CPE 325: Intro to Embedded Computer System

Final Project

Reaction Time Tester

Submitted by: Hannah Kelley

Date of Experiment: November 10, 2025

Report Deadline: December 3, 2025

Demonstration Deadline: December 1, 2025

Project Description

The goal of this project is to design and implement a reaction time measurement system using the MSP430 microcontroller. The system evaluates a user's reflexes by measuring how quickly they respond to a visual stimulus generated by onboard LEDs. This project demonstrates the use of hardware interrupts, timers, and UART serial communication, integrating digital input/output with precise timing control.

When the program begins, the user receives instructions through a serial terminal such as PuTTY or MobaXTerm via UART. Pressing Switch 1 starts the test. After a random delay of approximately 1–3 seconds, LED1 illuminates, signaling the user to react. The user then presses Switch 2, and their reaction time—reported in both clock cycles and milliseconds—is calculated and sent back over UART. After the result is displayed, the user may start another round by pressing Switch 1 again.

Theory

Topic 1: Parallel Ports

The MSP430 includes several 8-bit parallel I/O ports used for digital input and output. Each port is divided into individual pins that can be configured independently as either inputs or outputs using direction registers (PxDIR). Data written to the output registers (PxOUT) appears on the pins when they are configured as outputs, while the input registers (PxIN) allow the program to read logic levels present on pins configured as inputs. Port pins can also be assigned to peripheral functions such as timers, communication modules, and interrupts, depending on the device configuration. Function selection registers (PxSEL) determine whether a pin is used for general-purpose I/O or dedicated module operation. Many pins also support internal pull-up or pull-down resistors that can be enabled through the PxREN register. Parallel ports allow fast and deterministic communication with external circuits—such as LEDs, switches, sensors, and digital logic—making them fundamental to most embedded applications on the MSP430.

Topic 2: Debouncing

When a mechanical switch is pressed or released, the internal contacts do not transition cleanly from one state to another. Instead, they physically vibrate or "bounce" for a short period of time, typically a few milliseconds. This bouncing produces multiple rapid on-off transitions instead of a single, clean signal change. These unintended transitions can cause a digital circuit to register multiple button presses from a single physical action. To mitigate this problem debouncing is used. Debouncing ensures that only one logical transition is recorded for each physical button press or release. Debouncing can be done at the hardware or software level. On the hardware level, special circuits are employed to get rid of the messy signal bouncing generates. On the software level, short delays are implemented to ensure a button was pressed or released before continuing with the expected behavior.

Topic 3: Software Delay

In this project, debouncing was implemented using a software delay. When the program detects a change in the switch state, it does not immediately register the input as valid. Instead, it introduces a short pause in execution—20 milliseconds in this case—to allow the mechanical contacts within the switch to settle. After the delay, the switch state is checked again. If the switch remains in the same state, the input is confirmed as a valid press or release; otherwise, the change is ignored. This method prevents the program from misinterpreting the rapid on-off transitions caused by bouncing as multiple button presses.

Topic 4: Interrupts

Interrupts are an embedded system's way of reacting to certain events that occur in the system or in a peripheral device. In order to receive interrupt requests, the MSP430 must be configured to allow requests from the pin or peripheral module being used as the interrupt source. Both the specific interrupt enable bit and the Global Interrupt Enable (GIE) bit in the Status Register must be set before an interrupt can be serviced. When an interrupt occurs, the MSP430 automatically saves the current program counter and status register on the stack and jumps to the Interrupt Service Routine (ISR) that corresponds to the pending interrupt. The ISR contains the code that handles the event and typically clears the interrupt flag so the same request does not trigger repeatedly. After the ISR completes, the RETI instruction restores the saved state and normal execution continues where it left off. If multiple interrupts are pending at the same time, the MSP430 uses the Interrupt Vector Table (IVT) to determine which interrupt is handled first. The IVT contains the start addresses and fixed hardware priorities of every interrupt the MSP430 can service, with lower vector addresses having higher priority. Using interrupts increases efficiency and reduces power consumption because the CPU does not need to constantly poll devices. Instead, it can remain in a low-power state and wake only when an event requires attention.

Topic 5: Timers

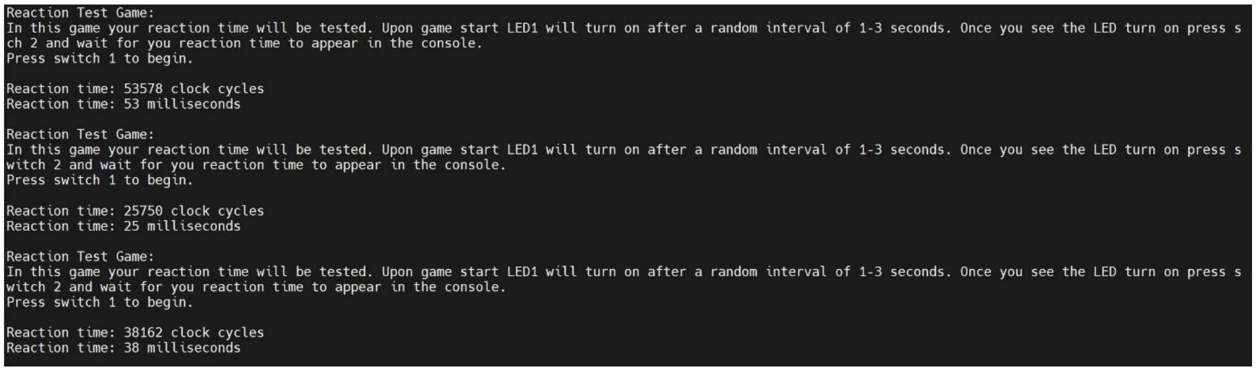
The MSP430 includes hardware timers (such as Timer_A and Timer_B) that provide precise timing and event control without requiring constant CPU involvement. Each timer operates from a configurable clock source—such as ACLK, SMCLK, or an external clock—and counts up at a fixed rate based on the selected clock prescaler. Timers contain one or more capture/compare registers, which allow them to perform several functions such as periodic interrupts, input timestamping, and outputting periodic signals such as pulse width modulation. When a timer event occurs, it sets a flag that can trigger an interrupt if enabled. The CPU then executes the corresponding ISR to handle the event. Because timers continue running even while the CPU is in low-power mode, they are essential for accurate timekeeping, periodic task scheduling, and generating real-time outputs in embedded applications.

Topic 6: Serial Communication via UART

Serial communication is the means of transferring data between two devices over a single line of communication a single bit at a time. This project uses UART (Universal Asynchronous Receiver-Transmitter), an asynchronous protocol that does not require the sender and receiver to have a shared clock. Instead, both devices must be configured to operate at the same baud rate—the rate at which bits are transmitted per second—to ensure proper timing and interpretation of data. Data is sent from the transfer shift register at the sending device to the receive shift register at the receiving device. Caution must be given when programming to ensure that the data sent is in the format expected by both devices.

Results & Observation

Program Output:



```
Reaction Test Game:
In this game your reaction time will be tested. Upon game start LED1 will turn on after a random interval of 1-3 seconds. Once you see the LED turn on press s
ch 2 and wait for you reaction time to appear in the console.
Press switch 1 to begin.

Reaction time: 53578 clock cycles
Reaction time: 53 milliseconds

Reaction Test Game:
In this game your reaction time will be tested. Upon game start LED1 will turn on after a random interval of 1-3 seconds. Once you see the LED turn on press s
witch 2 and wait for you reaction time to appear in the console.
Press switch 1 to begin.

Reaction time: 25750 clock cycles
Reaction time: 25 milliseconds

Reaction Test Game:
In this game your reaction time will be tested. Upon game start LED1 will turn on after a random interval of 1-3 seconds. Once you see the LED turn on press s
witch 2 and wait for you reaction time to appear in the console.
Press switch 1 to begin.

Reaction time: 38162 clock cycles
Reaction time: 38 milliseconds
```

Figure 1: Program Output in MobaXTerm

Report Questions:

1. **How do you set up and run the project?**
   In CCS, copy the code given in Table 1 of the appendix into a .c file. No settings were changed from the normal lab settings as seen in Tutorial 1 in Canvas. No additional peripherals need to be added and no pins need to be jumped. This project uses a UART connection via the power cord connected to the MSP430 board when flashing the program. In MobaXTerm or PuTTy set the port to the COM port shown as the “MSP430 Application UART” port in devices manager. Make sure to set the baud rate to 115,200 bps, parity and flow control to none, stop bits to 1, and data bits to 8. Once the user’s UART interface is set up the program can be built and flashed to the board.

Program Flowchart:

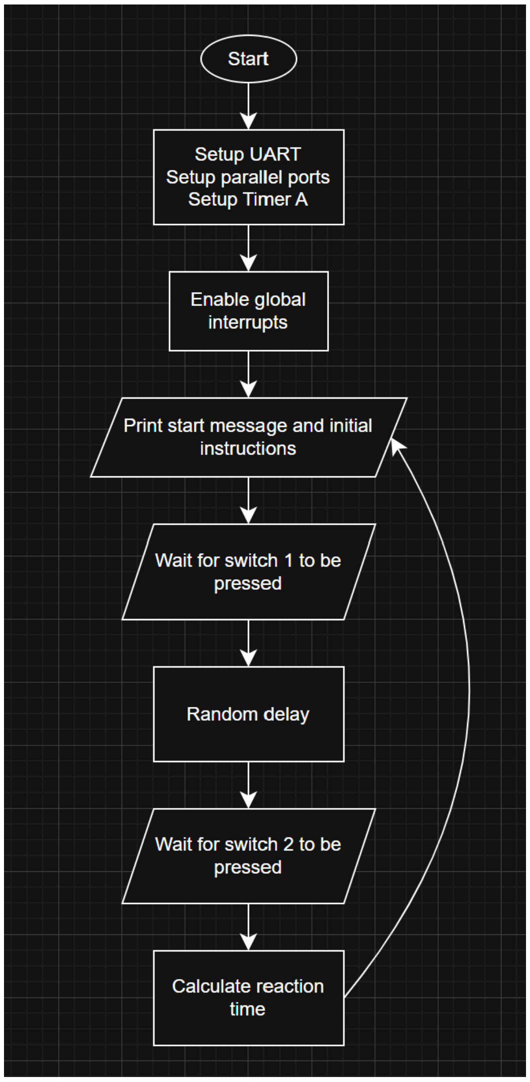


Figure 2: Program Flowchart

Conclusion

One of the main challenges in this project was configuring the MSP430 timer in capture mode. While timers had been used previously in simpler configurations, the capture functionality had not been discussed in depth in lectures or previous labs. As a result, understanding how the capture register stored timestamps, how capture events were triggered, and how to correctly interpret the values required extensive reference to the MSP430 documentation and experimentation. It took time to determine the correct clock source settings, input edge selection, and interrupt configuration needed to ensure that the captured times were accurate and consistent across multiple trials.

Another difficulty involved processing the captured values to generate a usable reaction time measurement. Because the timer counts continuously and wraps around once it reaches its maximum value, additional logic was needed to correctly handle situations in which the LED event and the button press occurred on opposite sides of this overflow point. Debugging this required careful observation of the timer behavior, verification through UART output, and iterative testing to ensure that the conversion to clock cycles and milliseconds was correct. Although not initially straightforward, successfully implementing capture mode provided a much deeper understanding of how hardware timers operate at a low level and how embedded systems generate and record precise timing information.

In conclusion, the reaction time measurement system successfully demonstrated the use of the MSP430’s hardware interrupts, timers, and UART communication to create a functional reflex-testing application. The system accurately measured user response time and provided results in both clock cycles and milliseconds. Through this project, we gained experience in integrating digital inputs, handling asynchronous interrupt events, configuring timers for precise measurement, and implementing serial communication. Although limited to a single LED stimulus and terminal-based output, the system could be expanded in the future with additional feedback methods or performance tracking features. Overall, the project met its objectives and strengthened our understanding of embedded system design and low-level microcontroller programming.

Appendix

Table 1: Program Source Code

```
/*--------------------------------------------------------------------------
 * File:        FinalProject.c
 * Description: reaction time game, press SW1 to start, then wait for LED1
to light up before pressing SW2. Reaction time is automatically calculated
and displayed via UART
 *
 * Board:       5529
 * Input:       user switch presses
 * Output:      LED1 and reaction time via UART
 * Author:      Hannah Kelley
 * Date:        November 3, 2025
 *-------------------------------------------------------------------------*/
#include <msp430.h>
#include <stdint.h>
#include <stdlib.h> // for rand (returns a max of 32767
#include <stdio.h>
#include <string.h>

#define SW1 (P2IN&BIT1)
#define SW2 (P1IN&BIT1)

void parallelPortSetup(void);
void UART_setup(void);
void TimerASetup(void);
void printStartMessage(void);
void UARTA1_putchar(char c);
char UART_getChar(void);
void PrintString(char * msg);

volatile unsigned long delay;
volatile long randDelay;
volatile unsigned long reactionCCs;
volatile unsigned long reactionMSs;
volatile int gameActive = 0; // 0 = waiting for SW1, 1= waiting for SW2
volatile uint16_t timerOverflows = 0;

int main() {
    WDTCTL = WDTPW | WDTHOLD; // hold WDT

    UART_setup();
    parallelPortSetup();
    TimerASetup();

    __enable_interrupt();

    while (1) {
        printStartMessage();
        while (SW1); // wait for SW1 to be pressed
        for (randDelay = (rand() % 16383) * 100; randDelay > 0; randDelay--
);
        TA0CTL = TASSEL_2 | MC_2 | TACLR;
        gameActive = 1;
        P1OUT |= BIT0; // turn on LED1
        while (gameActive);
    }
}

void parallelPortSetup(void) {
    // SW1 config
    P2DIR &= ~BIT1;     // P2.1 input
    P2REN |= BIT1;      // enable pullup register
    P2OUT |= BIT1;      // set pullup register

    // SW2 config
    P1DIR &= ~BIT1;     // P1.1 input
    P1REN |= BIT1;      // enable pullup register
    P1OUT |= BIT1;      // set pullup register
    P1IE |= BIT1;
    P1IES |= BIT1;
    P1IFG &= ~BIT1;

    // LED1 config
    P1DIR |= BIT0;      // P1.0 output
    P1OUT &= ~BIT0;     // LED off to start
}

void UART_setup(void) { // code previously used for lab 8
    // Configure UART communication on USCI_A1
    P4SEL |= BIT4 + BIT5;   // Set P4.4 and P4.5 for UART
    UCA1CTL1 |= UCSWRST;    // Put state machine in reset during setup
    UCA1CTL0 = 0;           // Default UART configuration
    UCA1CTL1 |= UCSSEL_2;   // Select SMCLK as UART clock source
    UCA1BR0 = 0x09;         // Set baud rate to 115200 (SMCLK/115200)
    UCA1BR1 = 0x00;         // Baud rate divider
    UCA1MCTL = 0x02;        // Modulation settings
    UCA1CTL1 &= ~UCSWRST;   // Initialize USCI state machine
}

void TimerASetup(void) {
    TA0CTL = MC_0; // stop timer
    TA0CTL = TASSEL_2 | MC_0 | TACLR; // SMCLK in continuous mode
}

void printStartMessage(void) {
    PrintString("Reaction Test Game: \n\r");
    PrintString("In this game your reaction time will be tested. "
            "Upon game start LED1 will turn on after a random interval of 1-
3 seconds. "
            "Once you see the LED turn on press switch 2 and wait for you
reaction time to appear "
            "in the console. \n\r");
    PrintString("Press switch 1 to begin.\n\r\n\r");
}

void UARTA1_putchar(char c) { // code previously used for lab 8
    while (!(UCA1IFG & UCTXIFG));   // Wait until TX buffer is ready
    UCA1TXBUF = c;                  // Transmit character via UART
}

char UART_getChar() { // code previously used for lab 8
```

```
    while (!(UCA1IFG & UCRXIFG));   // Wait until RX buffer has data
    return UCA1RXBUF;               // Return received character
}

void PrintString(char * msg) { // code previously used for lab 8
    // Print a string character-by-character via UART
    register int i = 0;
    for (i = 0; i < strlen(msg); i++) {
        UARTA1_putchar(msg[i]); // Send each character
    }
}

#pragma vector = PORT1_VECTOR
__interrupt void Port1_ISR(void) {
    if (P1IFG & BIT1) {

        TA0CTL = MC_0;

        reactionCCs = (unsigned long)TA0R + ((unsigned long)timerOverflows *
65536);
        reactionMSs = reactionCCs / 1000;
        timerOverflows = 0; // reset for next round

        P1OUT &= ~BIT0;

        char buffer[48];
        sprintf(buffer, "Reaction time: %lu clock cycles\r\n", reactionCCs);
        PrintString(buffer);
        sprintf(buffer, "Reaction time: %lu milliseconds\r\n\r\n",
reactionMSs);
        PrintString(buffer);

        gameActive = 0;
        P1IFG &= ~BIT1;
    }
}

#pragma vector = TIMER_A0_VECTOR
__interrupt void TimerA0_ISR(void) {
    switch (__even_in_range(TA0IV, 14)) {
    case 10:
        timerOverflows++;
        break;
    default:
        break;
    }
}
```