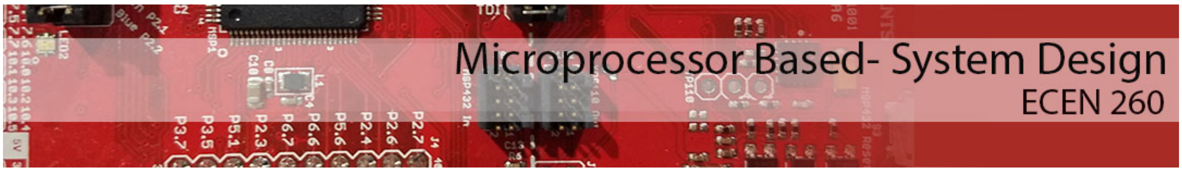

ECEN 260 - Final Project

Microwave Popcorn Timer

Never burn your popcorn again!

Daniel Loveless

Instructor: Brother Allred
December 18, 2021

Contents

List of Tables

List of Figures

1 Project Overview

This report describes a final project for ECEN 260 at Brigham Young University–Idaho. The Microwave Popcorn Timer encompasses various concepts of Microprocessor-Based System Design, for the purpose of ensuring perfectly cooked popcorn by helping the user know exactly when to stop cooking.

1.1 Objectives

The objective of this project is to incorporate the following concepts into a fully functional prototype product:

- Digital I/O
- Analog to Digital Conversion (ADC)
- Hardware interrupts
- UART/I2C/SPI communication protocol
- Display drivers

As you review this document, you will come to see how each of these concepts have been put to use in the Microwave Popcorn Timer to achieve the mission of ensuring you never have to taste burnt popcorn again!

1.2 Description

The Microwave Popcorn Timer operates using the digital input of a sound sensor module, given a certain volume threshold, to time the interval between 'pop's of exploding popcorn and alert the user once that interval has surpassed a specified duration.

The cutoff threshold for the pop interval can be adjusted using the attached potentiometer to accommodate for varying cook times between different brands of popcorn. The output is displayed to the user via the Graphics LCD, showing both the current status of the adjustable cutoff interval and the amount of time since the last registered 'pop' in the format: "TIME / INTERVAL s", plus a cute '*' symbol and flashing on-board LED's to indicate when a pop has been registered.

The device is equipped with a PCD8544/Nokia 5110 GLCD [1], a piezo buzzer, and a 10 kΩ potentiometer all driven by an MSP432 microcontroller on the TI MSP-EXP432P401R LaunchPad Development Kit [2]. The following, Section 2, details how to set up these devices to operate together and explains their operating principles. Section 3 explains their operating principles in more detail, highlighting the aforementioned microprocessor-based system design concepts. Please refer to Section 4 for more information on how these operating principles are implemented on the MSP432 via C code.

2 Schematics

By following the wiring diagrams and schematics below, you can conntect the GLCD display, sound sensor module, piezzo buzzer, and 10 kΩ potentiometer to match the specifications for this project and thus make your very own Microwave Popcorn Timer!

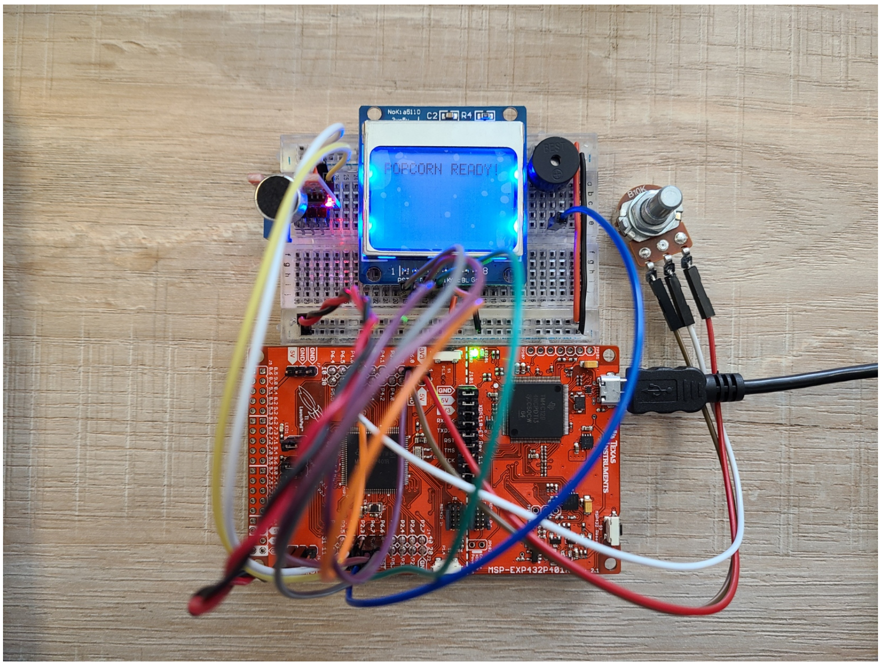

Figure 1: Image of fully assembled Microwave Popcorn Timer unit

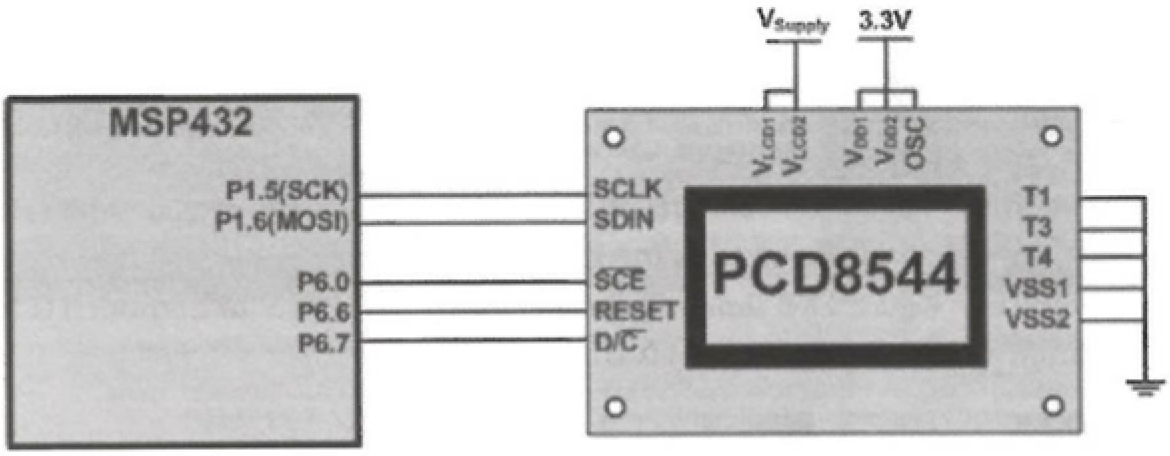

Figure 2: Schematic diagram for connecting the GLCD display

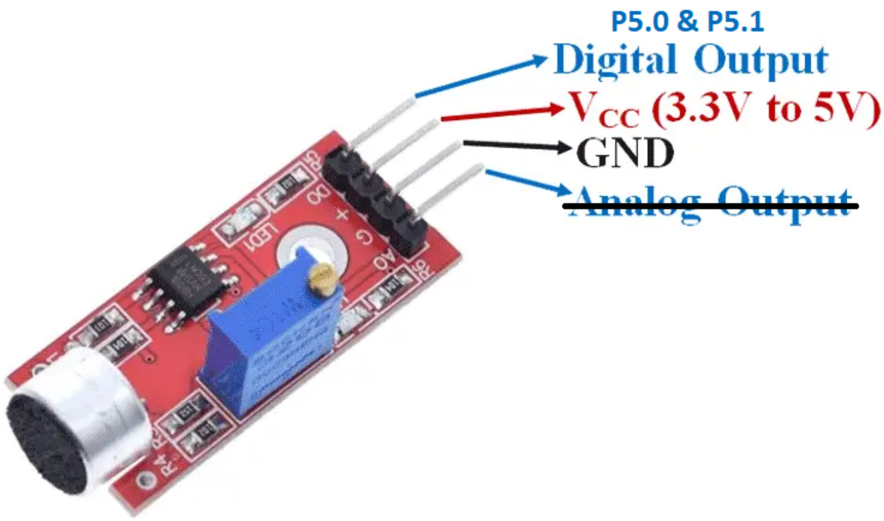

Figure 3: Schematic diagram for connecting the sound sensor module

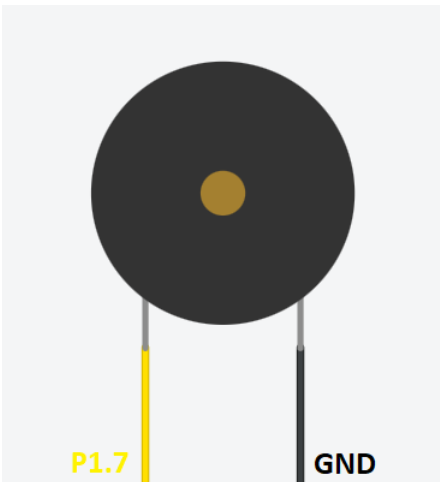

Figure 4: Schematic diagram for connecting the piezo buzzer

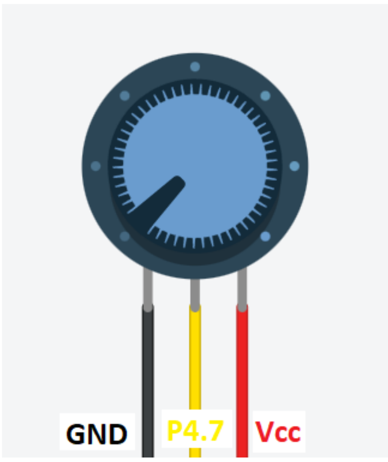

Figure 5: Schematic diagram for connecting the $10\,\mathrm{k}\Omega$ potentiometer

3 Specifications & Theory of Operation

3.1 Digital I/O & Hardware Interrupts

For accessing digital I/O, the MSP432 utilizes registers in memory that are tied to certain ports which have connected to them other modules of the chip and physical pins on the board. Modifying these registers allows us to change the settings and configuration of the desired ports and associated I/O.

By modifying these registers, pins can be configured as inputs or outputs and connected to on-board pull-up/down resistors, as needed. They can also be designated as analog input channels, and hardware interrupts can also be enabled for entire ports and their specific pins by altering the contents of their associated registers. See Chapter 5.2 and Chapter 6.4 of Programmable Microcontrollers: Applications on the MSP432 LaunchPad [3] for more information on digital I/O registers and port interrupts, respectively.

The C programming language conveniently provides a customizable data structure known as 'structs' that define registers under the names of their associated I/O and wrap them all together under the common name of their associated ports, similar to a folder system. See Section 4.1 for examples of how they are used in this project to configure the I/O.

All the peripheral devices used in this project are accessed this way, albeit with more complicated steps on top of them that will be discussed below. The microphone, buzzer, and LED's however, are purley digital I/O which utilize this method, with the microphone being able trigger interrupts resulting in events defined by code. The LED's are turned on and off by the rising and falling-edge interrupts from the microphone, respectively, to indicate that a 'pop' has been registered and an '*' is also sent to the display.

3.2 Analog to Digital Conversion

The MSP432 is equipped with a high-resolution analog-to-digital converter capable of 1 Msps 14-bit conversions. The on-board ADC14 Module is configured using structs to modify the memory registers responsible for holding its settings. It utilizes a method known as successive approximation to arrive at a digital representation of an analog signal. It does this by comparing the voltages of the analog signal and a generated value representing each bit of a 14-bit binary number to see which is greater, starting from the most significant bit (MSB) to the least significant bit (LSB) until it has filled out the entire number. See Chapter 8.4 [2] to learn more. That conversion can then be stored in a designated memory location that can be accessed by our code.

The 10 kΩ potentiometer is the source of our analog signal for this project and is responsible for setting the cutoff interval of the Microwave Popcorn Timer. This allows the user to be able to change the value to match the instructions for their specific brand of microwave popcorn.

3.3 Timer Module

The Microwave Popcorn Timer wouldn't be much of a timer without a timer to keep track of time. Enter: the Timer_A0 module! This is one of many timer modules on-board the MSP432 which takes a clock source specified in the module's control register, and uses it to iterate a counter that can be configured in stop, up, up/down, or continuous modes. The Timer_A0 module is configured in up/down mode for this project such that it generates an interrupt upon reaching a predefined value before changing direction to decrement until reaching zero and beginning the process again. That maximum value, given the clock source frequency of 32 768 Hz, can be set such that it generates an interrupt at a regular interval. For this application, a period of 0.1s was chosen, as the display shows time with an accuracy of one decimal point. This periodic interrupt serves as the source of our software timer driving the Microwave Popcorn Timer. Refer to Section 4.3 to see the software implementation of this timer, and Sec. 7.7.1 of the text [3] for information regarding the Timer_A module.

3.4 Graphics LCD Driver

A key part of this project is the user interface which includes displaying characters to the PCD8544 GLCD. The PCD8544 is comprised of 48 x 84 monochrome pixels with groupings of 6 x 8 pixels forming banks large enough to hold a character. There are 14 x 6 banks which are ordered from left to right, top to bottom.

When writing to the display, you are writing to the Display DRAM that holds the value of each pixel which can be addressed 8-bits at a time. The pixels in each bank are arranged into 6 vertical bytes of information, each bit representing one pixel, so top to bottom (pixels 0-7), left to right (columns 0-5) forms one bank and thus one character.

As such, each character has its own unique pixel layout that can be represented by 6 hexadecimal digits. Section 4.4.1 includes a font table that contains the unique 6-byte values for each character that can be written to a bank on the display including all 26 uppercase letters of the alphabet as well as numbers and some special characters. A visual representation of how these values are generated can be seen in Fig 6 More information about the PCD8544 can be found in its data sheet [1].

This project uses the SPI communication protocol of the MSP432 microcontroller on the TI MSP-EXP432P401R Launchpad [2] by modifying specific eUSCI registers to configure the display, as described in Section 9.3.1 of [3].

3.4.1 Font Table

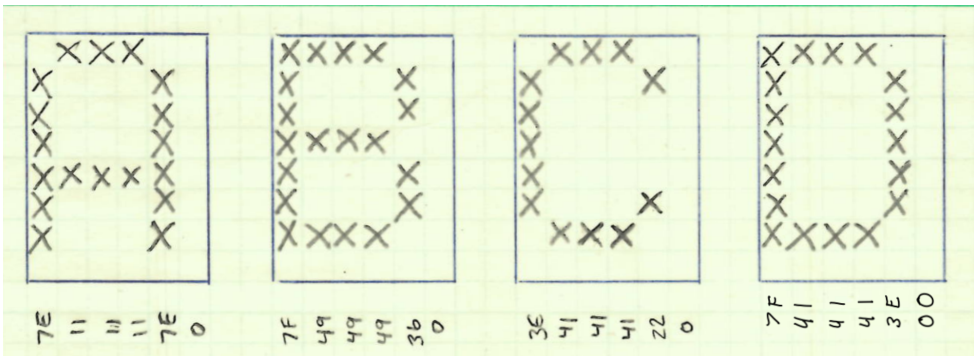

Figure 6: Visual representation of hexadecimal font table

4 Code

Below can be found the code for this project. Take note of the following subsections to gain an understanding of the different methodologies used to program the features of this application.

4.1 Digital I/O & Hardware Interrupts

In the programming for this project, I/O ports are configured by accessing their assigned memory locations. These registers of memory are called with structs that point to specific registers associated with the port. For example, LED1 on the board is tied to P1.0. To configure it as an output, the struct "P1− >DIR" bit masked with the single bit for LED1 (BIT0 out of the 8-bits of the memory register assigned to Port 1) is set to 1 (HIGH) using the '| =' operation. For an explanation of the overarching concepts behind I/O ports, see Section 3.1 and the cited documentation.

Interrupts can also be enabled for specific ports and pins using this same method. The Interrupt Enable register for Port 5 is accessed via the "P5− >IE" struct in the code below. Interrupts must also be enabled for the port as a whole via the Nested Vectored Interrupt Controller. For more information about the NVIC, see Chapter 6.3 of Programmable Microcontrollers: Applications on the MSP432 LaunchPad [3]

Those interrupts then must be handled by a designated block of code, in this case the function, "PORT5_IRQHandler()," is responsible for flashing an LED, modifying the global variable, "pop," which indicates whether a 'pop' has been registered, and resetting the time whenever an interrupt from the microphone is triggered. If you look closely, there are two interrupts, one for the rising edge of the mic signal and one for the falling edge to turn the LED on and back off with one 'pop' event.

4.1.1 Example Code

Example of digital I/O port configuration using structs and implementation of hardware interrupts as well as handling interrupt requests.

```
/*
 * Handle microphone interrupts
 */
void PORT5_IRQHandler(void){
    if(P5->IFG & MIC_RE){    // Rising-edge interrupt
        P5->IFG &= ~MIC_RE;  // Clear interrupt flag
        P2->OUT |= LED2;     // Turn on LED2
        pop = 1;             // Register a 'pop'
    }
    if(P5->IFG & MIC_FE){    // Falling-edge interrupt
        P5->IFG &= ~MIC_FE;  // Clear interrupt flag

        // Debounce signal
        int i;
        for(i=0;i<600;i++);

        time = 0;            // Reset time when a pop is registered
```

```c
        P2->OUT &= ~LED2;    // Turn off LED2
    }
}

/*
 * Configure Ports
 */
void ports_init(void){
    // Configure P1
    P1->DIR |=   LED1 | BUZZ;         // Set LED1 and buzzer as outputs
    P1->OUT &= ~(LED1 | BUZZ);        // Initialize LED1 and buzzer to off

    // Configure P2
    P2->DIR |=  LED2;                 // Set LED2 as output
    P2->OUT &= ~LED2;                 // Initialize to off

    // Configure P5
    P5->DIR &= ~(MIC_RE | MIC_FE);    // Set MIC as input
    P5->IE  |=  MIC_RE | MIC_FE;      // Enable interrupts for MIC
    P5->IES &= ~MIC_RE;               // Rising edge interrupt
    P5->IES |=  MIC_FE;               // Falling-edge interrupt

    // Port-level Interrupt Enable for Port 5
    NVIC->ISER[1] |= 0x00000080;
    __enable_interrupts();
}
```

4.2 Analog to Digital Conversion

In this program, the analog value retrieved from the 10 kΩ potentiomter is converted to digital and then saved into the global float variable, "cutoff," declared toward the top of "main.c" in Section 4.5. This sampling occurs each time "sample_ADC()" is called by the interrupt handler for the Timer_A0 module (Sec 4.3.1). That value is later rounded to one decimal place and presented on the display (Sec 4.4) as the 2nd value the user sees, being the duration the device will wait without hearing a 'pop' before alerting the user that the popcorn is done. This value can be adjusted by the user via the potentiometer.

4.2.1 Example Code

Example of using structs at the register level to configure the ADC14 Module, perform an ADC conversion, and access the resulting data.

```c
/*
 * Sample ADC14 and perform Analog-to-Digital Conversion (ADC)
 */
int sample_ADC(void){
    int result;

    ADC14->CTL0 |= 1;         // Start a conversion
    while (!ADC14->IFGR0);    // Wait until conversion is complete
    result = ADC14->MEM[0];   // Read conversion result
```

```c

    return result;
}

/*
 * Configure ADC14
 */
void ADC_init(void){
    // -> Configure P4.7 for A6 (analog input channel 6)
    P6->SEL0 |= POT;
    P6->SEL1 |= POT;
    // -> Configure CTL0
    ADC14->CTL0  = 0x00000010;  // Power on ADC and disable during config
    ADC14->CTL0 |= 0x04080310;  // Configure CTL0 according to instructions
    // -> Configure CTL1
    ADC14->CTL1  = 0x00000020;  // 12-bit resolution (see instructions)
    ADC14->CTL1 |= 0x00000000;  // Configure for memory register 0
    // -> Configure MCTL[0] (memory register 0) to receive input channel 6
    ADC14->MCTL[0] = 0x06;      // A6 input (P4.7), single ended, Vref = AVCC
    // Enable ADC14
    ADC14->CTL0 |= 0x02;        // Enable ADC14 after configuration
}
```

4.3 Timer Module

In the function "timer_init()" below, the control registers of the clock system are configured to provide the timer module with a clock source and the control register for the Timer_A0 module is accessed to configure the timer in up/down mode with interrupts enabled occurring at a period of 0.1s. The function "TA0_N_IRQHandler()" then handles those periodic interrupts, utilizing their regular interval to increment a global float variable called "time," sample the ADC to update the cutoff variable, blink an indicator LED, and check for the condition that 'time' has exceeded the cutoff value, triggering an alarm.

4.3.1 Example Code

Example of using clock system and timer module control registers to configure a timer for use in a program and handling the interrupts generated by said timer.

```c
/*
 * Handle timer interrupts
 */
void TA0_N_IRQHandler(void){
    TIMER_A0->CTL &= ~TIMER_A_CTL_IFG; // Clear interrupt flag
    if(!end){
        P1->OUT ^= LED1;               // Toggle blinking LED (P1.0)

        cutoff = sample_ADC() / 819.4; // (0, 4096) -> (0, 5)
        time += 0.2;                   // Increment time
        display_time();                // Update display

        if(time >= cutoff){            // Alarm condition
```

```
            time = 0;                           // Reset time to prevent double
    activation
            end = 1;                            // Meet end condition
            trigger_alarm();                    // Sound alarm
        }
    }
}

/*
 * Configure Timer_A0 in up/down mode with predefined period
 * using ACLK (supplied by 32768 Hz LFTX)
 */
void timer_init(void){
    PJ->SEL0 |= BIT0 | BIT1; // Needed for clock/timer

    // Configure clock system
    CS->KEY = CS_KEY_VAL;
    CS->CTL2 |= CS_CTL2_LFXT_EN;
    while(CS->IFG & CS_IFG_LFXTIFG)
        CS->CLRIFG |= CS_CLRIFG_CLR_LFXTIFG;
    CS->CTL1 |= CS_CTL1_SELA_0;
    CS->CLKEN |= CS_CLKEN_REFOFSEL;
    CS->CTL1 &= ~(CS_CTL1_SELS_MASK | CS_CTL1_DIVS_MASK);
    CS ->CTL1 |= CS_CTL1_SELS_2;
    CS->KEY = 0;

    // Configure Timer_A0
    TIMER_A0->CCR[0] = TIMER_PERIOD;
    TIMER_A0->CTL = TIMER_A_CTL_TASSEL_1 | TIMER_A_CTL_MC_3 | TIMER_A_CTL_CLR
    | TIMER_A_CTL_IE;

    NVIC->ISER[0] = 1 << ((TA0_N_IRQn) & 31);
    __enable_irq();
}
```

4.4 Graphics LCD Driver

In Section 4.4.2 are listed several functions pertaining to the configuration and driving of the GLCD for this project. Foremost are the functions responsible for configuring the eUSCI on-board the MSP432 for the SPI communication protocol to be able to communicate with the display. Then, the display is configured by accessing its command register via the SPI communication path just established. Once configured, the "GLCD_setCursor()" and "GLCD_data_write()" functions form the driver for this display which are put together in the "GLCD_clear()" and "GLCD_putchar()" functions which are then used throughout the program to present information on the display. Such instances might include the "print_float()," "display_time()," "print_message()," and "trigger_alarm()" functions put forth in Section 4.5.

4.4.1 Font Table

Section of code establishing the font table allowing the GLCD to display recognizeable characters.

```
// Character encoding for GLCD display
const int font_table[][6] = {
    {0x3e, 0x51, 0x49, 0x45, 0x3e, 0x00}, // 0 //  0
    {0x00, 0x42, 0x7f, 0x40, 0x00, 0x00}, // 1 //  1
    {0x42, 0x61, 0x51, 0x49, 0x46, 0x00}, // 2 //  2
    {0x21, 0x41, 0x45, 0x4b, 0x31, 0x00}, // 3 //  3
    {0x18, 0x14, 0x12, 0x7f, 0x10, 0x00}, // 4 //  4
    {0x27, 0x45, 0x45, 0x45, 0x39, 0x00}, // 5 //  5
    {0x3c, 0x4a, 0x49, 0x49, 0x30, 0x00}, // 6 //  6
    {0x01, 0x71, 0x09, 0x05, 0x03, 0x00}, // 7 //  7
    {0x36, 0x49, 0x49, 0x49, 0x36, 0x00}, // 8 //  8
    {0x0e, 0x49, 0x49, 0x29, 0x1e, 0x00}, // 9 //  9
    {0x00, 0x00, 0x00, 0x00, 0x00, 0x00}, //   // 10
    {0x7e, 0x11, 0x11, 0x11, 0x7e, 0x00}, // A // 11
    {0x7f, 0x49, 0x49, 0x49, 0x36, 0x00}, // B // 12
    {0x3e, 0x41, 0x41, 0x41, 0x22, 0x00}, // C // 13
    {0x7f, 0x41, 0x41, 0x41, 0x3e, 0x00}, // D // 14
    {0x7f, 0x49, 0x49, 0x49, 0x41, 0x00}, // E // 15
    {0x7f, 0x09, 0x09, 0x09, 0x01, 0x00}, // F // 16
    {0x3e, 0x41, 0x49, 0x49, 0x7a, 0x00}, // G // 17
    {0x7f, 0x08, 0x08, 0x08, 0x7f, 0x00}, // H // 18
    {0x41, 0x41, 0x7f, 0x41, 0x41, 0x00}, // I // 19
    {0x20, 0x40, 0x40, 0x40, 0x3f, 0x00}, // J // 20
    {0x7f, 0x08, 0x14, 0x22, 0x41, 0x00}, // K // 21
    {0x7f, 0x40, 0x40, 0x40, 0x40, 0x00}, // L // 22
    {0x7f, 0x02, 0x0c, 0x02, 0x7f, 0x00}, // M // 23
    {0x7f, 0x04, 0x08, 0x10, 0x7f, 0x00}, // N // 24
    {0x3e, 0x41, 0x41, 0x41, 0x3e, 0x00}, // O // 25
    {0x7f, 0x09, 0x09, 0x09, 0x06, 0x00}, // P // 26
    {0x3e, 0x41, 0x51, 0x61, 0x7e, 0x00}, // Q // 27
    {0x7f, 0x09, 0x19, 0x29, 0x46, 0x00}, // R // 28
    {0x26, 0x49, 0x49, 0x49, 0x32, 0x00}, // S // 29
    {0x01, 0x01, 0x7f, 0x01, 0x01, 0x00}, // T // 30
    {0x3f, 0x40, 0x40, 0x40, 0x3f, 0x00}, // U // 31
    {0x1f, 0x20, 0x40, 0x20, 0x1f, 0x00}, // V // 32
    {0x3f, 0x40, 0x38, 0x40, 0x3f, 0x00}, // W // 33
    {0x63, 0x14, 0x08, 0x14, 0x63, 0x00}, // X // 34
    {0x03, 0x04, 0x78, 0x04, 0x03, 0x00}, // Y // 35
    {0x61, 0x51, 0x49, 0x45, 0x43, 0x00}, // Z // 36
    {0x00, 0x00, 0x5f, 0x00, 0x00, 0x00}, // ! // 37
    {0x20, 0x10, 0x08, 0x04, 0x02, 0x00}, // / // 38
    {0x00, 0x60, 0x60, 0x00, 0x00, 0x00}, // . // 39
    {0x48, 0x54, 0x54, 0x54, 0x20, 0x00}, // s // 40
    {0x14, 0x08, 0x3e, 0x08, 0x14, 0x00}, // * // 41
};
```

4.4.2 Example Code

Sections of code pertaining to the configuration and dricing of the GLCD display.

```c
/*
 * Displays a character
 */
void GLCD_putchar(int c){
    int i;
    for(i = 0; i < 6; i++)
        GLCD_data_write(font_table[c][i]);
}

/*
 * Clears the GLCD by writing zeros to the entire screen
 */
void GLCD_clear(void){
    int i;
    for (i = 0; i < (WIDTH * HEIGHT / 8); i++){
        GLCD_data_write(0x00);
    }
    GLCD_setCursor(0,0);
}

/*
 * Write to GLCD controller data register
 */
void GLCD_data_write(unsigned char data){
    P6->OUT |= DC;      // select data register
    SPI_write(data);    // send data via SPI
}

/*
 * Moves the cursor
 */
void GLCD_setCursor(unsigned char x, unsigned char y){
    GLCD_command_write(0x80 | x);   // set column
    GLCD_command_write(0x40 | y);   // set bank row (8 rows per bank)
}

/*
 * Send the initialization commands to PCD8544 GLCD controller
 */
void GLCD_init(void){
    SPI_init();
    // Hardware reset of GLCD controller
    P6->OUT |= RESET;   // deassert Reset PIN

    // Send command words to the display
    GLCD_command_write(0x21);   // extended command mode
    GLCD_command_write(0x98);   // set LCD Vop
    GLCD_command_write(0x04);   // set temp coefficient
    GLCD_command_write(0x17);   // set bias mode
    GLCD_command_write(0x20);   // leave extended mode
    GLCD_command_write(0x0C);   // set display to normal operating mode
}

/*
```

```
 * Write to GLCD controller command register
 */
void GLCD_command_write(unsigned char data){
    P6->OUT &= ~DC;       // select command register
    SPI_write(data);      // send data via SPI
}

/*
 * Sends data to the SPI transmit buffer
 */
void SPI_write(unsigned char data){
    P6->OUT &= ~CE;            // assert /CE
    EUSCI_B0->TXBUF = data;  // write data
    while (EUSCI_B0->STATW & BIT0); // wait for transmission to be done
    P6->OUT |= CE;             // deassert /CE
}

/*
 * Configure the SPI on UCB0
 */
void SPI_init(void){
    // Configure eUSCI_B0
    EUSCI_B0->CTLW0 = 0x0001;       // Put UCB0 in reset mode
    EUSCI_B0->CTLW0 = 0x69C1;       // Determine protocol parameters (PH=0, PL=1, MSB first, Master, SPI, SMCLK)
    EUSCI_B0->BRW = 3;              // 3 MHz / 3 = 1 MHz
    EUSCI_B0->CTLW0 &= ~0x0001;     // enable UCB0 after configuration

    P1->SEL0 |= BIT5 | BIT6;        // P1.5 and P1.6 for UCB0
    P1->SEL1 &= ~(BIT5 | BIT6);     // P1.5 and P1.6 for UCB0

    P6->DIR |= (CE | RESET | DC); // P6.0, P6.6, P6.7 are output pins
    P6->OUT |= CE;                  // CE idle high
    P6->OUT &= ~RESET;              // assert Reset PIN
}
```

4.5 main.c

Here is the entirety of the file, ”main.c,” containing all the code for this project. Imported, is the file, ”msp.h,” that has defined all the structs accessed by this program.

```
#include <stdio.h>
#include "msp.h"

#define   LED1 BIT0 // P1.0 LED1 for blink indicator
#define   LED2 BIT0 // P2.0 LED2 red segment for pop indicator
#define   BUZZ BIT7 // P1.7 Buzzer output
#define MIC_RE BIT0 // P5.0 digital microphone rising-edge input
#define MIC_FE BIT1 // P5.1 digital microphone falling-edge input
#define    POT BIT0 // P4.0 analog potentiometer input
#define     CE BIT0 // P6.0 chip enable (CS)
#define  RESET BIT6 // P6.6 reset
#define     DC BIT7 // P6.7 data/control
```

```c
#define TIMER_PERIOD 1638 // 1638/32768Hz * 2 = 0.1s
#define WIDTH 84     // Display width
#define HEIGHT 48    // Display height

/* Function Prototypes */
void ports_init(void);
void timer_init(void);
void ADC_init(void);
int sample_ADC(void);
void GLCD_command_write(unsigned char);
void GLCD_data_write(unsigned char);
void GLCD_init(void);
void GLCD_clear(void);
void GLCD_setCursor(unsigned char, unsigned char);
void GLCD_putchar(int);
void SPI_init(void);
void SPI_write(unsigned char);
void print_float(float);
void display_time(void);
void print_message(void);
void trigger_alarm(void);
void TA0_N_IRQHandler(void);
void PORT1_IRQHandler(void);

/* Global Variables */
float cutoff;      // Seconds between pops
float time = 0.0; // Start timer at 0.0s
int pop = 0;       // Pop condition
int end = 0;       // End condition

// Character encoding for GLCD display
const int font_table[][6] = {
    {0x3e, 0x51, 0x49, 0x45, 0x3e, 0x00}, // 0 //  0
    {0x00, 0x42, 0x7f, 0x40, 0x00, 0x00}, // 1 //  1
    {0x42, 0x61, 0x51, 0x49, 0x46, 0x00}, // 2 //  2
    {0x21, 0x41, 0x45, 0x4b, 0x31, 0x00}, // 3 //  3
    {0x18, 0x14, 0x12, 0x7f, 0x10, 0x00}, // 4 //  4
    {0x27, 0x45, 0x45, 0x45, 0x39, 0x00}, // 5 //  5
    {0x3c, 0x4a, 0x49, 0x49, 0x30, 0x00}, // 6 //  6
    {0x01, 0x71, 0x09, 0x05, 0x03, 0x00}, // 7 //  7
    {0x36, 0x49, 0x49, 0x49, 0x36, 0x00}, // 8 //  8
    {0x0e, 0x49, 0x49, 0x29, 0x1e, 0x00}, // 9 //  9
    {0x00, 0x00, 0x00, 0x00, 0x00, 0x00}, //   // 10
    {0x7e, 0x11, 0x11, 0x11, 0x7e, 0x00}, // A // 11
    {0x7f, 0x49, 0x49, 0x49, 0x36, 0x00}, // B // 12
    {0x3e, 0x41, 0x41, 0x41, 0x22, 0x00}, // C // 13
    {0x7f, 0x41, 0x41, 0x41, 0x3e, 0x00}, // D // 14
    {0x7f, 0x49, 0x49, 0x49, 0x41, 0x00}, // E // 15
    {0x7f, 0x09, 0x09, 0x09, 0x01, 0x00}, // F // 16
    {0x3e, 0x41, 0x49, 0x49, 0x7a, 0x00}, // G // 17
    {0x7f, 0x08, 0x08, 0x08, 0x7f, 0x00}, // H // 18
    {0x41, 0x41, 0x7f, 0x41, 0x41, 0x00}, // I // 19
    {0x20, 0x40, 0x40, 0x40, 0x3f, 0x00}, // J // 20
    {0x7f, 0x08, 0x14, 0x22, 0x41, 0x00}, // K // 21
```

```
    {0x7f, 0x40, 0x40, 0x40, 0x40, 0x00}, // L // 22
    {0x7f, 0x02, 0x0c, 0x02, 0x7f, 0x00}, // M // 23
    {0x7f, 0x04, 0x08, 0x10, 0x7f, 0x00}, // N // 24
    {0x3e, 0x41, 0x41, 0x41, 0x3e, 0x00}, // O // 25
    {0x7f, 0x09, 0x09, 0x09, 0x06, 0x00}, // P // 26
    {0x3e, 0x41, 0x51, 0x61, 0x7e, 0x00}, // Q // 27
    {0x7f, 0x09, 0x19, 0x29, 0x46, 0x00}, // R // 28
    {0x26, 0x49, 0x49, 0x49, 0x32, 0x00}, // S // 29
    {0x01, 0x01, 0x7f, 0x01, 0x01, 0x00}, // T // 30
    {0x3f, 0x40, 0x40, 0x40, 0x3f, 0x00}, // U // 31
    {0x1f, 0x20, 0x40, 0x20, 0x1f, 0x00}, // V // 32
    {0x3f, 0x40, 0x38, 0x40, 0x3f, 0x00}, // W // 33
    {0x63, 0x14, 0x08, 0x14, 0x63, 0x00}, // X // 34
    {0x03, 0x04, 0x78, 0x04, 0x03, 0x00}, // Y // 35
    {0x61, 0x51, 0x49, 0x45, 0x43, 0x00}, // Z // 36
    {0x00, 0x00, 0x5f, 0x00, 0x00, 0x00}, // ! // 37
    {0x20, 0x10, 0x08, 0x04, 0x02, 0x00}, // / // 38
    {0x00, 0x60, 0x60, 0x00, 0x00, 0x00}, // . // 39
    {0x48, 0x54, 0x54, 0x54, 0x20, 0x00}, // s // 40
    {0x14, 0x08, 0x3e, 0x08, 0x14, 0x00}, // * // 41
};

/**
 * main.c
 */
void main(void)
{
  WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;    // stop watchdog timer

    ports_init();    // Configure Ports
    timer_init();    // Configure Timer_A0
    ADC_init();      // Configure ADC14
    GLCD_init();     // Configure GLCD

    while(1);        // Main loop
}

/*
 * Displays time since last pop vs cutoff interval
 */
void display_time(void){
    GLCD_clear();
    print_float(time);
    GLCD_putchar(38);
    print_float(cutoff);
    GLCD_putchar(40);
    if(pop){
        GLCD_putchar(10);
        GLCD_putchar(41);
        pop = 0;
    }
}

/*
```

```c
 * Prints a floating point number with 2 digits
 * (including one decimal point)
 */
void print_float(float num){
    char num_str[3];
    sprintf(num_str, "%.1f", num);

    int i;
    for(i = 0; i < 3; i++){
        switch(num_str[i]){
            case '0':
                GLCD_putchar(0);
                break;
            case '1':
                GLCD_putchar(1);
                break;
            case '2':
                GLCD_putchar(2);
                break;
            case '3':
                GLCD_putchar(3);
                break;
            case '4':
                GLCD_putchar(4);
                break;
            case '5':
                GLCD_putchar(5);
                break;
            case '6':
                GLCD_putchar(6);
                break;
            case '7':
                GLCD_putchar(7);
                break;
            case '8':
                GLCD_putchar(8);
                break;
            case '9':
                GLCD_putchar(9);
                break;
            case '.':
                GLCD_putchar(39);
                break;
        }
    }
}

/*
 * Flash lights and sound buzzer 4 times when popcorn is done
 */
void trigger_alarm(void){
    GLCD_clear();

    int i;
```

```c
    for(i = 0; i < 4; i++){
        P1->OUT |= BUZZ;          // Activate buzzer
        print_message();          // Display "POPCORN READY!"
        __delay_cycles(300000);   // Wait
        P1->OUT &= ~BUZZ;         // Buzzer off
        GLCD_clear();             // Clear message
        __delay_cycles(300000);   // Wait
    }
    print_message();              // Lingering message
    end = 1;                      // End program
}

/*
 * Display "POPCORN READY!"
 */
void print_message(void){
    GLCD_clear();
    GLCD_putchar(26); // P
    GLCD_putchar(25); // O
    GLCD_putchar(26); // P
    GLCD_putchar(13); // C
    GLCD_putchar(25); // O
    GLCD_putchar(28); // R
    GLCD_putchar(24); // N
    GLCD_putchar(10); //
    GLCD_putchar(28); // R
    GLCD_putchar(15); // E
    GLCD_putchar(11); // A
    GLCD_putchar(14); // D
    GLCD_putchar(35); // Y
    GLCD_putchar(37); // !
}

/*
 * Handle microphone interrupts
 */
void PORT5_IRQHandler(void){
    if(P5->IFG & MIC_RE){    // Rising-edge interrupts
        P5->IFG &= ~MIC_RE;  // Clear interrupt flag
        P2->OUT |= LED2;     // Turn on LED2
        pop = 1;             // Register a 'pop'
    }
    if(P5->IFG & MIC_FE){    // Falling-edge interrupts
        P5->IFG &= ~MIC_FE;  // Clear interrupt flag

        // Debounce signal
        int i;
        for(i=0;i<500;i++);

        time = 0;            // Reset time when a pop is registered
        P2->OUT &= ~LED2;    // Turn off LED2
    }
}
```

```c
/*
 * Handle timer interrupts
 */
void TA0_N_IRQHandler(void){
    TIMER_A0->CTL &= ~TIMER_A_CTL_IFG; // Clear interrupt flag
    if(!end){
        P1->OUT ^= BIT0;               // Toggle blinking LED (P1.0)

        cutoff = sample_ADC() / 819.4; // (0, 4096) -> (0, 5)
        time += 0.1;                   // Increment time
        display_time();                // Update display

        if(time >= cutoff){            // Alarm condition
            time = 0;
            trigger_alarm();
        }
    }
}

/*
 * Displays a character
 */
void GLCD_putchar(int c){
    int i;
    for(i = 0; i < 6; i++)
        GLCD_data_write(font_table[c][i]);
}

/*
 * Clears the GLCD by writing zeros to the entire screen
 */
void GLCD_clear(void){
    int i;
    for (i = 0; i < (WIDTH * HEIGHT / 8); i++){
        GLCD_data_write(0x00);
    }
    GLCD_setCursor(0,0);
}

/*
 * Write to GLCD controller data register
 */
void GLCD_data_write(unsigned char data){
    P6->OUT |= DC;        // select data register
    SPI_write(data);      // send data via SPI
}

/*
 * Moves the cursor
 */
void GLCD_setCursor(unsigned char x, unsigned char y){
    GLCD_command_write(0x80 | x);   // set column
    GLCD_command_write(0x40 | y);   // set bank row (8 rows per bank)
}
```

```c
/*
 * Send the initialization commands to PCD8544 GLCD controller
 */
void GLCD_init(void){
    SPI_init();
    // Hardware reset of GLCD controller
    P6->OUT |= RESET;   // deassert Reset PIN

    // Send command words to the display
    GLCD_command_write(0x21);   // extended command mode
    GLCD_command_write(0x98);   // set LCD Vop
    GLCD_command_write(0x04);   // set temp coefficient
    GLCD_command_write(0x17);   // set bias mode
    GLCD_command_write(0x20);   // leave extended mode
    GLCD_command_write(0x0C);   // set display to normal operating mode
}

/*
 * Write to GLCD controller command register
 */
void GLCD_command_write(unsigned char data){
    P6->OUT &= ~DC;      // select command register
    SPI_write(data);     // send data via SPI
}

/*
 * Sends data to the SPI transmit buffer
 */
void SPI_write(unsigned char data){
    P6->OUT &= ~CE;          // assert /CE
    EUSCI_B0->TXBUF = data; // write data
    while (EUSCI_B0->STATW & BIT0); // wait for transmission to be done
    P6->OUT |= CE;           // deassert /CE
}

/*
 * Configure the SPI on UCB0
 */
void SPI_init(void){
    // Configure eUSCI_B0
    EUSCI_B0->CTLW0 = 0x0001;      // Put UCB0 in reset mode
    EUSCI_B0->CTLW0 = 0x69C1;      // Determine protocol parameters (PH=0, PL=1, MSB first, Master, SPI, SMCLK)
    EUSCI_B0->BRW = 3;             // 3 MHz / 3 = 1 MHz
    EUSCI_B0->CTLW0 &= ~0x0001;    // enable UCB0 after configuration

    P1->SEL0 |= BIT5 | BIT6;       // P1.5 and P1.6 for UCB0
    P1->SEL1 &= ~(BIT5 | BIT6);    // P1.5 and P1.6 for UCB0

    P6->DIR |= (CE | RESET | DC); // P6.0, P6.6, P6.7 are output pins
    P6->OUT |= CE;                 // CE idle high
    P6->OUT &= ~RESET;             // assert Reset PIN
}
```

```
/*
 * Sample ADC14 and perform Analog-to-Digital Conversion (ADC)
 */
int sample_ADC(void){
    int result;

    ADC14->CTL0 |= 1;        // Start a conversion
    while (!ADC14->IFGR0);   // Wait until conversion is complete
    result = ADC14->MEM[0]; // Read conversion result

    return result;
}

/*
 * Configure ADC14
 */
void ADC_init(void){
    // -> Configure P4.7 for A6 (analog input channel 6)
    P6->SEL0 |= POT;
    P6->SEL1 |= POT;
    // -> Configure CTL0
    ADC14->CTL0  = 0x00000010;  // Power on ADC and disable during config
    ADC14->CTL0 |= 0x04080310;  // Configure CTL0 according to instructions
    // -> Configure CTL1
    ADC14->CTL1  = 0x00000020;  // 12-bit resolution (see instructions)
    ADC14->CTL1 |= 0x00000000;  // Configure for memory register 0
    // -> Configure MCTL[0] (memory register 0) to receive input channel 6
    ADC14->MCTL[0] = 0x06;      // A6 input (P4.7), single ended, Vref = AVCC
    // Enable ADC14
    ADC14->CTL0 |= 0x02;        // Enable ADC14 after configuration
}

/*
 * Configure Timer_A0 in up/down mode with predefined period
 * using ACLK (supplied by 32768 Hz LFTX)
 */
void timer_init(void){
    PJ->SEL0 |= BIT0 | BIT1; // Needed for clock/timer

    // Configure clock system
    CS->KEY = CS_KEY_VAL;
    CS->CTL2 |= CS_CTL2_LFXT_EN;
    while(CS->IFG & CS_IFG_LFXTIFG)
        CS->CLRIFG |= CS_CLRIFG_CLR_LFXTIFG;
    CS->CTL1 |= CS_CTL1_SELA_0;
    CS->CLKEN |= CS_CLKEN_REFOFSEL;
    CS->CTL1 &= ~(CS_CTL1_SELS_MASK | CS_CTL1_DIVS_MASK);
    CS ->CTL1 |= CS_CTL1_SELS_2;
    CS->KEY = 0;

    // Configure Timer_A0
    TIMER_A0->CCR[0] = TIMER_PERIOD;
```

```
    TIMER_A0->CTL = TIMER_A_CTL_TASSEL_1 | TIMER_A_CTL_MC_3 | TIMER_A_CTL_CLR | TIMER_A_CTL_IE;

    NVIC->ISER[0] = 1 << ((TA0_N_IRQn) & 31);
    __enable_irq();
}

/*
 * Configure Ports
 */
void ports_init(void){
    // Configure P1
    P1->DIR |=   LED1 | BUZZ;        // Set LED1 and buzzer as outputs
    P1->OUT &= ~(LED1 | BUZZ);       // Initialize LED1 and buzzer to off

    // Configure P2
    P2->DIR |=  LED2;                // Set LED2 as output
    P2->OUT &= ~LED2;                // Initialize to off

    // Configure P5
    P5->DIR &= ~(MIC_RE | MIC_FE);   // Set MIC as input
    P5->IE  |=  MIC_RE | MIC_FE;     // Enable interrupts for MIC
    P5->IES &= ~MIC_RE;              // Rising edge interrupt
    P5->IES |=  MIC_FE;              // Falling-edge interrupt

    // Port-level Interrupt Enable for Port 5
    NVIC->ISER[1] |= 0x00000080;
    __enable_interrupts();
}
```

5 Test Plan and Test Results

For this project, there are five main points of operation. The timer module, microphone, potentiometer, buzzer, and GLCD display should all work together to indicate the status of the Microwave Popcorn Timer and successfully alert the user to the slowing of the popping sufficient to advise stopping the microwave.

1. TIMER INTERRUPTS: Timer_A0 should request interrupts at an interval of 0.1s. This can be observed by the blinking of LED1 five times per second, as each interrupt toggles the LED's state. The display should also update at this same interval.

2. MICROPHONE INTERRUPTS: A 'pop' registered on the microphone should result in two interrupts generated by its digital output pin: one rising-edge and one falling-edge interrupt. Whether a 'pop' has been detected is indicated by the flashing of red LED2 accompanied by an '*' blinking on the far right of the display, both turning on and then off with the rising and falling edges of the digital signal, respectively. The falling-edge interrupt should also result in the resetting of the timer to 0.0s which can be observed on the left of the GLCD display as the numerator of the data presented.

3. ANALOG READING: Adjusting the dial of the potentiometer should result in an updated value for the pop-interval cutoff which can be observed as the denominator of the data presented, appearing in the center of the GLCD display. Turning the dial through its full range of motion should translate to the adjustment of this value from 0.0-5.0s.

4. BUZZER OUTPUT: Once the timer exceeds the pop-interval cutoff value during a periodic check of this condition by the Timer_A0 Interrupt Request Handler, the buzzer should sound four times in sync with a message on the display, alerting the user that the microwave should be stopped.

5. DISPLAY WRITING: While the microphone, buzzer, and timer modules each have ways of observing correct operation without the presence of the display (via the on-board LED's and sound), there is no method of observing the current status of the Microwave Popcorn Timer and its values for timer and pop-interval cutoff without the GLCD operating correctly. The GLCD's correct operation can be observed by the presentation of the data described in the above events and the message, "POPCORN READY!" blinking four times when the alarm is triggered. The message should persist until the device is reset or disconnected from power. This being the only way of observing correct operation is not optimal, however, as it relies on the other components functioning properly, so a test image may prove beneficial to be added in future iterations of this project.

Finally, all these components operating correctly should result in the user being able to view the values for the timer, pop-interval cutoff, and pop indication symbol at any given time. Once the timer exceeds the interval cutoff, the user should be alerted that the popcorn is done. If tests for the above operations succeeded, but undesired behavior is still observed, this indicates there is a software issue.

5.1 Expected and Observed Results

Below are the specific steps for the described scenarios, along with the expected and observed results of those steps in Table 1.

1. With all components wired according to the schematics, supply the MSP-EXP432P401R with board power via the micro-USB port or power ins and press the reset switch (S3).

2. Set the sensitivity by adjusting the trim pot on the sound sensor module until snapping your fingers a foot away from the sensor triggers the LED on the module itself, but not so sensitive that humming or gently speaking from a similar distance will trigger it. (You may wish to place the device beside a microwave nearest its vents and clap or snap your fingers loudly from within the door to simulate a muffled pop).

3. Turn the dial of the potentiometer all the way to the left. Then, slowly rotate the dial clockwise until it can no longer turn, observing the values change on the GLCD display.

4. With the desired cutoff interval selected according to the packing instructions, begin cooking a bag of microwave popcorn (or simulate with clapping or snapping) and press the reset switch (S3) once the popping has picked up in frequency.

	Expected Outcome	Observed Outcome
1	LED1 should start blinking at a rate of 5 bps and continue indefinitely. The timer value (numerator) on the display should begin incrementing consistently in 0.1s intervals.	Observed expected results.
2	Popping (or snapping/clapping) from inside the microwave should cause the red LED2 to blink and the timer value (the numerator on the display) to reset to 0.0s and begin incrementing again. The '*' symbol should also blink on the display to the right of the data, in step with red LED2.	Observed expected outcome, but with an inconsistent sensitivity due to unstable breadboard connections.
3	This should result in the pop-interval cutoff value (the denominator on the display) smoothly transitioning from 0.0s to 5.0s in 0.1s steps.	Observed expected result.
4	Once the popping has slowed to the specified cutoff interval, the timer value should increase until arriving at the same value as the pop-interval cutoff on the display. The buzzer should sound along with the message, "POPCORN READY!" four times with the message on the display persisting until resetting or disconnecting power from the board.	Observed expected result. However, the rate of popping would occasionally vary, causing the alarm to trigger prematurely.

Table 1: Test cases and recorded results.

5.2 Potential & Known Issues

1. The Microwave Popcorn Timer must be started after the popcorn has already begun popping, as the silence before popping begins and/or slower popping at first will trigger

the alarm. This can be fixed by rewriting the logic that calls the "trigger_alarm()" function in the Timer_A0 Interrupt Request Handler.

2. Variations in the interval between pops can also cause the alarm to trigger prematurely. Perhaps this could be solved by taking the average of the last several samples of pop intervals and basing the logic to trigger the alarm off that value instead.

3. The digital output of the sound sensor module responds to all sound within the volume range set by its trim pot, and so the Microwave Popcorn Timer is susceptible to room noise or even the noise of the microwave itself triggering false readings. Analyzing the analog input of the microphone against the digital signal may serve as a solution for this to filter out unwanted noise, however this comes at the cost of increased complexity.

4. The interrupts generated by the microphone on Port 5 during some tests were never handled by the Port 5 Interrupt Request Handler until after the alarm sounded, resulting in incorrect operation. Slowing down the Timer_A0 interval to a period of 0.2s fixed this, as the problem is likely that the microprocessor is too busy with such frequent interrupts from the timer and/or that the functions called by the Timer_A0 Interrupt Request Handler take too much time or resources. Replacing the "__delay_cycles()" functions and while loops with timer interrupts from other on-board timer modules and cutting down on floating point operations might resolve this issue.

5.3 Video Demonstration

A video demonstrating correct operation of the Microwave Popcorn Timer can be found at https://vm.tiktok.com/ZM83Aqpn4/.

6 Discussion and Conclusion

As was pointed out in the previous section, there are many areas where the design of this project can be improved. However, the passing results of the test plan indicate that I was able to implement each of the Microprocessor-Based System Design course concepts into a single device that serves a useful, real-world application. The Microwave Popcorn Timer stands out as a unique and thoughtful device with the potential to be implemented as an actual end product with some alterations and improvements to its design and code. I will continue to implement upon this version 1.0 design as I think of other new ideas and use cases. My ultimate vision for a product like this might be an standalone microwave unit with smart sensing features that can automatically adjust settings to thoroughly and evenly cook food while avoiding overcooking/burning with minimal user input. This could feasibly be done with the implementation of a thermal imaging sensor, gimbaled magnetron and feedback control algorithms similar to the one presented in this project, potentially incorporating a machine learning model AI. I hope you enjoyed following along with this project and welcome any feedback, suggestions, or comments you might have. Please leave them on my GitHub at https://github.com/dtloveless/Microwave-Popcorn-Timer.

References

[1] https://www.sparkfun.com/datasheets/LCD/Monochrome/Nokia5110.pdf, Apr 1999. Data Sheet PCD8544 48 $\times$ 84 pixels matrix LCD controller/driver.

[2] "Simplelink msp432p401r high-precision adc launchpad development kit." https://www.ti.com/store/ti/en/p/product/?p=MSP-EXP432P401R. Accessed: 2021-11-11.

[3] C. Unsalan, H. Gurhan, and M. Yucel, *Programmable Microcontrollers: Applications on the MSP432 LaunchPad*. McGraw-Hill Education, 2017.