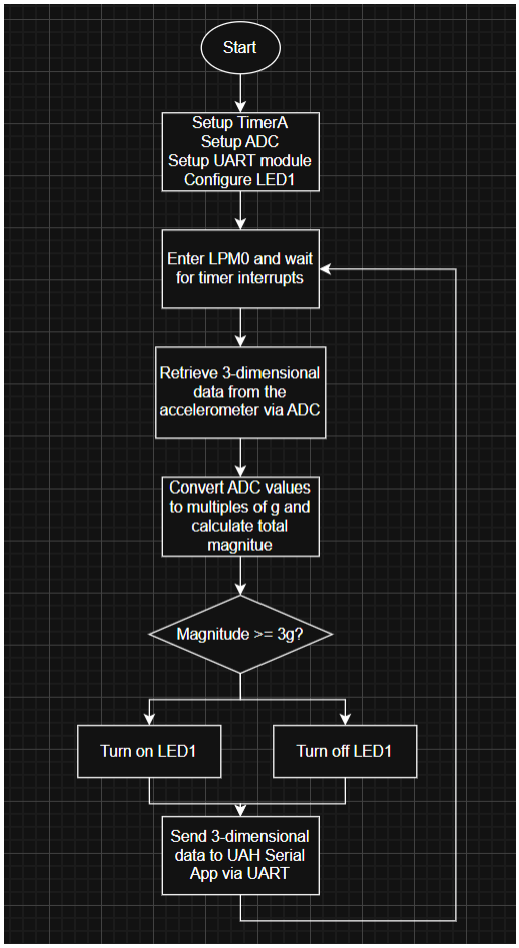

Figure 02: Program 1 and 2 Flowchart

Program 3:

Program Description:

For this experiment, a digital-to-analog converter (DAC) was constructed using an 8-bit R–2R resistor ladder network, as shown in Figure 3. This circuit converts the 8-bit digital output from the MSP430F5529 microcontroller into an analog voltage. The R–2R ladder was built using 1 kΩ and 2 kΩ resistors, where each bit of the digital input contributes a weighted portion of the total output voltage. The least significant bit (BIT0) provides the smallest contribution, while the most significant bit (BIT7) provides the largest.

The DAC circuit was connected to Port 3 of the MSP430F5529, which provides all eight digital output pins required for the conversion. This port drives the R–2R ladder directly, generating an analog voltage corresponding to the digital code output by the microcontroller.

The desired functionality of the system was as follows:

- The DAC generates waveforms at a default frequency of 30 Hz.
- When no switches are pressed, the output waveform is a sine wave, created using a lookup table stored in memory.
- When switch SW1 is pressed, a sawtooth waveform is generated on the fly, without using a lookup table.
- When switch SW2 is pressed, the output frequency quadruples, producing a waveform at 120 Hz instead of 30 Hz.

This implementation demonstrates the process of generating and controlling analog signals using digital data, emphasizing the flexibility of DAC-based waveform synthesis through microcontroller programming.

Program Output:

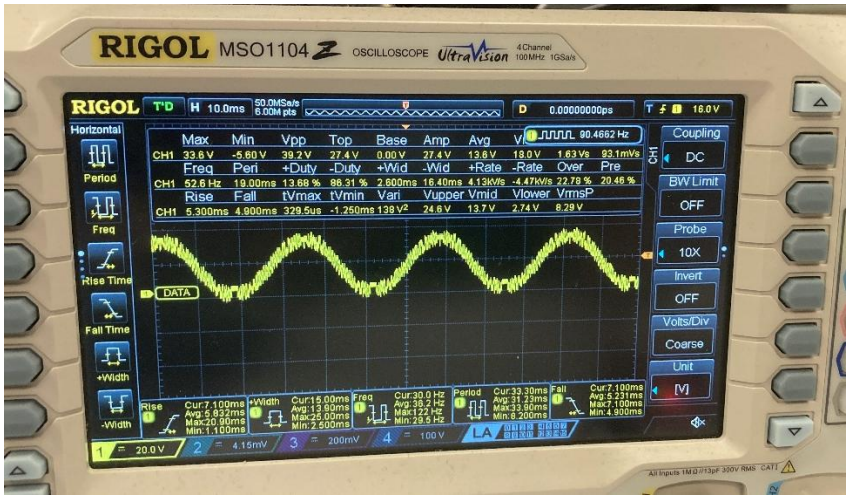

Figure 03: Oscilloscope Output with No Switches Pressed

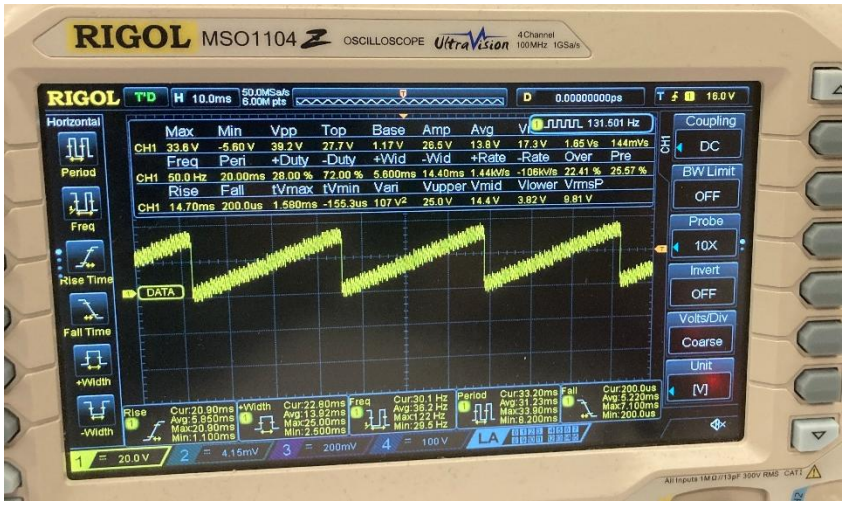

Figure 04: Oscilloscope Output with Switch 1 Pressed

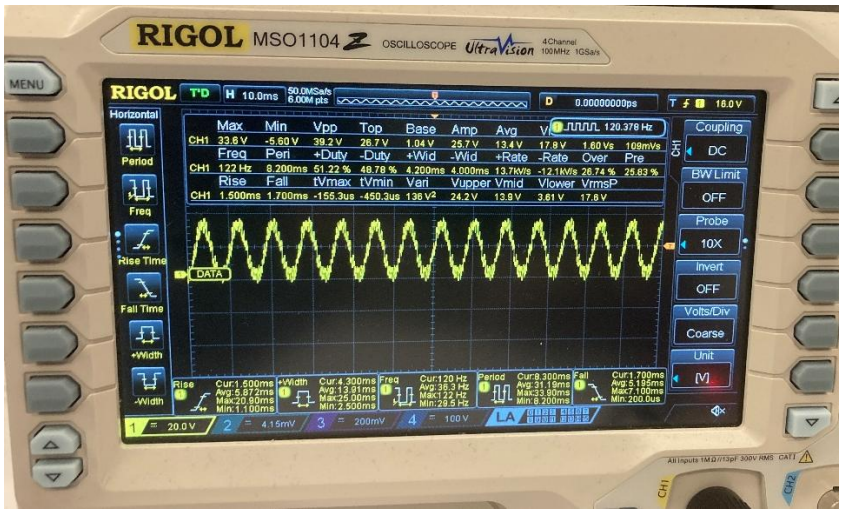

Figure 05: Oscilloscope Output with Switch 2 Pressed

Conclusion

I encountered several challenges during this lab, particularly when configuring the UAH Serial App and learning to use the oscilloscope. Even after implementing the provided UART code for interfacing with the Serial App, I initially struggled to display the accelerometer data on the graphs. After several troubleshooting attempts, I ultimately resolved the issue by redownloading the app and re-entering the same configuration settings from a file—though the exact cause of the problem remains unclear.

This lab also marked my first experience using an oscilloscope, which presented its own learning curve. However, by the end of the experiment, I gained a much better understanding of how to operate it effectively and interpret its readings.

This lab provided valuable experience working with both the ADC and DAC modules of the microcontroller. Through the experiments, I gained a better understanding of how analog signals are sampled, digitized, and reconstructed. Configuring the ADC taught me how resolution, reference

voltage, and sampling rate affect measurement accuracy, while working with the DAC demonstrated how digital data can be converted back into analog output. Overall, this lab helped bridge the gap between theoretical signal concepts and practical hardware implementation, reinforcing the importance of precise configuration and signal integrity in embedded systems.

Appendix

Table 01: Program 1 Source Code

```
/*-------------------------------------------------------------------------
 * File: Lab9_P1_P2.c
 * Desciption: Take accelerometer input values and display them in UAH Serial App.
 * When the magnitude of the input values is greater than or equal to 3g turn on
 * the RED LED (LED1)
 *
 * Input: accelerometer x, y, and z values
 * Output: ADC values to UAH Serial App and LED1
 * Author: Hannah Kelley
 *-------------------------------------------------------------------------*/
#include <msp430.h>
#include <math.h>
#define ADC_TO_G(n) ((3.0 * n / 4095 - 1.5) / 0.3) //convert to g
volatile long int ADC_x, ADC_y, ADC_z;
volatile float x_per, y_per, z_per;
volatile float magnitude;
void TimerA_setup(void) {
   TA0CCTL0 = CCIE;                 // Enabled interrupt
   TA0CCR0 = 3276;                  // 3277 / 32768 = .1s for ACLK
   TA0CTL = TASSEL_1 + MC_1 + TACLR;    // ACLK, up mode
   P1DIR |= BIT0;                   // brief blink to show end of setup
}
void ADC_setup(void) { // given in demo code
   int i =0;
   P6DIR &= ~(BIT3 + BIT4 + BIT5);          // Configure P6.3, P6.4 as input pins
   P6SEL |= BIT3 + BIT4 + BIT5;             // Configure P6.3, P6.4 as analog pins
   // configure ADC converter
   ADC12CTL0 = ADC12ON | ADC12SHT0_6 | ADC12MSC;
   ADC12CTL1 = ADC12SHP | ADC12CONSEQ_3;       // Use sample timer, single sequence
   ADC12MCTL0 = ADC12INCH_3;             // ADC A3 pin -X-axis
   ADC12MCTL1 = ADC12INCH_4;         // ADC A4 pin - Y-axis
   ADC12MCTL2 = ADC12INCH_5 | ADC12EOS;
                                // EOS - End of Sequence for Conversions
   ADC12IE = ADC12IE0;                  // Enable ADC12IFG.1
   for (i = 0; i < 0x3600; i++);      // Delay for reference start-up
   ADC12CTL0 |= ADC12ENC;                // Enable conversions
}
void UART_putCharacter(char c) { // given in demo code
   while (!(UCA1IFG&UCTXIFG));    // Wait for previous character to transmit
   UCA1TXBUF = c;               // Put character into tx buffer
}
void UART_setup(void) { // given in demo code
   P4SEL |= BIT4 + BIT5;             // Set up Rx and Tx bits
   UCA1CTL0 = 0;                    // Set up default RS-232 protocol
   UCA1CTL1 |= BIT0 + UCSSEL_2;       // Disable device, set clock
```

```
   UCA1BR0 = 27;                  // 1048576 Hz / 38400
   UCA1BR1 = 0;
   UCA1MCTL = 0x94;
   UCA1CTL1 &= ~BIT0;             // Start UART device
}
void sendData(void) {
   int i;
   // Part 1 - get samples from ADC
   x_per = (ADC_TO_G(ADC_x));   // Calculate percentage outputs
   y_per = (ADC_TO_G(ADC_y));
   z_per = (ADC_TO_G(ADC_z));
   // Use character pointers to send one byte at a time
   unsigned char *xptr=(unsigned char *)&x_per;
   unsigned char *yptr=(unsigned char *)&y_per;
   unsigned char *zptr=(unsigned char *)&z_per
   UART_putCharacter(0x55);         // Send header
   for(i = 0; i < 4; i++) {         // Send x percentage - one byte at a time
      UART_putCharacter(xptr[i]);
   }
   for(i = 0; i < 4; i++) {         // Send y percentage - one byte at a time
      UART_putCharacter(yptr[i]);
   }
   for(i = 0; i < 4; i++) {         // Send z percentage - one byte at a time
      UART_putCharacter(zptr[i]);

   magnitude = sqrt((x_per * x_per) + (y_per * y_per) + (z_per * z_per));
   if (magnitude >= 3) // When the magnitude reaches the critical value of 2G, activate LED1;
otherwise, keep LED1 turned off.{
      P1OUT |= BIT0;
   }
   else {
      P1OUT &= ~BIT0;
   }
}
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) {
   ADC12IFG = 0x00;
```

```
  ADC_x = ADC12MEM0;            // Move results, IFG is cleared
  ADC_y = ADC12MEM1;
  ADC_z = ADC12MEM2;
  __bic_SR_register_on_exit(LPM0_bits); // Exit LPM0
}
#pragma vector = TIMER0_A0_VECTOR
__interrupt void timerA_isr() {
  sendData();                  // Send data to serial app
  __bic_SR_register_on_exit(LPM0_bits); // Exit LPM0
}
```

Table 02: Program 2 Source Code

```
/*-------------------------------------------------------------------------
 * File:        Lab09_P3.c
 * Description: generates waveforms at 30-120 Hz based on switch inputs
 *
 * Board:       5529
 * Input:       SW1 and SW2
 * Output:      waveforms on oscilloscope
 * Author:      Hannah Kelley
 * Date:        November 3, 2025
 *-------------------------------------------------------------------------*/
#include <msp430.h>
#include <stdint.h>
#include <stdbool.h>
#include "Assignment_p3_sine_lut_256.h"
#define SW1 (P2IN&BIT1)
#define SW2 (P1IN&BIT1)
unsigned char index = 0; // for tracking sine wave lookup index
int main() {
    WDTCTL = WDTPW | WDTHOLD;
    // Timer B config
    TB0CCTL0 = CCIE; // enable interrupts
    TB0CTL = TBSSEL_2 + MC_1; // SMCLK in up mode
    TB0CCR0 = 135; // 2 kHz frequency
    // P3 config
    P3DIR = 0xFF; // all P3 pins set to output
    P3OUT = 0; // initialize all pins to 0
    P1DIR = BIT0; // P1.0 = red LED
    // SW1 config
    P2DIR &= ~BIT1; // P2.1 input
    P2REN |= BIT1;  // enable pullup register
    P2OUT |= BIT1; // set pullup register
    // SW2 config
    P1DIR &= ~BIT1; // P2.1 input
    P1REN |= BIT1;  // enable pullup register
    P1OUT |= BIT1; // set pullup register

    while (1) {
        __bis_SR_register(LPM0_bits + GIE);
    }
}
// Timer A ISR
#pragma vector = TIMERB0_VECTOR
```

```
__interrupt void timer2ISR(void) {
    TB0CCTL0 &= ~CCIFG;
    if ((SW2 == 0) && (!(SW1 == 0))) {
        index += 4;
        P3OUT = lut256[index];
    }
    else if ((SW1 == 0) && (!(SW2 == 0))) {
        index++;
        P1OUT ^= BIT0;
        P3OUT = index;
    }
    else {
        index++;
        P3OUT = lut256[index];
    }
}
```