CPE 325: Intro to Embedded Computer System

Lab10

Serial Communication and UART

Submitted by: Gianna Foti

Date of Experiment:______30 October 2024_______________

Report Deadline:_______31 October 2024________________

Lab Section: ______CPE353-02______

Demonstration Deadline: _____31 October 2024___________

Introduction

In this lab, I developed two programs using UART communication on the MSP430 microcontroller. The chatbot, “Market Bot,” runs over a 115,200 baud connection, interacting with users to take their name, process orders for items (eggs, chicken, and milk), and calculate the total cost. It handles invalid inputs with prompts and supports multiple orders per session. Bonus features include backspace handling for input correction as well as ANSI colors to enhance the interaction, with red for bot messages and purple for user responses.. The second program generates a triangular wave with an 8-unit amplitude and 2.5 Hz frequency, transmitting it via a 57,600 baud connection to the UAH Serial app for four cycles. This task highlights the importance of precise timing and waveform generation in embedded systems. This lab provided practical experience with UART communication, user input handling, and signal generation.

Theory Topics

1. Accelerometers
    a. These devices measure the acceleration force exerted on them in multiple dimensions. The ADXL335, used in this lab, provides analog output voltages proportional to the accelerations in each axis (X, Y, and Z) and needs to be converted from analog to digital using the MSP430's ADC12 module. In applications like crash detection systems, accelerometers play a critical role. Here, the microcontroller monitors the combined acceleration magnitude from all axes. If this magnitude exceeds a predefined threshold, such as 2g, it indicates a significant impact or crash. In response, the system initiates specific actions—such as turning on an LED—to simulate airbag deployment. This mechanism is particularly useful in automotive safety systems, where accelerometers can trigger safety responses in real-time.
2. ADC and DAC
    a. The ADC12 on the MSP430F5529 converts the accelerometer's analog signals to digital values for processing. The 12-bit resolution provides 4096 distinct levels, allowing for

fine measurement of voltage variations from the accelerometer, thus capturing small accelerations.

   b. The lab includes generating waveforms via DAC, simulating different signals such as sine and sawtooth. For tasks like waveform generation, timing and precision are essential, which are managed by setting reference voltages and using timers like Timer A and Timer B in the MSP430

3. Explain why you chose the sampling rate you used for the accelerometer

   a. The sampling rate of 10 samples per second was selected to effectively capture changes in acceleration along the X, Y, and Z axes, providing enough detail for monitoring movement or impact without overwhelming the microcontroller with excessive data. This rate translates to a sampling period of 0.1 seconds. Given the ACLK frequency of 32768 Hz, the timer count needed to achieve this period is approximately 3276. By setting TA0CCR0 to 3276, the timer triggers every 0.1 seconds, achieving the required sampling rate with precise timing.

4. Explain how you chose a threshold value for the bubble level

   a. There was no bubble level in this lab.

Part 1 and Part 2:

Program Description:

For Part 1, this program interfaces with the ADXL335 accelerometer to capture real-time acceleration data on the x, y, and z axes. The data from each axis is sampled 10 times per second, converted to gravitational acceleration (g), and transmitted to the UAH serial app via UART. The output for each axis (X, Y, Z) is sent as separate percentage values, providing real-time monitoring of positional changes on the serial app. In Part 2, the program detects a crash event by calculating the total acceleration magnitude from the x, y, and z axes. When this magnitude reaches or exceeds 2g, simulating a crash threshold, the

RED LED on the MSP430F5529 turns on, indicating potential airbag deployment. This threshold-based signaling enhances the practical utility of the accelerometer as a crash sensor.

Process (the demo was very helpful):

1. First, I set up UART communication:
    a. I configured the TX (P3.3) and RX (P3.4) pins and initialized UART at a 38,400 baud rate, allowing smooth data transmission to the UAH serial app to display the accelerometer readings.
2. Next, I configured the timer and ADC:
    a. I set up Timer A to trigger every 0.1 seconds, creating a consistent interval for sampling the accelerometer data. I configured the ADC to read the x, y, and z analog signals from the ADXL335, converting them into digital values for processing.
3. Then, I handled the conversion and transmission of accelerometer data (Part 1):
    a. After each sample, I converted the ADC values into gravitational units (g) for each axis (x, y, and z). These values were sent byte-by-byte to the serial app, displaying each axis separately for clear, real-time monitoring of movement.
4. For Part 2, I implemented crash detection logic:
    a. I calculated the total acceleration magnitude by combining the x, y, and z values and checked if it met the 2g threshold. If it did, the RED LED turned on, simulating an airbag deployment signal. If not, the LED remained off, allowing continuous monitoring without interruption.
5. Finally, I optimized the main loop:
    a. The main loop starts ADC sampling and sends data to the UAH app every 0.1 seconds. I added low-power mode to conserve energy between samples, ensuring the program remains efficient and responsive throughout its operation.

Note: The demo was used which was very similar to the code I used, not much had to be added or changed. I simply added the z coordinate and added a magnitude to the code.

Formulas:

1. Formula for Accelerometer Interfacing
   a. G(n) = [(3.0 * (n/4095) - 1.5) / 0.3]
   b. (Given in demo)
2. Formula for Magnitude of Acceleration
   a. M = sqrt(x^2 + y^2 + z^2)
   b. (Given in lab requirements)
3. Formula for Angular Deviation
   a. Pitch Angle
      i. Θ = arctan2(x/sqrt(y^2 + z^2))
   b. 2. Roll Angle
      i. Φ = arctan2(y/sqrt(x^2 + z^2))

Flowchart:

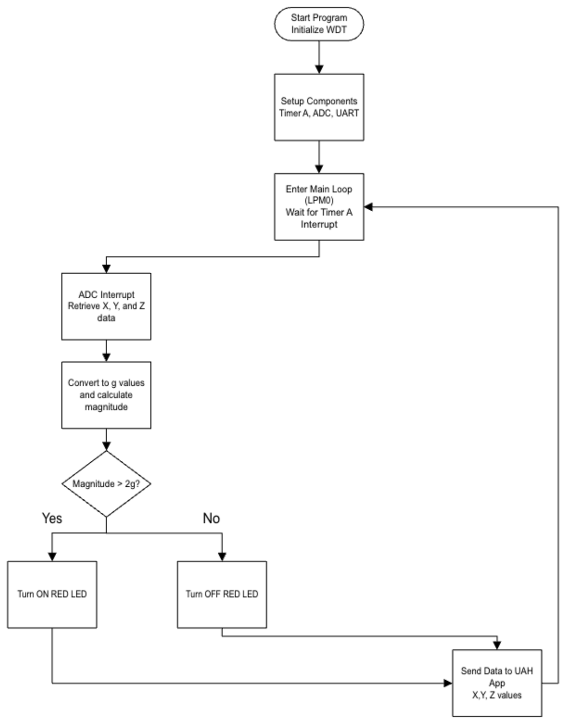
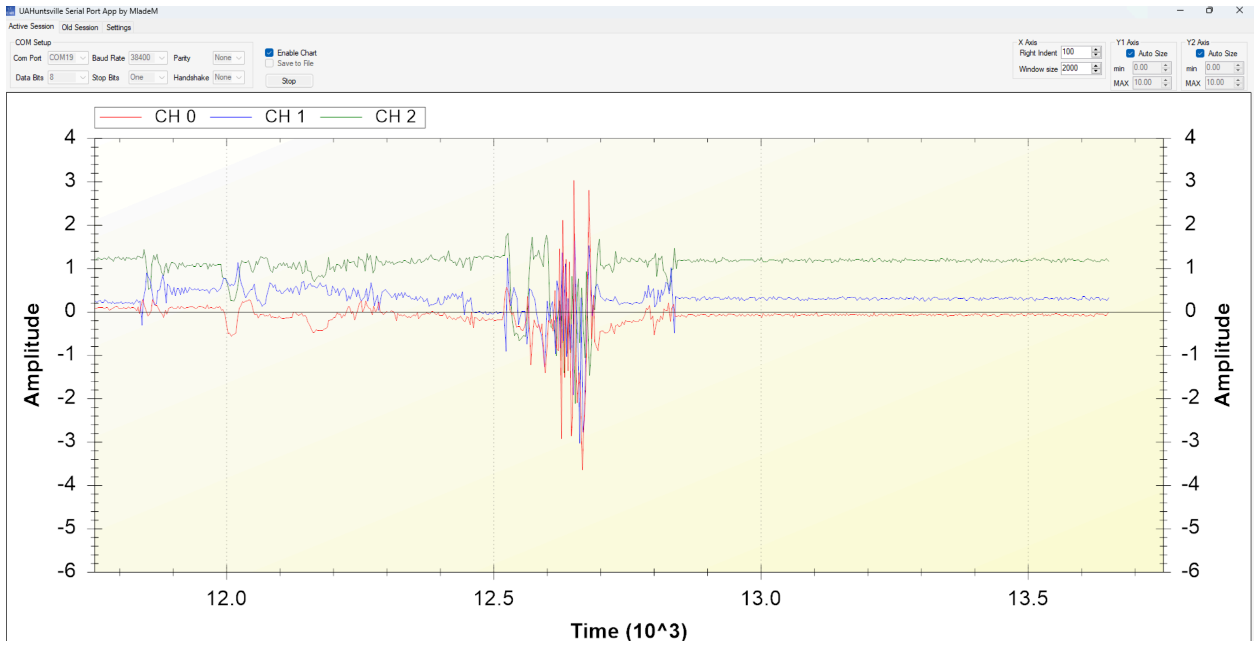

Start Program Initialize WDT
Setup Components Timer A, ADC, UART
Enter Main Loop (LPM0) Wait for Timer A Interrupt
ADC Interrupt Retrieve X, Y, and Z data
Convert to g values and calculate magnitude
Magnitude > 2g?
Yes
No
Turn ON RED LED
Turn OFF RED LED
Send Data to UAH App X,Y, Z values

Figure 1: Program 2 Flowchart

Program Output:


UAHuntsville Serial Port App by MladeM

Active Session | Old Session | Settings

COM Setup: Com Port COM19, Baud Rate 38400, Parity None, Data Bits 8, Stop Bits One, Handshake None

Enable Chart ✓, Save to File ☐

X Axis: Right Indent 100, Window size 2000

Y1 Axis: Auto Size ✓, min 0.00, MAX 10.00

Y2 Axis: Auto Size ✓, min 0.00, MAX 10.00

Figure 2: Program 2 Output

Part 3:

Program 3 Description:

This program generates multiple waveforms using digital-to-analog conversion on the MSP430F5529 microcontroller. By default, it outputs a sine wave at 30 Hz, created using a lookup table for smooth and accurate signal generation. When SW1 is held, the program switches to a saw-tooth waveform, calculated in real-time instead of using the lookup table. Holding down SW2 triples the waveform frequency, making the signal more responsive to user input.

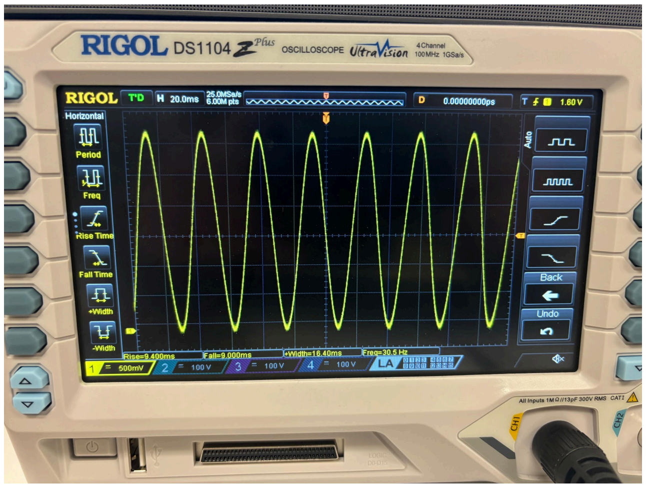
Process:

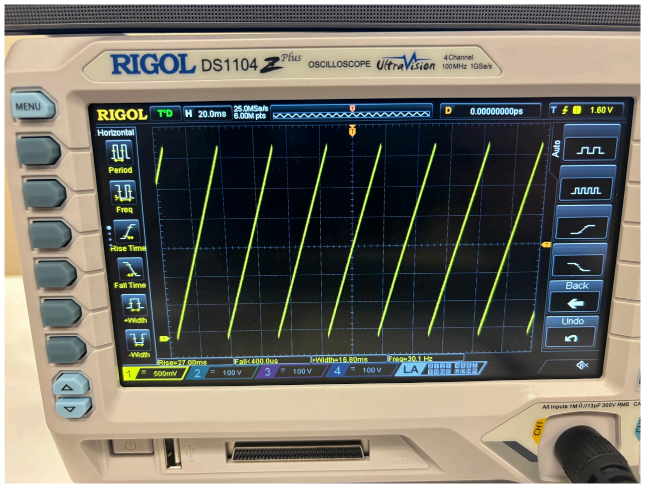
1. First, I set up the PWM configuration for waveform output: I used Timer B on the MSP430, setting it to SMCLK with up mode and a CCR0 value of 135 for a 2 kHz base frequency. This enabled precise control for generating waveforms and controlling the buzzer output on P3.4.

2. Then, I configured the switches for waveform selection: I set up SW1 (P2.1) to select a sawtooth waveform and SW2 (P1.1) to triple the frequency when held. Pull-up resistors were enabled on both switches to ensure stable signal readings.
3. Next, I implemented the Timer B interrupt service routine (ISR): The ISR increments an index for waveform generation. When neither switch is pressed, the ISR outputs values from the sine lookup table, creating a 30 Hz sine wave by default.
4. Then, I added logic for waveform selection:
   a. If SW1 is pressed, the ISR bypasses the sine lookup table, outputting the raw index values to create a sawtooth wave on P3.4. Additionally, it toggles the RED LED on P1.0 to indicate waveform generation.
   b. If SW2 is pressed, the index increments by 2, effectively tripling the frequency and producing a higher-frequency sine wave.
5. Finally, I configured the main loop for efficiency: By entering low-power mode (LPM0) and enabling global interrupts, the program conserves energy while maintaining responsiveness. This approach allows for continuous, real-time waveform generation and switching based on user input.

Note: The demo was used which was very similar to the code I used, not much had to be added or changed.

Program Output:





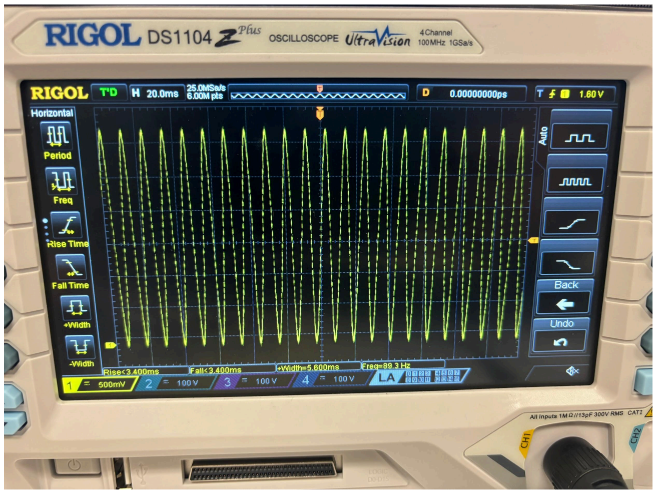


Figure 3: Program 2 Output

Conclusion

In conclusion, this lab provided hands-on experience in interfacing with sensors, real-time data processing, and signal generation using the MSP430 microcontroller. In Part 1, I successfully configured the ADXL335 accelerometer to capture and display x, y, and z-axis accelerations on the UAH serial app, offering real-time feedback on positional changes. Part 2 built on this by implementing crash detection, where the program calculates the total acceleration magnitude, turning on the RED LED to simulate airbag deployment when a threshold is met. This task highlighted practical applications of accelerometers in safety systems. Part 3 focused on digital signal generation, where I used Timer B and a lookup table to create a sine wave at 30 Hz by default. The program allows switching to a sawtooth waveform on-the-fly and dynamically triples the output frequency. This demonstrated the microcontroller's versatility in generating and adjusting waveforms based on user input. Overall, the lab reinforced skills in ADC, UART communication, PWM, and real-time control, essential for embedded systems applications.

Appendix

Table 1: Program 1 with Parts 1 and 2 source code

```
/*------------------------------------------------------------------------------
 * File: Lab10_P1_P2.c
 * Desciption:
 * This program interfaces the MSP430 microcontroller with the ADXL335 accelerometer
 * to capture and process acceleration data along the x, y, and z axes. Real-time
 * acceleration values (in g) are sent via UART to the UAH serial app for display.
 *
 * Part 1:
 * - Samples acceleration values at a rate of 10 Hz.
 * - Converts ADC values to gravitational acceleration (g) for each axis (x, y, z).
 * - Sends the converted values to the UAH serial app, displayed as separate x, y, and z readings.
 *
 * Part 2:
 * - Implements crash detection by calculating the total acceleration magnitude from all axes.
 * - If the calculated magnitude reaches or exceeds 2g, the RED LED is turned on to indicate
 *   a potential airbag deployment.
 * - This part demonstrates threshold-based signaling for safety applications.
 *
 * Input: ACCELEROMETER | ANALOG SIGNAL
 * Output: Xg, Yg, Zg TO UAH SERIAL APP & RED LED ON WHEN MAGNITUDE REACHES 2g | DIGITAL SIGNAL
 * Author: Gianna Foti
 *------------------------------------------------------------------------------*/

#include <msp430.h>
#include <math.h>

volatile long int ADCXval, ADCYval, ADCZval;
volatile float Xper, Yper, Zper;
volatile float magyMagnitude;

void TimerA_setup(void)
{
    TA0CCTL0 = CCIE;                    // Enabled interrupt
    TA0CCR0 = 3276;                     // 3277 / 32768 = .1s for ACLK
    TA0CTL = TASSEL_1 + MC_1;           // ACLK, up mode
    P1DIR |= BIT0;                      //blink bro
}



void ADC_setup(void) {
    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) {
   while (!(UCA0IFG&UCTXIFG));    // Wait for previous character to transmit
   UCA0TXBUF = c;                  // Put character into tx buffer
}

void UART_setup(void) {
   P3SEL |= BIT3 + BIT4;               // 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
}

#define ADC_TO_G(n) ((3.0 * n / 4095 - 1.5) / 0.3) //convert to g
void sendData(void)
{
   int i;

   // Part 1 - get samples from ADC
   Xper = (ADC_TO_G(ADCXval));    // Calculate percentage outputs
   Yper = (ADC_TO_G(ADCYval));    // Calculate percentage outputs
   Zper = (ADC_TO_G(ADCZval));    // Calculate percentage outputs

   // Use character pointers to send one byte at a time
   char *xpointer=(char *)&Xper;
   char *ypointer=(char *)&Yper;
   char *zpointer=(char *)&Zper;

   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]);
   }
   for(i = 0; i < 4; i++)
   {           // Send z percentage - one byte at a time
      UART_putCharacter(zpointer[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) {
   ADC12IFG = 0x00;
   ADCXval = ADC12MEM0;                  // Move results, IFG is cleared
   ADCYval = ADC12MEM1;
   ADCZval = ADC12MEM2;
   __bic_SR_register_on_exit(LPM0_bits); // Exit LPM0
}

#pragma vector = TIMER0_A0_VECTOR
__interrupt void timerA_isr() {
   magyMagnitude = sqrt((Xper * Xper) + (Yper * Yper) + (Zper * Zper));
   if (magyMagnitude > 2) // When the magnitude reaches the critical value of 2G, activate LED1; otherwise, keep LED1
turned off.
   {
      P1OUT |= BIT0;
   }
   else
   {
      P1OUT &= ~BIT0;
   }
   sendData();                        // Send data to serial app
   __bic_SR_register_on_exit(LPM0_bits); // Exit LPM0
}
```

Table 2: Program 3 source code

```
/*------------------------------------------------------------------------------
File:         Lab10_P3.c
Description:  This program generates waveforms using PWM on the MSP430. The
              default waveform is a sine wave generated at 30 Hz using a
              lookup table. The functionality includes:
              - Generating a sine waveform when no switches are pressed.
```

```
            - Generating a sawtooth waveform on the fly when SW1 is held.
            - Tripling the waveform frequency to 90 Hz when SW2 is held.
            The output is routed through the buzzer on P3.4, with visual
            indicators using the RED LED on P1.0 (for sound state) and
            potentially the GREEN LED on P4.7 (not implemented in this version).
Input:      P2.1 - SW1 (Hold for Sawtooth waveform) | P1.1 - SW2 (Hold to
            triple frequency)
Output:     P1.0 - RED LED (indicates sound on) | P4.7 - GREEN LED
            (not used in this implementation) | P3.4 - BUZZER (PWM output)
Author:     Gianna Foti
------------------------------------------------------------------------------*/

#include <msp430.h>            // Include MSP430-specific header
#include "sine_lut_256.h"      // Include the sine lookup table header

#define Switch1 (P2IN & BIT1)  // Define macro for reading Switch 1
#define Switch2 (P1IN & BIT1)  // Define macro for reading Switch 2

unsigned char index = 0;       // Variable to keep track of the sine wave index

int main(void)
{
   WDTCTL = WDTPW | WDTHOLD;  // Stop the watchdog timer to prevent resets

   // Timer B Configuration
   TB0CCTL0 = CCIE;           // Enable Timer B interrupt
   TB0CTL = TBSSEL_2 + MC_1;   // Use SMCLK as the timer source, set to up mode
   TB0CCR0 = 135;             // Set CCR0 for the desired frequency (2kHz)

   P3DIR = 0xFF;              // Set all pins on Port 3 as outputs
   P3OUT = 0;                 // Initialize Port 3 output to 0
   P1DIR = BIT0;              // Set P1.0 as output for the RED LED

   buttons();                 // Call the button configuration function

   __bis_SR_register(LPM0_bits + GIE); // Enter low power mode with global interrupts enabled
   return 0;                  // Return from main (never reached in LPM)
}

// Function to configure switches
void buttons()
{
   // Configure Switch 1
   P2DIR &= ~BIT1;            // Set P2.1 as input
   P2REN |= BIT1;             // Enable pull-up/pull-down resistor on P2.1
   P2OUT |= BIT1;             // Set pull-up resistor on P2.1

   // Configure Switch 2
   P1DIR &= ~BIT1;            // Set P1.1 as input
   P1REN |= BIT1;             // Enable pull-up/pull-down resistor on P1.1
   P1OUT |= BIT1;             // Set pull-up resistor on P1.1
}
```

```
// Timer B interrupt service routine
#pragma vector=TIMERB0_VECTOR
__interrupt void timerISR2(void)
{
    index += 1;                  // Increment the sine wave index

    // Check if Switch 2 is pressed and Switch 1 is not pressed
    if ((Switch2 == 0) && (!(Switch1 == 0)))
    {
        index += 2;              // Increase index by 2 (modify frequency)
        P3OUT = lut256[index];  // Output sine value from lookup table
    }
    // Check if Switch 1 is pressed and Switch 2 is not pressed
    else if ((Switch1 == 0) && (!(Switch2 == 0)))
    {
        P1OUT ^= BIT0;          // Toggle the RED LED
        P3OUT = index;          // Output current index value to Port 3
    }
    else
    {
        P3OUT = lut256[index];  // Default: Output sine value from lookup table
    }
}
```