University of British Columbia
Electrical and Computer Engineering
ELEC291/ELEC292 Winter 2022
Instructor: Dr. Jesus Calvino-Fraga
Section 201

Project 1: Reflow Oven Controller

Date of Submission: March 2, 2023

Table of Contents

Introduction

This project report introduces the Reflow Oven Controller, a device designed to regulate the temperature of a standard toaster oven using a solid-state relay [1]. Our controller is programmed in assembly language and is capable of measuring temperatures between 25°C to 240°C using a K-type thermocouple with cold junction compensation. The controller has a user interface that allows for selectable reflow profile parameters and displays of temperature, running time, and reflow process current state. The device can speak aloud as part of the user interface using Pulse Coded Modulation (PCM) together with a Digital to Analog Converter (DAC). It plays back the current oven temperature every 5 seconds, and indicates the current state of the reflow process.

The Reflow Oven Controller we have designed has a pushbutton that starts the reflow process and another one that stops it at any moment of the reflow process. The device can send the current oven temperature in Celsius through the serial port of our computer, and the PuTTy running on the computer will read the information and plot the temperature in real-time to provide feedback about the reflow process.

Since safety is an essential aspect of the Reflow Oven Controller, as a safety measure, the reflow process aborts if the oven doesn't reach at least 50 degree celsius in the first 60 seconds of operation. Similarly, the temperature validation data must be collected and analyzed using the lab multimeters. For our process, the maximum acceptable temperature error of the controller is ±3°C for the range 25°C to 240°C.

Our reflow Oven Controller project was successfully designed, built, programmed, and tested as a device capable of regulating the temperature of a standard toaster oven using a solid-state relay. The device is programmed in assembly language and has an easy-to-use user interface that allows for selectable reflow profile parameters and real-time feedback about the reflow process. Safety measures were taken to ensure that the device operates safely and within acceptable temperature error limits.

An overview of the system can be seen in **Figure 1.** and **Figure 2.**.

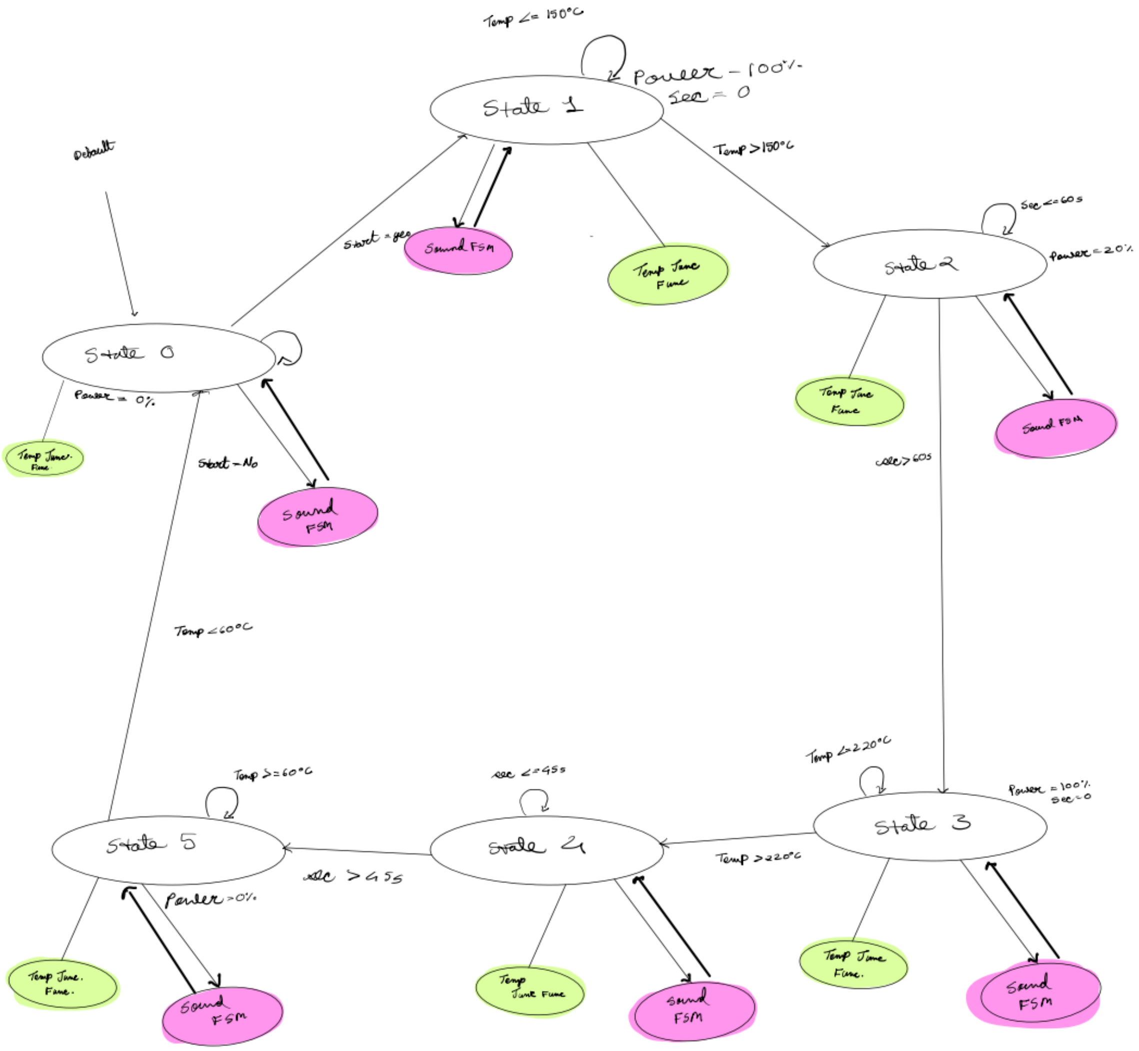
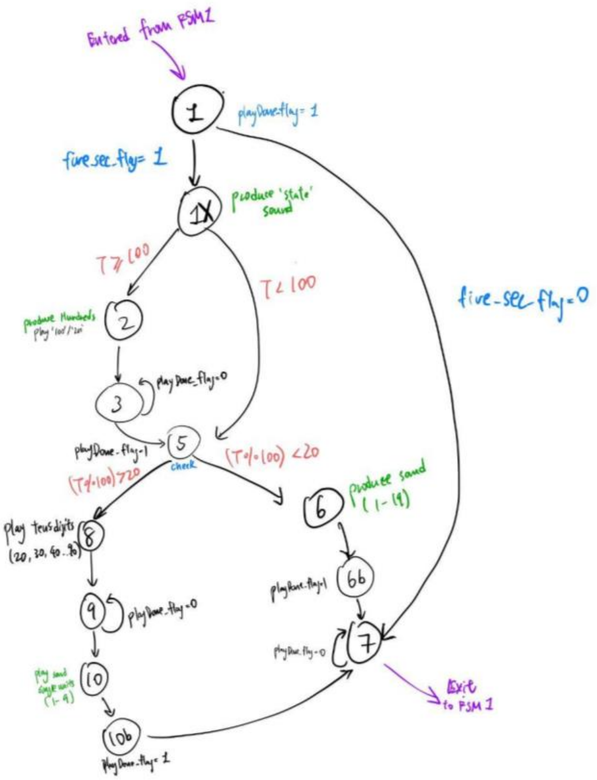


Figure 1a. Software block diagram for the overall design of the reflow oven controller with PWM FSM.



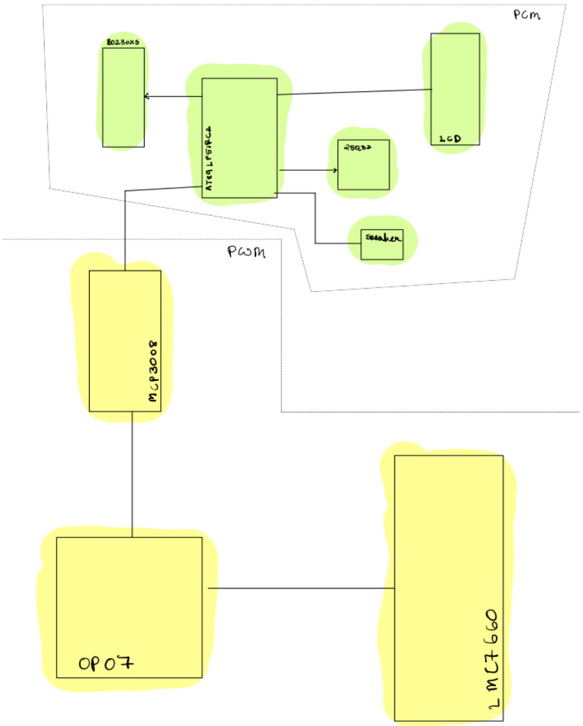


Figure 1b. Expansion of the software block diagram for the sound FSM.

Figure 2. Hardware Block Diagram of the overall design of the reflow oven controller

Investigation

Idea Generation

We started by identifying key functions from the project description and studying the AT89LP51RC2 datasheet [2]. Our initial idea for PWM was to manually configure a timer, but our professor's example code showed us an easier way. We adapted his idea of generating PWM with a changing signal and implemented a finite state machine with flags to keep our system in individual states.

Our team used the provided example code as a starting point to play sound from the 25Q32. To play different sounds at different temperatures, we debated storing sounds for all possible numbers on the chip, but decided against it due to the time it would take to determine the start and end addresses. Instead, we used a second state machine following the one provided by the professor in post [3] on Piazza. We split the sounds into three groups, 1-20, 10-100 in increments of 10, and 100 and 200, and used conditional branches to change states.

Investigation Design

To assemble the circuit on the breadboard, we utilized various resources including datasheets and internet sources to identify electrical components and ensure proper assembly [1][2][4][5]. During the hardware testing phase, we used a DMM to check the voltage across certain components, which helped us identify a faulty opamp. We used block diagrams to help us code for the FSMs and oscilloscopes were employed to confirm whether the FSMs were changing states. Meanwhile, temperature readings were verified using PuTTy. Once we completed the PWM aspect of the project, we encountered some difficulties integrating the sound component, as the two FSMs were interfering with each other. However, by continually debugging and checking the speaker's output, we were eventually able to overcome the issue and ensure that the correct states and temperatures were being read out.

Data Collection

The project data was gathered utilizing a range of advanced tools such as the oscilloscope, multimeter, PuTTy, and python. Below is an outline of the procedures employed and the tools utilized to obtain the data:

- *Voltage Outputs:* To ensure proper functionality, the output voltage of specific components was assessed using the multimeter. For instance, the output of the opamp was measured to validate its outputting the accurate voltage.
- *PWM Signals:* The PWM output signal on the AT89lP51RC2 [2] was evaluated and monitored using the oscilloscope to confirm the signal's proper alteration based on the FSM state.
- *Oven Temperature:* The oven temperature was persistently monitored using PuTTy and python to ensure that it remained within the appropriate temperature ranges for specific FSM states.

Data Synthesis

Our group made sure the correct conclusions were reached by thorough debugging, testing and comparing obtained results with expected ones. This involved meeting two main criteria:

- Ensuring the temperature from PuTTy was matching the temperature read out by the speaker.
- Ensuring the state changes were happening at the right temperature and time by comparing it with time elapsed on an external stopwatch and temperature reading from PuTTy.

Analysis of Results

Our group ensured high precision of results by continuously cross-referencing our actual oven temperature with the expected results from the multimeter. This also allowed us to spot potential sources of inaccuracies and debug accordingly. However, as seen in **Figure 8** and **Figure 9**, there are a few points where there is a discrepancy between the two sets of data. This is bound to happen due to conditions of the environment around us which can fluctuate the two results. The way we know that our data is working correctly is because the two data correspond quite closely to each and stay within ±3°C between 25°C to 240°C for most parts. Keeping in mind the safety aspect of the project, we were sure to abort the reflow process when the oven didn't reach 50°C in the first 60 seconds of the reflow process. Additionally, we aimed to keep the error in temperature within the maximum acceptable range of ±3°C between 25°C to 240°C.

Design

Use of Process

Various project components were thought through and worked on separately before being incorporated into the final design. Prior to implementation, ideas and solutions for individual project components were thoroughly researched, assessed, optimized, and directed with an engineering problem-solving approach. The controller's features were first enumerated and discussed while making sure we would still have enough electrical thought through the pins on the AT89LP51RC2 microprocessor [2] in order to meet the design specifications. Additionally, we investigated the 8051 assembly language's [5] potential and limitations in relation to this project. Since we tried to make our code as simple as possible, dealing with the jump instruction index boundaries limitation proved to be difficult. To control the reflow process and push buttons that alter the reflow settings and LCD display mode, a finite state machine was proposed.

Using the AT89LP51RC2 microprocessor [2], we designed and planned the development of our code. Trial runs of the reflow settings were done in our system's testing, and we also made sure that our temperature readings were accurate within a reasonable error.

Need and Constraint Specification

A reflow oven controller that our team created and put together was intended to control a 1500 watt oven toaster using a microcontroller. The oven must be capable of soldering components onto a PCB board after the reflow process. Pulse-Width Modulation (PWM), is the technique used to control the oven power by transmitting a variable square wave signal that would control the power output. The selected microcontroller included an interactive, updating LCD display with characteristics like soak and reflow temperature, and it needed to be programmed in assembly language. Every 5 seconds, the oven temperature has to be read aloud along with voice feedback. We created sound files that contained the audio for six different numbers and reflow process stages. Pushbuttons were used to operate the interactive LCD display, which made it possible to change the reflow process's parameters. This device's primary function was to solder the applied components, such as capacitors and microprocessors, to a printed circuit by carefully controlling the temperature of the board.

Problem Specification

In order to develop additional features for our reflow process, we had to take into account certain constraints outlined in the provided specification. One example is the correlation between power, temperature, and states. Thus, adding a button to change the power could potentially disrupt the reflow process by changing the temperature. Additionally, for the PCM part, we had to ensure that the readout time is less than 5 seconds as the sound is read aloud every 5 seconds. Going over the 5-second limit could interrupt the read-aloud in another state.

With these constraints in mind, we decided to include a timer in our design that measures the time elapsed since the start of the reflow process. This timer is initiated with the press of a button on our circuit board and the time is displayed on the LCD. If the user wants to return to the initial screen on the LCD, they can simply press the button again and the timer will take them back to the start.

Solution Generation

Our solution was to implement two separate FSM's, one for the sound, and one for the PWM (See **Appendix J** and **Appendix K).**

The sound FSM (PCM) generates sound based on the input temperature and state flags from another PWM. It uses a "five_second_flag" to ensure sound is produced every 5 seconds. It breaks down the temperature into hundreds, tens, and units digits and plays the corresponding sound for each digit. It also ensures that each sound is played fully before transitioning to the next state. The sound indexes are obtained by extracting a certain length of audio from a pre-recorded .wav file using "Computer_Sender.exe". One mistake that was made in our sound part was not adding enough wait times so the read-aloud process could be completed which resulted in having the wrong sound in each state of our process or reading the same sound in each state of the process. The way we tackled this problem is by having an initial state zero with wait times which our sound states could go back to so that there is enough time in each state for the read-aloud process to be completed.

The PWM FSM (see **Appendix J**) is used to alter how much power is provided by the oven at each reflow state. There are flags for each state of the reflow process, that is set within the state loop, while every other state flag is cleared. In each state, the power of the oven is changed depending on what reflow process it is in.

Solution Evaluation

We had several design concepts for some of the project criteria:

Sound:

- Design Concept 1: Trigger a designated audio signal for each state of the sound Finite State Machine (FSM).
- Design Concept 2: Generate individual functions or loops that can be called within the sound Finite State Machine (FSM) to trigger a specific audio signal(see **Appendix K)**.
- Chosen Design: We decided to proceed with design concept 2, as it simplifies the process of identifying and resolving any sound-related problems.

Cold Junction:

- Design Concept 1: Assuming a cold junction temperature of 22 degrees Celsius, as it is the typical temperature of the laboratory room where the reflow oven is being tested.
- Design Concept 2: Incorporate the LM355 temperature sensor readings as the temperature of the cold junction when performing calculations of the oven temperature.
- Chosen Design: We opted for design concept 1 as it would entail fewer debugging efforts for both hardware and software in the event of any problems.

Op Amp Gain:

- Design Concept 1: Generate a random operational amplifier (op amp) gain and obtain corresponding resistor values based on the gain obtained.

- Design Concept 2: Determine the operational amplifier (op amp) gain required to produce an output signal that corresponds to the actual temperature magnitude being measured.
- Chosen Design: We have chosen design concept 2 as it would minimize the number of calculations necessary to obtain the temperature value

Detailed Design

Sound Wave Circuit

The 25Q32 is the SPI Memory block, onto which the .wav files are flashed and stored. To play the sound files, we use a timer running at 22050 Hz to read a byte from the 25Q32, and that is sent to the internal DAC in the AT89 microcontroller [6], which outputs a signal on a certain pin.

The LM386 is an audio amplifier that is connected to the output pin of the DAC. It takes the signal from that pin and amplifies it enough such that the speaker can generate a good quality and volume sound. Adjusting the potentiometer that the LM386 is connected to can reduce the distortion on the audio as well. The MCP1700 is a voltage regulator used to provide 3.3V to the 25Q32 (source: MCP1700 datasheet). It takes an input of 5V and converts it to 3.3V, while also increasing the output current to obey power conservation rules.

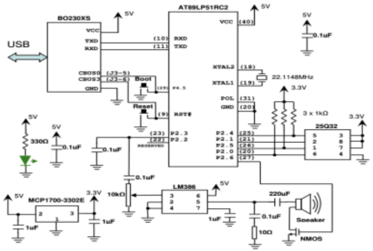


Figure 3. Sound wave circuit connection to the A89LP51RC2 microcontroller.

Finite State Machine for Sound System

To create a voice indicator for the current temperature in our reflow controller, we utilized a Finite State Machine (FSM2) with 9 states, shown in **Figure 1b**. The purpose of the FSM is to sequentially break down the temperature value into tens and single digits, allowing for clear and concise voice output (see **Appendix K**). For example, if the temperature is 143 degrees Celsius, the FSM will first produce "one hundred," followed by "forty," and finally "three."\

We check the "five_sec_flag" in STATE_ONE to see if 5 seconds have passed, and jump to STATE_ONE_X to play the sounds for the "states" and "temperature" if the flag is on. Otherwise, we jump to STATE_SEVEN. We use the state_flags from FSM1 to determine which state's sound to play and use the macro function PLAY_SOUND to play the sound. In STATE_ONE_X, we use subb a,#100 to determine if the temperature is greater or equal to 100 and jump to either STATE_FIVE or STATE_TWO accordingly. We use div ab in STATE_TWO to determine whether the hundreds digit is 1 or 2, and jump to the appropriate state. STATE_THREE uses a "done_playing_flag" to ensure the sound finishes before moving to another state. STATE_FIVE determines whether the tens digit is a multiple of 10 or between 1-19, and jumps to the appropriate state. In STATE_SIX, we play each digit individually using XRL. STATES_SIX_B, NINE, and TEN_B use a PlayDone_flag as well. In STATE_EIGHT, we use div ab to determine which increment of 10 to play. We use "xrl" in STATE_TEN to compare the remainder with each value between 1-9. Finally, we return to FSM1 in STATE_SEVEN by jumping to the state with the state flag set to 1.

Setting up Sound Indexes

We used a pre-existing "Wav_from_Text.vbs" file to create our "voice assistance" with numbers, welcome messages, and state sounds. However, extracting the desired length of audio from the file was a challenge. To address this, we utilized “Computer_Sender.exe” with hexadecimal inputs to manually determine the starting memory address and number of bytes for each sound. We had to do this manually as the provided sound index generator was inaccurate.

PLAY_SOUND Macro block

We created a macro block named PLAY_SOUND(%0, %1,%2, %3,%4,%5) to simplify the process of sending memory addresses and the number of bytes to the SPI memory. The macro block takes in six inputs, the first three (%0-%3) being memory addresses, and the next three (%3-%5) being the number of bytes to play. It then recalls sound indexes and plays the sound accordingly, setting the speaker on. This eliminates the need to copy and paste addresses into the accumulator and lcall Send_SPI.

Testing and debugging FSM2

FSM2 testing is manageable as the temp variable in FSM2 depends on FSM1 calculation. Speaker sound is verified using PuTTy to compare values. Speaker sound and welcome message were tested, but audio quality was poor due to low circuit resistance. Increased resistance improved sound quality.

State Machines for PWM Control of Reflow Oven:

To control the oven using PWM and encode different stages of the reflow process, we used a finite state machine with 6 states(se **Appendix J)**. Our program enters “state0” upon reset to configure temperature and time settings. Upon hitting the “START” button, we transition to state 1, “state1”. To control PWM, we use a two-byte variable named “pwm” that stores the power ratio we want. We use this variable in our Timer 2 ISR to control the PWM signal.

In state 1, we included two subsections to implement a safety feature. If we don't reach the soak temperature within 60 seconds, we abort the process by jumping to “ABORT” in state 0 and turning off the power supplied to the oven.

To transition between the rest of the states, we rely on the system to internally determine when to switch states. We store the current temperature in “temp” and the current time in “seconds” and compare these values to the preset soak/reflow temperature and time. If our system reaches the soak temperature in state 1, we move to state 2 and stay for the designated soak time. Then, we move to state 3 and stay there until our reflow temperature is reached. We transition to state 4 and stay for the specified reflow time before automatically transitioning to state 5, where we permanently set the PWM signal to low and wait for the oven to cool.

To test proper state transitions, we used an oscilloscope to display the PWM signal and a debugging LED to track state changes. The LED helped us catch small comparison errors that would keep us in the same state or cause us to skip states.

PWM Signal Generation

Appendix E shows the code used to generate an output PWM signal in Timer 2. Essentially, we compare “pwm” to “Count1ms”, which is a variable used to count how much of a millisecond has passed.

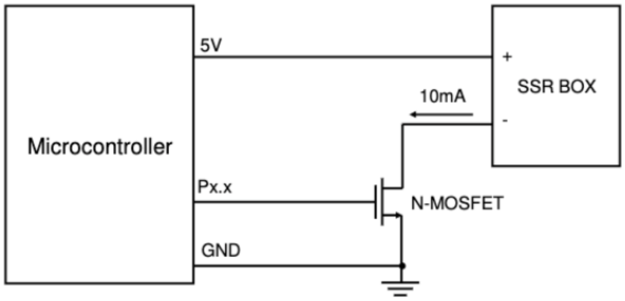


Figure 4. AT89LP51RC2 [2] Microcontroller connection to SSR box (from “Project1 - Reflow Oven Controller 2023”)

The SSR box is used to control the reflow oven. The SSR box itself is controlled by a PWM output from our chosen pin, and an N-MOSFET (see **Figure 4.)**. When the PWM signal is high, the N-MOSFET connects the SSR box to the ground and allows current to flow through, powering the SSR box and consequently the oven. When the PWM signal is low, the N-MOSFET does not allow current to flow through, essentially stopping the SSR box and the reflow oven from being powered. The PWM signal’s on/off behavior is used to generate analog voltage values by using discrete on/off pulses. For example, with a 5V source, if our PWM signal is on 20% of the time and off 80% of the time, it will appear as if we are actually supplying the oven with 1V.

Timer 2 Setup

We decided to use timer 2 to control our PWM signal with a period of 1 second, based on the advice of our professor. To achieve this, we set the timer's frequency to 1000Hz (i.e., a period of 0.001 seconds) and reused most of our timer 2 code from a previous lab.

In our implementation, we used flags to keep track of time and trigger functions. Specifically, we created a one-second flag called "one_second_flag" to track the reflow process time and signal the temperature printing function. We also implemented a five-second flag called "five_second_flag" to trigger a sound from the second FSM.

Initially, we attempted to call the temperature function from within the ISR for faster temperature readings. However, this caused glitches in our code. After research, we realized that calling functions from within an ISR can interfere with the timer triggering at the proper frequency. As a result, we called the temperature function from the states instead.

The initialization function, Timer2_Init, is available in **Appendix E**.

SPI Communication

The SPI communication code for the 25Q32 and MCP3008 components used in the project (see **Appendix D**). Initially, we reuse the code from Lab 3 for communication between the MCP3008 and the AT89 microcontroller [4], as the components were the same. However, we had to use a specific code provided by the professor to read data from the 25Q32. To initialize communication with the 25Q32, we used a similar template as for the MCP3008 by setting the MY_MISO pin bit and clearing the MY_SCLK pin.

When trying to print values to PuTTy, we encountered problems as our code only opened the terminal without printing any values. We realized we forgot to initialize SPI communication and change the pin locations. We also had trouble with displaying the correct temperature, but after checking the thermocouple wire and signal with a multimeter, we discovered that we stored the LSBs in the MSBs, resulting in a faulty temperature reading.

To ensure temperature readings were sent every one second, we added code to the TEMP_JUNCTION function to only print values when the one-second flag was set.

Temperature Reading

To read the temperature, we placed a thermocouple wire inside of the reflow oven and fed it to the input of a difference amplifier(see **Figure 5.)**. We designed for the op-amp to have a gain of 100 - making R1 = 10K Ohm and R2 = 100 Ohm. The output of the opamp was fed to a pin on the microcontroller so that we could read the signal.

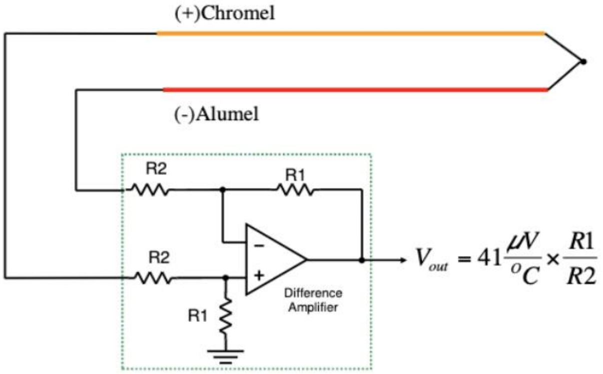


Figure 5. Thermocouple connection to OP07 Op-Amp [4] (from “Project1 - Reflow Oven Controller 202”).

To provide V+ on the OP07 Op-Amp chip (see **Figure 6.)**, we connected pin 7 to power, and to provide V-, we connected pin 4 to the output of an LMC7660, which provided the negative voltage for us.

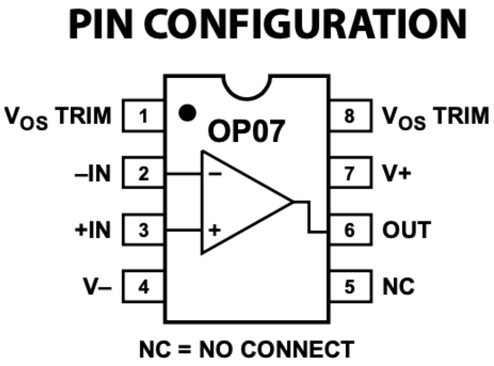


Figure 6. OP07 chip pin configuration.

We put the temperature calculation and printing code in a function called "TEMP_JUNCTION" using math32.inc library. This simplified our FSM and made it easier to read. We adjusted the formula, $factor = \frac{4.092}{1023*41.5E-6*GAIN}$, to match our amplifier's gain.

At first, we aimed for a gain of 96 to avoid complex calculations, but it resulted in incorrect numbers. We then tested gains of 300 and 100, finally achieving success with a gain of 100. To simplify calculations, we assumed a constant cold junction temperature of 22°C.

Initially, we struggled to display the correct number and attempted to modify our software and calculations without success. We suspected our amplifier was faulty and used a multimeter to test the signals and voltages. We found the differential voltage was not being amplified and used an oscilloscope to confirm the signal was distorted. Adding a capacitor did not help, but replacing the OP07 [2] chip fixed the issue.

Live-Long Learning

Our team faced a knowledge gap in implementing PWM on the microcontroller using timers. Initially, we studied the datasheet to manually send a signal to the pin for PWM output, but eventually, our professor provided examples for configuring the timers to PWM mode. We were not aware of this capability, which could have made our task easier. Even though we could not implement a working version of PWM with the timer method, studying the different capabilities of timers was valuable for future applications.

CPEN 211 proved to be a useful course for our project. It introduced us to the basics of assembly, and we learned to use diagrams to map our conditionals. This helped us navigate complex designs and code conditionals in assembly much faster. Additionally, the course taught us about state machines and how a state changes only when a specific input is received. This fundamental concept was helpful in programming states in 8051 assembly, despite not having access to specific templates for FSMs in Verilog.

Another course that helped us was ELEC 201. We gained experience in using oscilloscopes to read our output signals, which proved beneficial in debugging our PWM code. The oscilloscope helped us identify if we were stuck in a particular state or skipping states, and it helped us determine if our code was faulty or if the oven was broken.

Our team had a knowledge gap in implementing PWM on the microcontroller, which we overcame by studying the timer's capabilities. CPEN 211 and ELEC 201 courses were helpful in providing us with the necessary skills and knowledge to navigate complex designs, program states, and debug our code using oscilloscopes. These experiences were invaluable and will benefit us in future projects.

Conclusion:

This report outlines the development of a voice-feedback reflow oven controller with an LCD display for safe soldering of an EFM8 PCB board. Project components were carefully studied, assessed, and optimized using a problem-solving engineering methodology before being included in the final design. The oven controller follows a model shown in Appendix B and employs states listed in Figure 1a in its state machine. A safety feature terminates the reflow process if the thermocouple's temperature reading does not increase to at least 60°C within the first 60 seconds. The project involved combining LM335, thermocouple, and ADC to measure temperatures, computing and sending temperature to LCD and live plot using Python software, and concatenating sound files into a speaker wav file. Implementation strategies included a BCD counter for voice feedback and start announcing every 5 seconds during soak/reflow states. The project emphasized practical and theoretical knowledge of op-amps, transistors, and microcontroller assembly coding. The project took several weeks to complete, with approximately 70-80 hours spent on it, mainly reading temperature values, debugging issues with the finite state machine and speaker, and requiring several all-nighters to meet the deadline.

References

[1] Calviño-Fraga, J., Lecture slides: Project 1 – Reflow Oven Controller, 2023.

[2] Atmel, AT89LP51RC2 Datasheet - Atmel Corporation, alldatasheet.com, 2011.

[3] Calviño-Fraga, J., How to play the current oven temperature?, ELEC 291/292 201 2022W2 Piazza post @223, February, 2023.

[4] Analog Devices, Ultralow Offset Voltage Operational Amplifier, OP07, analog.com, 2002-2011.

[5] Intel, MCS51 Microcontroller Family User's Manual, MIT.edu, February, 1994.

[6] Calviño-Fraga, J., Introduction to the AT89LP51RC2 microcontroller lecture - AT89 instructions, UserManual.wiki., 2019.

Bibliography

Glisson, T. H.., Introduction to Circuit Analysis and Design, Springer, 2012.

Appendix A : Source Code - Variable Declaration

Variable Declaration

```
$NOLIST
$MODLP51RC2
$LIST

CLK             EQU 22118400 ; Microcontroller system crystal frequency in Hz

;Timer 2 for checking the amount of time that has passed
TIMER2_RATE   EQU 1000        ; 1000Hz, for a timer tick of 1ms
TIMER2_RELOAD EQU ((65536-(CLK/TIMER2_RATE)))

SHIFT_PB equ P0.6 ;originally 2.4
;TEMP_SOAK_PB equ P4.5
TIME_SOAK_PB equ P0.3
TEMP_REFL_PB equ P0.2
TIME_REFL_PB equ P0.0
TEMP_SOAK_PB equ P0.5
PWM_OUTPUT   equ P1.4 ;dupliacted pin for p3.4
START          equ P0.7
DEBUG          equ P0.1


TIMER1_RATE    EQU 22050      ; 22050Hz is the sampling rate of the wav file we are playing
TIMER1_RELOAD  EQU 0x10000-(CLK/TIMER1_RATE)
BAUDRATE       EQU 115200
BRG_VAL        EQU (0x100-(CLK/(16*BAUDRATE)))

SPEAKER   EQU P2.6

;pins used fo rSPI
FLASH_CE  EQU  P2.5
MY_MOSI   EQU  P2.4
MY_MISO   EQU  P1.6
MY_SCLK   EQU  P2.7


; Commands supported by the SPI flash memory according to the datasheet
WRITE_ENABLE   EQU 0x06  ; Address:0 Dummy:0 Num:0
WRITE_DISABLE  EQU 0x04  ; Address:0 Dummy:0 Num:0
READ_STATUS    EQU 0x05  ; Address:0 Dummy:0 Num:1 to infinite
READ_BYTES     EQU 0x03  ; Address:3 Dummy:0 Num:1 to infinite
READ_SILICON_ID  EQU 0xab  ; Address:0 Dummy:3 Num:1 to infinite
FAST_READ      EQU 0x0b  ; Address:3 Dummy:1 Num:1 to infinite
WRITE_STATUS   EQU 0x01  ; Address:0 Dummy:0 Num:1
WRITE_BYTES    EQU 0x02  ; Address:3 Dummy:0 Num:1 to 256
ERASE_ALL      EQU 0xc7  ; Address:0 Dummy:0 Num:0
ERASE_BLOCK    EQU 0xd8  ; Address:3 Dummy:0 Num:0
READ_DEVICE_ID   EQU 0x9f  ; Address:0 Dummy:2 Num:1 to infinite

;SPI pins used for MCP3008 ADC
CE_ADC         EQU  P2.0
MY_MOSI_MCP3008   EQU  P2.1
MY_MISO_MCP3008   EQU  P1.3
MY_SCLK_MCP3008   EQU  P1.2


LCD_RS equ P3.2
LCD_E  equ P3.3
LCD_D4 equ P3.4
LCD_D5 equ P3.5
LCD_D6 equ P3.6
LCD_D7 equ P3.7

```

```


org 0x0000
    ljmp main

; Timer/Counter 0 overflow interrupt vector
org 0x000B
    reti ;ljmp Timer0_ISR

org 0x001B ; Timer/Counter 1 overflow interrupt vector. Used in this code to replay the wave file.
    ljmp Timer1_ISR

; Timer/Counter 2 overflow interrupt vector
org 0x002B
    ljmp Timer2_ISR

;variables
dseg at 30H
w:   ds 3 ; 24-bit play counter.  Decremented in Timer 1 ISR.
time_soak: ds 1
time_refl: ds 1
temp_soak: ds 1
temp_refl: ds 1
time_soak_count: ds 1
time_refl_count: ds 1
;counter: ds 1
temp: ds 4
pwm: ds 2
five_second_counter: ds 1
seconds: ds 1
Count1ms: ds 2
x:   ds 4
y:   ds 4
bcd: ds 5
result: ds 2



        ; In the 8051 we have variables that are 1-bit in size.  We can use the setb, clr, jb, and jnb
; instructions with these variables.  This is how you define a 1-bit variable:
        bseg
        half_seconds_flag: dbit 1 ; Set to one in the ISR every time 500 ms had passed
        state0flag: dbit 1
        state1flag: dbit 1
        state2flag: dbit 1
        state3flag: dbit 1
        state4flag: dbit 1
        state5flag: dbit 1
        five_second_flag: dbit 1
        done_playing_flag: dbit 1
        mf: dbit 1
        one_second_flag: dbit 1

        $NOLIST
        $include(math32.inc)
        $include(LCD_4bit.inc)
        $LIST

        cseg
```

Appendix A : Source Code - Timer 1 ISR

Timer 1 ISR

```
;------------------------------------;
; ISR for Timer 1.  Used to playback  ;
; the WAV file stored in the SPI      ;
; flash memory.                       ;
;------------------------------------;
Timer1_ISR:
    ; The registers used in the ISR must be saved in the stack
    push acc
    push psw

    ; Check if the play counter is zero.  If so, stop playing sound.
    mov a, w+0
    orl a, w+1
    orl a, w+2
    jz stop_playing

    ; Decrement play counter 'w'.  In this implementation 'w' is a 24-bit counter.
    mov a, #0xff
    dec w+0
    cjne a, w+0, keep_playing
    dec w+1
    cjne a, w+1, keep_playing
    dec w+2

keep_playing:
    setb SPEAKER
    lcall Send_SPI ; Read the next byte from the SPI Flash...
    ;mov P0, a ; WARNING: Remove this if not using an external DAC to use the pins of P0 as GPIO
    ;add a, #0x80
    mov DADH, a ; Output to DAC. DAC output is pin P2.3
    orl DADC, #0b_0100_0000 ; Start DAC by setting GO/BSY=1
    sjmp Timer1_ISR_Done

stop_playing:
    clr TR1 ; Stop timer 1
    setb FLASH_CE  ; Disable SPI Flash
    clr SPEAKER ; Turn off speaker.  Removes hissing noise when not playing sound.
    mov DADH, #0x80 ; middle of range
    orl DADC, #0b_0100_0000 ; Start DAC by setting GO/BSY=1

Timer1_ISR_Done:
    pop psw
    pop acc
    reti
```

Appendix C : Source Code - Flash SPI Block

Flash SPI Block

```

       ;---------------------------------;
; Sends AND receives a byte via   ;
; SPI.                                ;
;---------------------------------;
       Send_SPI:
               SPIBIT MAC
               ; Send/Receive bit %0
               rlc a
               mov MY_MOSI, c
               setb MY_SCLK
               mov c, MY_MISO
               clr MY_SCLK
               mov acc.0, c
               ENDMAC

               SPIBIT(7)
               SPIBIT(6)
               SPIBIT(5)
               SPIBIT(4)
               SPIBIT(3)
               SPIBIT(2)
               SPIBIT(1)
               SPIBIT(0)

        ret


       ;---------------------------------;
; SPI flash 'write enable'            ;
; instruction.                        ;
;---------------------------------;
       Enable_Write:
           clr FLASH_CE
           mov a, #WRITE_ENABLE
           lcall Send_SPI
           setb FLASH_CE
           ret


       ;---------------------------------;
; This function checks the 'write ;
; in progress' bit of the SPI         ;
; flash memory.                       ;
;---------------------------------;
       Check_WIP:
           clr FLASH_CE
           mov a, #READ_STATUS
           lcall Send_SPI
           mov a, #0x55
           lcall Send_SPI
           setb FLASH_CE
           jb acc.0, Check_WIP ;  Check the Write in Progress bit
           ret



       ; Send a character using the serial port
       putchar:
               jnb TI, putchar
               clr TI
               mov SBUF, a
               ret
```

```
;--------------------------------;
; Receive a byte from serial port ;
;--------------------------------;
getchar:
    jbc    RI,getchar_L1
    sjmp getchar
getchar_L1:
    mov    a,SBUF
    ret
```

Appendix D : Source Code - SPI Initialization

SPI Initialization

```
INIT_SPI_FLASH:
    setb MY_MISO
    clr MY_SCLK

INIT_SPI:
        setb MY_MISO_MCP3008   ; Make MISO an input pin
        clr MY_SCLK_MCP3008    ; For mode (0,0) SCLK is zero
        ret
DO_SPI_G:
        push acc
        mov R1, #0     ; Received byte stored in R1
        mov R2, #8     ; Loop counter (8-bits)
DO_SPI_G_LOOP:
        mov a, R0      ; Byte to write is in R0
        rlc a          ; Carry flag has bit to write
        mov R0, a
        mov MY_MOSI_MCP3008, c
        setb MY_SCLK_MCP3008   ; Transmit
        mov c, MY_MISO_MCP3008  ; Read received bit
        mov a, R1      ; Save received bit in R1
        rlc a
        mov R1, a
        clr MY_SCLK_MCP3008
        djnz R2, DO_SPI_G_LOOP
        pop acc
        ret



; Configure the serial port and baud rate
InitSerialPort:
        ; Since the reset button bounces, we need to wait a bit before
        ; sending messages, otherwise we risk displaying gibberish!
        mov R1, #222
        mov R0, #166
        djnz R0, $   ; 3 cycles->3*45.21123ns*166=22.51519us
        djnz R1, $-4 ; 22.51519us*222=4.998ms
        ; Now we can proceed with the configuration
    orl    PCON,#0x80
    mov    SCON,#0x52
    mov    BDRCON,#0x00
    mov    BRL,#BRG_VAL
    mov    BDRCON,#0x1E ; BDRCON=BRR|TBCK|RBCK|SPD;
ret
```

Appendix E : Source Code - Timer 2

Timer 2

```
Timer2_Init:
    mov T2CON, #0 ; Stop timer/counter.  Autoreload mode.
    mov TH2, #high(TIMER2_RELOAD)
    mov TL2, #low(TIMER2_RELOAD)
    ; Set the reload value
    mov RCAP2H, #high(TIMER2_RELOAD)
    mov RCAP2L, #low(TIMER2_RELOAD)
    ; Init One millisecond interrupt counter.  It is a 16-bit variable made with two 8-bit parts
    clr a
    mov Count1ms+0, a
    mov Count1ms+1, a
    ; Enable the timer and interrupts
        setb ET2  ; Enable timer 2 interrupt
        setb TR2  ; Enable timer 2
    ret


;---------------------------------;
; ISR for timer 2                 ;
;---------------------------------;
Timer2_ISR:
    clr TF2  ; Timer 2 doesn't clear TF2 automatically. Do it in ISR
    ;cpl P1.0 ; To check the interrupt rate with oscilloscope. It must be precisely a 1 ms pulse.

    ; The two registers used in the ISR must be saved in the stack
    push acc
    push psw

    ; Increment the 16-bit one mili second counter (16 bits can store up to 35 535, we only need up to 1000 so we're good)
    inc Count1ms+0    ; Increment the low 8-bits first (b/c low 8 bits can only store up to 255)
    mov a, Count1ms+0 ; If the low 8-bits overflow, then increment high 8-bits
    jnz Inc_Done
    inc Count1ms+1

Inc_Done:
    clr c
    mov a, pwm+0
    subb a, Count1ms+0
    mov a, pwm+1
    subb a, Count1ms+1
    ; if count1ms > pwm_ratio, carry is set
    ;cpl c
    mov PWM_OUTPUT, c


    ; Check if one second has passed
    mov a, Count1ms+0
    cjne a, #low(1000), Timer2_ISR_done ; Warning: this instruction changes the carry flag!
    mov a, Count1ms+1
    cjne a, #high(1000), Timer2_ISR_done

    ; 1000 milliseconds have passed.  Set a flag so the main program knows
    setb one_second_flag ; Let the main program know one second had passed

```

```
        clr a
        mov Count1ms+0, a ; clearing these bc we have determined that we've reached 1000
        mov Count1ms+1, a ;clearing

        ;lcall TEMP_JUNCTION
        inc seconds
        mov a, five_second_counter
        cjne a, #5, State_Logic ;i dont think we need to compare to flag? just gotta compare to counter
        ; otherwise if we ARE at 5 seconds continue on
        setb five_second_flag ;
        mov five_second_counter, #0x00
        sjmp State_Logic_Skip_Inc
        ;carry on


        ; state 2 is for preheat/soak
        State_Logic: inc five_second_counter
        State_Logic_Skip_Inc: jnb state2flag, state4Count ;jump to this line if we don't want to increment 5 sec counter

    Timer2_ISR_done:
        pop psw
        pop acc
        reti
```

Appendix F : Source Code - LCD and Variable Storage

LCD and Variable Storage

```
SendToLCD:
mov b, #100
div ab
orl a, #0x30 ; Convert hundreds to ASCII
lcall ?WriteData ; Send to LCD
mov a, b       ; Remainder is in register b
mov b, #10
div ab
orl a, #0x30 ; Convert tens to ASCII
lcall ?WriteData; Send to LCD
mov a, b
orl a, #0x30 ; Convert units to ASCII
lcall ?WriteData; Send to LCD
ret

Change_8bit_Variable MAC
jb %0, %2
Wait_Milli_Seconds(#50) ; de-bounce
jb %0, %2
jnb %0, $
jb SHIFT_PB, skip%Mb
dec %1
sjmp skip%Ma
skip%Mb:
inc %1
skip%Ma:
ENDMAC

loadbyte mac
mov a, %0
movx @dptr, a
inc dptr
```

```
endmac

Save_Configuration:
    push IE ; Save the current state of bit EA in the stack
    clr EA ; Disable interrupts
    mov FCON, #0x08 ; Page Buffer Mapping Enabled (FPS = 1)
    mov dptr, #0x7f80 ; Last page of flash memory
    ; Save variables
    loadbyte(temp_soak) ; @0x7f80
    loadbyte(time_soak) ; @0x7f81
    loadbyte(temp_refl) ; @0x7f82
    loadbyte(time_refl) ; @0x7f83
    loadbyte(#0x55) ; First key value @0x7f84
    loadbyte(#0xAA) ; Second key value @0x7f85
    mov FCON, #0x00 ; Page Buffer Mapping Disabled (FPS = 0)
    orl EECON, #0b01000000 ; Enable auto-erase on next write sequence
    mov FCON, #0x50 ; Write trigger first byte
    mov FCON, #0xA0 ; Write trigger second byte
    ; CPU idles until writing of flash completes.
    mov FCON, #0x00 ; Page Buffer Mapping Disabled (FPS = 0)
    anl EECON, #0b10111111 ; Disable auto-erase
    pop IE ; Restore the state of bit EA from the stack
    ret


Initial_Message:  db 'TS:   TR:       T   ', 0
Line:          db 'ts:   tR:          ', 0

getbyte mac
clr a
movc a, @a+dptr
mov %0, a
inc dptr
Endmac
Load_Configuration:
    mov dptr, #0x7f84 ; First key value location.
    getbyte(R0) ; 0x7f84 should contain 0x55
    cjne R0, #0x55, Load_Defaults
    getbyte(R0) ; 0x7f85 should contain 0xAA
    cjne R0, #0xAA, Load_Defaults
    ; Keys are good.  Get stored values.
    mov dptr, #0x7f80
    getbyte(temp_soak) ; 0x7f80
    getbyte(time_soak) ; 0x7f81
    getbyte(temp_refl) ; 0x7f82
    getbyte(time_refl) ; 0x7f83
    ret

Load_Defaults:
mov temp_soak, #150
mov time_soak, #45
mov temp_refl, #225
mov time_refl, #30
ret
```

Appendix G : Source Code - Reading ADC Channel and PuTTy communication

Reading ADC Channel and sending chars to PuTTy

```
Send_BCD mac
     push ar0
     mov r0,%0
     lcall ?Send_BCD
     pop ar0
    endmac

?Send_BCD:
    push acc
    ;send most significant digit
    mov a, r0
    swap a
    anl a, #0fh
    orl a, #30h
    lcall putchar
    ;send least sigfig
    mov a, r0
    anl a ,#0fh
    orl a, #30h
    lcall putchar

    pop acc
    ret

  Read_ADC_Channel MAC
       mov b, #%0
       lcall _Read_ADC_Channel
       ENDMAC

 _READ_ADC_Channel:
  clr CE_ADC
  mov R0, #00000001B ; Start bit:1
  lcall DO_SPI_G
  mov a, b
  swap a
  anl a, #0F0H
  setb acc.7 ; Single mode (bit 7).
  mov R0, a
  lcall DO_SPI_G
  mov a, R1 ; R1 contains bits 8 and 9
  anl a, #00000011B  ; We need only the two least significant bits
  mov R7, a ; Save result high.
  mov R0, #55H ; It doesn't matter what we transmit...
  lcall DO_SPI_G
  mov a,R1
  mov R6,a
  setb CE_ADC
  ret
```

```
PLAY_SOUND MAC
    ; PLAY_SOUND(%0, %1, %2, %3, %4, %5) inputs will be hex numbers (diff segments of address)
    ; input has to automatically be in hex format with the #
    clr TR1 ; Stop Timer 1 ISR from playing previous request
    ; I think we'll want to remove the above instruction and include it right AFTER this has been called
    setb FLASH_CE
    clr SPEAKER

    clr FLASH_CE
    mov a, #READ_BYTES
    lcall Send_SPI

    mov a, %0

    lcall Send_SPI

    mov a, %1
    lcall Send_SPI

    mov a, %2
    lcall Send_SPI
    mov a, #0x00
    lcall Send_SPI

    mov w+2, %3
    mov w+1, %4
    mov w+0, %5

    setb SPEAKER
    setb TR1

ENDMAC
```

Appendix H : Source Code - Temperature Reading

Reading Temp and Sending to PuTTy

```
TEMP_JUNCTION:

    Read_ADC_Channel(0)
        mov x+0, R6
        mov x+1, R7
        mov x+2, #0
        mov x+3, #0

        load_y(964)
        lcall mul32
        load_y(1000)
        lcall div32
        load_y(22)
        lcall add32


        mov temp+0, x+0
        mov temp+1, x+1
        mov temp+2, x+2
        mov temp+3, x+3

        lcall hex2bcd
        jb one_second_flag, TEMP_JUNCTION2
        ret


TEMP_JUNCTION2:
        Send_BCD(bcd+2)
        Send_BCD(bcd+1)
        Send_BCD(bcd+0)

        mov a, #'\r'
        lcall putchar

        mov a, #'\n'
        lcall putchar
    clr one_second_flag
 ret
```

Appendix I : Source Code -Sound Functions

Functions for Playing Different Sounds

```
Delay:
        Wait_Milli_Seconds(#250)
        Wait_Milli_Seconds(#250)
        Wait_Milli_Seconds(#250)
        Wait_Milli_Seconds(#250)
        ret

    sound_zero:
    PLAY_SOUND(#0x02, #0x8C, #0xBD,#0x00, #0x3A,#0x98)
    lcall Delay
    ret

    sound_one:
    PLAY_SOUND(#0x02, #0xDC, #0xBB,#0x00, #0x3A,#0x98)
    lcall Delay
    ret

    sound_two:
    PLAY_SOUND(#0x03, #0x1D, #0xBC, #0x00, #0x3A, #0x98)
    lcall Delay
    ret

    sound_three:
    PLAY_SOUND(#0x03, #0x4E, #0xB3,#0x00, #0x3A,#0x98)
    lcall Delay
    ret

    sound_four:
    PLAY_SOUND(#0x03, #0x99, #0x4B,#0x00, #0x3A,#0x98)
    lcall Delay
    ret

    sound_five:
    PLAY_SOUND(#0x03, #0xE3, #0xE3,#0x00, #0x3A,#0x98)
    lcall Delay
    ret


    sound_six:
    PLAY_SOUND(#0x04, #0x2E, #0x7B,#0x00, #0x3A,#0x98)
    lcall Delay
    ret


    sound_seven:
    PLAY_SOUND(#0x04, #0x79, #0x13,#0x00, #0x3A,#0x98)
    lcall Delay
    ret

    sound_eight:
    PLAY_SOUND(#0x04, #0xC3, #0xAB,#0x00, #0x3A,#0x98)
    lcall Delay
    ret

    sound_nine:
    PLAY_SOUND(#0x05, #0x0E, #0x43,#0x00, #0x3A,#0x98)
    lcall Delay
    ret


    sound_ten:
    PLAY_SOUND(#0x05, #0x58, #0xDB,#0x00, #0x30,#0xFC)
    lcall Delay
    ret

    sound_eleven:
    PLAY_SOUND(#0x05, #0x8F, #0xD7, #0x00, #0x3A,#0x98)
```

```
lcall Delay
ret


sound_twelve:
PLAY_SOUND(#0x05, #0xDA, #0x6F,#0x00, #0x3A,#0x98)
lcall Delay
ret

sound_thirteen:
PLAY_SOUND(#0x06, #0x25, #0x07,#0x00, #0x3A,#0x98)
lcall Delay
ret


sound_fourteen:
PLAY_SOUND(#0x06, #0x6F, #0x9F,#0x00, #0x3A,#0x98)
lcall Delay
ret

sound_fifteen:
PLAY_SOUND(#0x06, #0xB8, #0x37,#0x00, #0x42,#0x68)
lcall Delay
ret


sound_sixteen:
PLAY_SOUND(#0x07, #0x0C, #0x9F,#0x00, #0x4E,#0x20)
lcall Delay
ret

sound_seventeen:
PLAY_SOUND(#0x07, #0x6A, #0xBF, #0x00, #0x4E,#0x20)
lcall Delay
ret


sound_eighteen:
PLAY_SOUND(#0x07, #0xC8, #0xDF,#0x00, #0x4E,#0x20)
lcall Delay
ret

sound_nineteen:
PLAY_SOUND(#0x08, #0x26, #0xFF,#0x00, #0x4E,#0x20)
lcall Delay
ret

sound_twenty:
PLAY_SOUND(#0x08, #0x85, #0x1F,#0x00, #0x4E,#0x20)
lcall Delay
ret

sound_thirty:
PLAY_SOUND(#0x08, #0xE3, #0x3F,#0x00, #0x3A,#0x98)
lcall Delay
ret


sound_forty:
PLAY_SOUND(#0x09, #0x1D, #0xD7,#0x00, #0x3A,#0x98)
lcall Delay
ret

sound_fifty:
PLAY_SOUND(#0x09, #0x68, #0x6F,#0x00, #0x3A,#0x98)
lcall Delay
ret

sound_sixty:
PLAY_SOUND(#0x09, #0xB3, #0x07,#0x00, #0x3A,#0x98)
```

```
lcall Delay
ret

sound_seventy:
PLAY_SOUND(#0x09, #0xFD, #0x9F,#0x00, #0x3A,#0x98)
lcall Delay
ret

sound_eighty:
PLAY_SOUND(#0x0A, #0x48, #0x37,#0x00, #0x3A,#0x98)
lcall Delay
ret

sound_ninety:
PLAY_SOUND(#0x0A, #0x92, #0xCF,#0x00, #0x3A,#0x98)
lcall Delay
ret

sound_100:
PLAY_SOUND(#0x0A, #0xF0, #0x3F, #0x00,#0x4E, #0x20)
ret

sound_200:
PLAY_SOUND(#0x0B, #0x2E, #0x54,#0x00, #0x4E,#0x20)
lcall Delay
ret



sound_state_zero:
PLAY_SOUND(#0x00, #0x00, #0x2D,#0x00, #0xD6,#0xD8)
lcall Delay
Wait_Milli_Seconds(#250)
Wait_Milli_Seconds(#250)
Wait_Milli_Seconds(#250)
ret



sound_state_one:
PLAY_SOUND(#0x01, #0x00, #0xFF,#0x00, #0x30,#0xFC)
lcall Delay
ret


sound_state_two:
PLAY_SOUND(#0x01, #0x33, #0xDF,#0x00, #0x36,#0xB0)
lcall Delay
ret


sound_state_three:
PLAY_SOUND(#0x01, #0x95, #0xD7,#0x00, #0x4E,#0x20)
lcall Delay
ret


sound_state_four:
PLAY_SOUND(#0x01, #0xF7, #0xCF,#0x00, #0x3A,#0x98)
lcall Delay
ret

sound_state_five:
PLAY_SOUND(#0x02, #0x33, #0x00,#0x00, #0x4E,#0x20)
lcall Delay
ret

```

Appendix H : Source Code - Timer, Flag and Pin Initializations in Main Loop

Initialization of timers, flags, and pins in MAIN LOOP

```
;--------------------------------;
; Main program. Includes hardware ;
; initialization and 'forever'       ;
; loop.                              ;
;--------------------------------;
main:
    ; Initialization
       mov SP, #0x7F

       mov P0M0, #0
       mov P0M1, #0

    ; Enable the timer and interrupts
       ;setb ET1  ; Enable timer 1 interrupt
    ; setb TR1 ; Timer 1 is only enabled to play stored sound


    ;mov counter, #0
    mov seconds, #0
    clr five_second_flag
    clr one_second_flag
    lcall Timer2_Init

    ;setb EA ; Enable interrupts


        lcall Load_Configuration

        ; enabling SPI communication
        lcall INIT_SPI_FLASH
        lcall INIT_SPI
        lcall InitSerialPort

        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        ; Configure P2.4, P2.5, P2.7 as open drain outputs (they need 1k pull-ups to 3.3V)
    orl P2M0, #0b_1011_0000
    orl P2M1, #0b_1011_0000
    setb MY_MISO  ; Configured as input
    setb FLASH_CE ; CS=1 for SPI flash memory
    clr MY_SCLK   ; Rest state of SCLK=0
    clr SPEAKER   ; Turn off speaker.

; Configure timer 1
    anl TMOD, #0x0F ; Clear the bits of timer 1 in TMOD
    orl TMOD, #0x10 ; Set timer 1 in 16-bit timer mode.  Don't change the bits of timer 0
    mov TH1, #high(TIMER1_RELOAD)
    mov TL1, #low(TIMER1_RELOAD)
    ; Set autoreload value
    mov RH1, #high(TIMER1_RELOAD)
    mov RL1, #low(TIMER1_RELOAD)

    ; Enable the timer and interrupts

    setb ET1  ; Enable timer 1 interrupt
    ; setb TR1 ; Timer 1 is only enabled to play stored sound


    ; Configure the DAC.  The DAC output we are using is P2.3, but P2.2 is also reserved.
    mov DADI, #0b_1010_0000 ; ACON=1
    mov DADC, #0b_0011_1010 ; Enabled, DAC mode, Left adjusted, CLK/4
    mov DADH, #0x80 ; Middle of scale
    mov DADL, #0
    orl DADC, #0b_0100_0000 ; Start DAC by GO/BSY=1
    check_DAC_init:
    mov a, DADC
    jb acc.6, check_DAC_init ; Wait for DAC to finish

    setb EA ; Enable interrupts

        lcall LCD_4BIT
    setb state0flag
    clr state1flag
    clr state2flag
    clr state3flag
    clr state4flag
    clr state5flag

    setb SHIFT_PB


    ; set default pwm output to 20 instead of 0
    mov pwm+0, #low(0)
    mov pwm+1, #high(0)

;;;;;;;;;;;;;;;;;;;;;;;;
```

Appendix I: Source Code - Variable Declaration

Printing Initial Values on LCD

```
Set_Cursor(1, 1)
Send_Constant_String(#Initial_Message)
Set_Cursor(2, 1)
Send_Constant_String(#Line)

;Display Variables
Set_Cursor(2, 4)
mov a, time_soak
lcall SendToLCD
 Set_Cursor(1, 4)
 mov a, temp_soak
lcall SendToLCD
Set_Cursor(2, 9)
 mov a, time_refl
lcall SendToLCD
Set_Cursor(1, 9)
 mov a, temp_refl
lcall SendToLCD
mov seconds, #0
sjmp loop
```

Appendix J: Source Code - PWM FSM

FSM for PWM Control

```
loop:

    Change_8bit_Variable(TEMP_SOAK_PB, temp_soak, loop_a)
    Set_Cursor(1, 4)
    mov a, temp_soak
    lcall SendToLCD
```

```
        lcall Save_Configuration
        sjmp loop_a

    loop_a:
        Change_8bit_Variable(TIME_SOAK_PB, time_soak, loop_b)
        Set_Cursor(2, 4)
        mov a, time_soak
        lcall SendToLCD
        lcall Save_Configuration
        sjmp loop_b

    loop_b:
        Change_8bit_Variable(TEMP_REFL_PB, temp_refl, loop_c)
        Set_Cursor(1, 11)
        mov a, temp_refl
        lcall SendToLCD
        lcall Save_Configuration
        sjmp loop_c

    loop_c:
        Change_8bit_Variable(TIME_REFL_PB, time_refl, loop_d)
        Set_Cursor(2, 11)
        mov a, time_refl
        lcall SendToLCD
        lcall Save_Configuration
        sjmp loop_d

    loop_d:
        Change_8bit_Variable(TIME_REFL_PB, time_refl, state0)
        Set_Cursor(2, 14)
        mov a, temp
        lcall SendToLCD
        lcall Save_Configuration
        sjmp state0

    state0:
        ;**set power to 0%**
        clr state5flag ; just in case,
        setb state0flag
        mov pwm+0, #low(0)
        mov pwm+1, #high(0)
        ;Wait_Milli_Seconds(#255)

        ljmp state_one
        state0_2:
            mov seconds, #0
        ;mov counter, #0

        lcall TEMP_JUNCTION
        clr DEBUG
        jb START, loop_leap ;START is pushbutton to start reflow
        Wait_Milli_Seconds(#50)    ; Debounce delay
        jb START, loop_leap
        jnb START, state1

    loop_leap:
            ljmp loop
    state1:
        clr state0flag
        setb state1flag
        ljmp state_one
        ;ljmp state_zero ;jumps to second finite state machine
        state1_2:
            mov pwm+0, #low(1000) ;CHANGE BACK TO 1000
        mov pwm+1, #high(1000)
        Wait_Milli_Seconds(#255)
        setb DEBUG
        clr a
        ;mov counter, #0 ;setting seconds to zero
        ;setb DEBUG
        lcall TEMP_JUNCTION
        clr c
        mov a, temp
        subb a, temp_soak
        jc state1_x
        mov seconds, #0
        sjmp state2
    state1_x:
        mov a, seconds
        xrl a, #60
        jz ABORT_CHECK
        jnz state1

    ABORT_CHECK:
        mov a, temp
        subb a, #50
        jc ABORT
        sjmp state1

    ABORT:
            clr state1flag
        ljmp state0

    state2:
        clr state1flag
        setb state2flag ;use this to increment counter in Timer ISR
        ljmp state_one
        ;mov seconds, #0
        state2_2:
           mov pwm+0, #low(200)
        mov pwm+1, #high(200)
        Wait_Milli_Seconds(#255)
        lcall TEMP_JUNCTION
        ;clr DEBUG
        ;jb state3flag, state3
        mov a, seconds
        xrl a, time_soak
        jz state3
        ;cjne a, time_soak, state2
        ;mov counter, #0
        sjmp state2

    state3:
        ;setb DEBUG
        clr state2flag
        setb state3flag
        ljmp state_one
        ;mov seconds, #0
        state3_2:
           mov pwm+0, #low(1000)
        mov pwm+1, #high(1000)
        Wait_Milli_Seconds(#255)
        lcall TEMP_JUNCTION
        mov a, temp
        clr c
        subb a, temp_refl
        jc state3
        mov seconds, #0; setting seconds to zero so that when we get to state 4 it starts counting
        sjmp state4

    state4:
        ;clr DEBUG
        clr state3flag
        setb state4flag
        ljmp state_one
        ;mov seconds, #0
        state4_2:
           mov pwm+0, #low(200)
        mov pwm+1, #high(200)
        Wait_Milli_Seconds(#255)
        lcall TEMP_JUNCTION
        mov a, seconds
        cjne a, time_refl, state4
        sjmp state5

    state5:
        ;setb DEBUG
        clr state4flag
        setb state5flag
        ljmp state_one
        ;mov seconds, #0
        state5_2:
            mov pwm+0, #low(0)
        mov pwm+1, #high(0)
        Wait_Milli_Seconds(#255)
        lcall TEMP_JUNCTION
        mov seconds, #0; setting seconds to zero
        mov a, temp
        cjne a, #59, state5
        ljmp loop
```

Appendix K : Source Code - Sound FSM

FSM for Sound Control

```

FSM2:
    ; function
    ;Finit staet machine for sound, set a flag for states then produce sound
    ;state_zero:
    ;jb START, state_zero ;START is pushbutton to start reflow
    ;Wait_Milli_Seconds(#50)    ; Debounce delay
    ;jb START, state_zero
    ;jnb START, state_one    ; if buttons is pressed go to state one

state_one:
    setb done_playing_flag
    ; compare temperature, if temp <100 -> state 5 else -> state 2
    jb five_second_flag, state_one_x
    ljmp state_seven

state_one_x:
        clr done_playing_flag
    jb state0flag, zero_state_s
    jb state1flag, one_state_s
    jb state2flag, two_state_s
    jb state3flag, three_state_s
    jb state4flag, four_state_s
    jb state5flag, five_state_s

    zero_state_s:
        lcall sound_state_zero
    sjmp after_saying_state
    one_state_s:
        lcall sound_state_one
    sjmp after_saying_state
    two_state_s:
        lcall sound_state_two
    sjmp after_saying_state
    three_state_s:
        lcall sound_state_three
    sjmp after_saying_state
    four_state_s:
        lcall sound_state_four
    sjmp after_saying_state
    five_state_s:
       lcall sound_state_five
    sjmp after_saying_state


    after_saying_state:
       setb done_playing_flag
    mov a, temp
    clr c
    subb a, #100
    jc state_five ;smaller than 100 to state 5
    ljmp state_two

state_two:
    ;temp larger than 100, play 100 or 200
        clr done_playing_flag
    mov a, temp
    mov b, #100
    div ab
    cjne a, #2, call_sound_100
    sjmp call_sound_200


    call_sound_200:
    lcall sound_200
    setb done_playing_flag
    ;jnb TR1, done_playing
    ljmp state_three


    call_sound_100:
    lcall sound_100
    setb done_playing_flag
    ljmp state_three


state_three:
    ; check if done playing sound
    jnb done_playing_flag, state_three
    ljmp state_five

; no state 4 lolsies

state_five:
    mov a, temp
    mov b, #100
    div ab
    ;remainder is b
    mov a, b
    clr c
    subb a, #20
    jc state_six ; <20 -> state 6
    ljmp state_eight ; >= 20 -> state 8

state_six:
    ;check all 0 to 20 and call sound
    ;jnb  five_second_flag, no_sound
    ;cjne a, #0, sound_zero_fun
    mov a, temp
    mov b, #100
    div ab
    ;remainder is b
    mov a, b

    xrl a, #0
    ljmp state_seven ;all we have left to say is "zero"
    clr done_playing_flag

    xrl a, #1
```

```
jz sound_one_fun
ljmp sound_two_fun_2
sound_one_fun:
lcall sound_one
ljmp state_five_b

sound_two_fun_2:
xrl a, #2
jz sound_two_fun
ljmp sound_three_fun_3
sound_two_fun:
lcall sound_two
ljmp state_five_b

sound_three_fun_3:
xrl a, #3
jz sound_three_fun
ljmp sound_four_fun_4
sound_three_fun:
lcall sound_three
ljmp state_five_b

sound_four_fun_4:
xrl a ,#4
jz sound_four_fun
ljmp sound_five_fun_5
sound_four_fun:
lcall sound_four
ljmp state_five_b

sound_five_fun_5:
xrl a ,#5
jz sound_five_fun
ljmp sound_six_fun_6
sound_five_fun:
lcall sound_five
ljmp state_five_b

sound_six_fun_6:
xrl a, #6
jz sound_six_fun
ljmp  sound_seven_fun_7
sound_six_fun:
lcall sound_six
ljmp state_five_b

sound_seven_fun_7:
xrl a, #7
jz sound_seven_fun
ljmp sound_eight_fun_8
sound_seven_fun:
lcall sound_seven
ljmp state_five_b

sound_eight_fun_8:
xrl a, #8
jz sound_eight_fun
ljmp sound_nine_fun_9
sound_eight_fun:
lcall sound_eight
ljmp state_five_b

sound_nine_fun_9:
xrl a, #9
jz sound_nine_fun
ljmp sound_ten_fun_10
sound_nine_fun:
lcall sound_nine
ljmp state_five_b

```

```
sound_ten_fun_10:
xrl a, #10
jz sound_ten_fun
ljmp sound_ele_fun_11
sound_ten_fun:
lcall sound_ten
ljmp state_five_b

sound_ele_fun_11:
xrl a ,#11
jz sound_eleven_fun
ljmp sound_tw_fun_12
sound_eleven_fun:
lcall sound_eleven
ljmp state_five_b

sound_tw_fun_12:
xrl a ,#12
jz sound_twelve_fun
ljmp sound_thirteen_fun_13
sound_twelve_fun:
lcall sound_twelve
ljmp state_five_b

sound_thirteen_fun_13:
xrl a, #13
jz sound_thirteen_fun
ljmp sound_fourteen_fun_14
sound_thirteen_fun:
lcall sound_thirteen
ljmp state_five_b

sound_fourteen_fun_14:
xrl a, #14
jz sound_fourteen_fun
ljmp sound_fifteen_fun_15
sound_fourteen_fun:
lcall sound_fourteen
ljmp state_five_b

sound_fifteen_fun_15:
xrl a, #15
jz sound_fifteen_fun
ljmp sound_sixteen_fun_16
sound_fifteen_fun:
lcall sound_fifteen
ljmp state_five_b

sound_sixteen_fun_16:
xrl a, #16
jz sound_sixteen_fun
ljmp sound_seventeen_fun_17
sound_sixteen_fun:
lcall sound_sixteen
ljmp state_five_b

sound_seventeen_fun_17:
xrl a, #17
jz sound_seventeen_fun
ljmp  sound_eteen_fun_18
sound_seventeen_fun:
lcall sound_seven
ljmp state_five_b

sound_eteen_fun_18:
xrl a ,#18
jz sound_eighteen_fun
ljmp sound_nteen_fun_19
sound_eighteen_fun:
lcall sound_eighteen
```

```
    ljmp state_five_b

    sound_nteen_fun_19:
    xrl a ,#19
    jz sound_nineteen_fun
    sound_nineteen_fun:
    lcall sound_nineteen
    ljmp state_five_b



    state_five_b:
    ;jnb TR1, done_playing02
    setb done_playing_flag
    ljmp state_seven

state_seven:
    jnb done_playing_flag, state_seven
    sjmp continue

    continue:
    ; this is where we incorporate jumping with flags ;check this
    clr five_second_flag
    jb state0flag, zero_state
Wait_Milli_Seconds(#50)
    jb state1flag, one_state
Wait_Milli_Seconds(#50)
    jb state2flag, two_state
Wait_Milli_Seconds(#50)
    jb state3flag, three_state
Wait_Milli_Seconds(#50)
    jb state4flag, four_state
Wait_Milli_Seconds(#50)
    jb state5flag, five_state


    zero_state:
       ljmp state0_2
    one_state:
       ljmp state1_2
    two_state:
       ljmp state2_2
    three_state:
       ljmp state3_2
    four_state:
       ljmp state4_2
    five_state:
       ljmp state5_2

state_eight:
    mov a, temp
    mov b, #100
    div ab
    ;remainder is b
    mov a, b
    ; sound 20-90 increment of 10
    mov b, #10
    div ab ; want quotient from here

    ;check 5 sec
    ;jnb  five_second_flag, no_sound03
    clr done_playing_flag

    ; remainder + 20
    xrl a , #2
    jz sound_twenty_fun
    ljmp sound_30_fun

    sound_twenty_fun:
    lcall sound_twenty
```

```
        ljmp state_eight_b

        sound_30_fun:
        xrl a, #3
        jz sound_thirty_fun
        ljmp  sound_40_fun
        sound_thirty_fun:
        lcall sound_thirty
        ljmp state_eight_b


        sound_40_fun:
        xrl a, #4
        jz sound_forty_fun
        ljmp  sound_50_fun
        sound_forty_fun:
        lcall sound_forty
        ljmp state_eight_b


        sound_50_fun:
        xrl a, #5
        jz sound_fifty_fun
        ljmp sound_60_fun
        sound_fifty_fun:
        lcall sound_fifty
        ljmp state_eight_b


        sound_60_fun:
        xrl a, #6
        jz sound_sixty_fun
        ljmp sound_70_fun
        sound_sixty_fun:
        lcall sound_sixty
        ljmp state_eight_b


        sound_70_fun:
        xrl a, #7
        jz sound_seventy_fun
        ljmp sound_80_fun
        sound_seventy_fun:
        lcall sound_seventy
        ljmp state_eight_b

        sound_80_fun:
        xrl a, #8
        jz sound_eighty_fun
        ljmp sound_90_fun
        sound_eighty_fun:
        lcall sound_eighty
        ljmp state_eight_b


        sound_90_fun:
        xrl a, #9
        jz sound_ninety_fun
        ljmp state_eight_b
        sound_ninety_fun:
        lcall sound_ninety
        ljmp state_eight_b


    state_eight_b:
        mov a,b ;mov remainder to a
        ;jnb TR1, do_not_play
        ;do_not_play: ljmp done_playing02
        setb done_playing_flag
        ljmp state_nine
```

```

state_nine:
    jnb done_playing_flag, state_nine
    ljmp state_ten

state_ten:
    clr done_playing_flag

    sound_one_fun_:
    xrl a, #1
    jz sound_one_fun01
    ljmp sound_2_fun_
    sound_one_fun01:
    lcall sound_one
    ljmp state_ten_b

    sound_2_fun_:
    xrl a, #2
    jz sound_two_fun01
    ljmp sound_3_fun_
    sound_two_fun01:
    lcall sound_two
    ljmp state_ten_b

    sound_3_fun_:
    xrl a, #3
    jz sound_three_fun01
    ljmp sound_4_fun_
    sound_three_fun01:
    lcall sound_three
    ljmp state_ten_b

    sound_4_fun_:
    xrl a, #4
    jz sound_four_fun01
    ljmp sound_5_fun_
    sound_four_fun01:
    lcall sound_four
    ljmp state_ten_b

    sound_5_fun_:
    xrl a, #5
    jz sound_five_fun01
    ljmp sound_6_fun_
    sound_five_fun01:
    lcall sound_five
    ljmp state_ten_b


    sound_6_fun_:
    xrl a, #6
    jz sound_six_fun01
    ljmp sound_7_fun_
    sound_six_fun01:
    lcall sound_six
    ljmp state_ten_b


    sound_7_fun_:
    xrl a, #7
    jz sound_seven_fun01
    ljmp sound_8_fun_
    sound_seven_fun01:
    lcall sound_seven
    ljmp state_ten_b


    sound_8_fun_:
    xrl a, #8
    jz sound_eight_fun01
    ljmp sound_9_fun_
    sound_eight_fun01:
    lcall sound_eight
    ljmp state_ten_b


    sound_9_fun_:
    xrl a, #9
    jz sound_nine_fun01
    ljmp state_ten_b
    sound_nine_fun01:
    lcall sound_nine
    ljmp state_ten_b

state_ten_b:
    setb done_playing_flag
    ljmp state_seven

END

;   :) the end
```

APPENDIX L : Figures

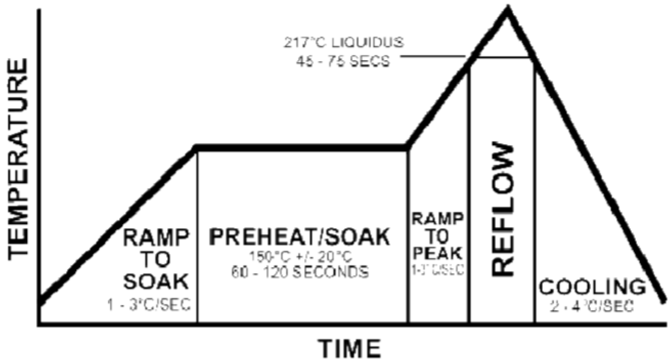


Figure 7. Reflow Process Graph[1].

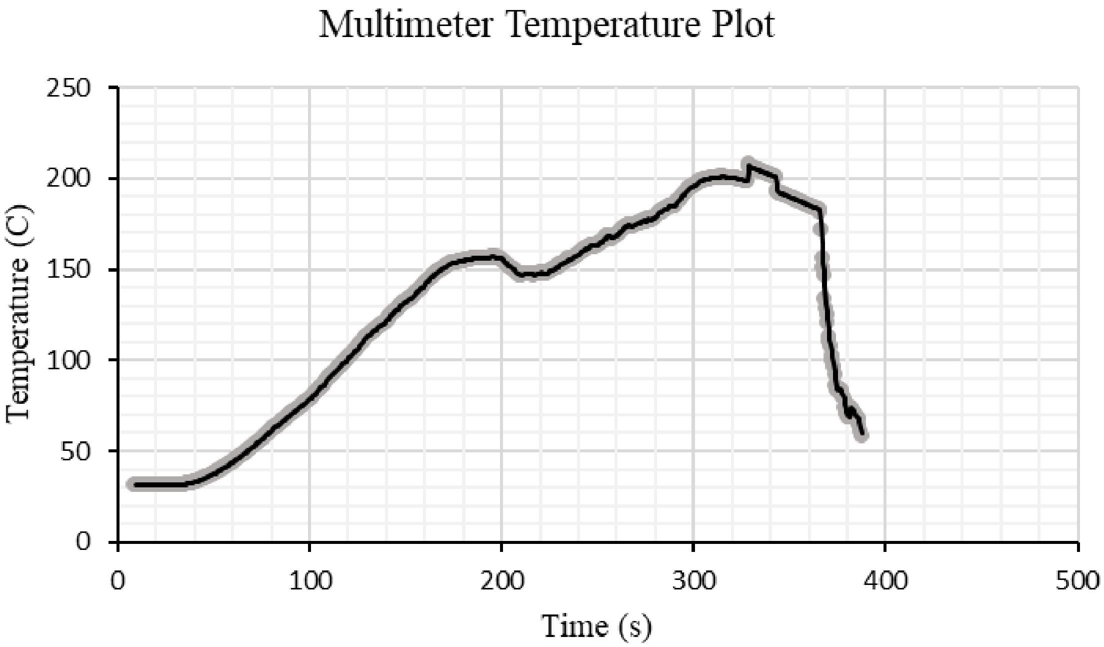


Figure 8. Multimeter Temperature Plot

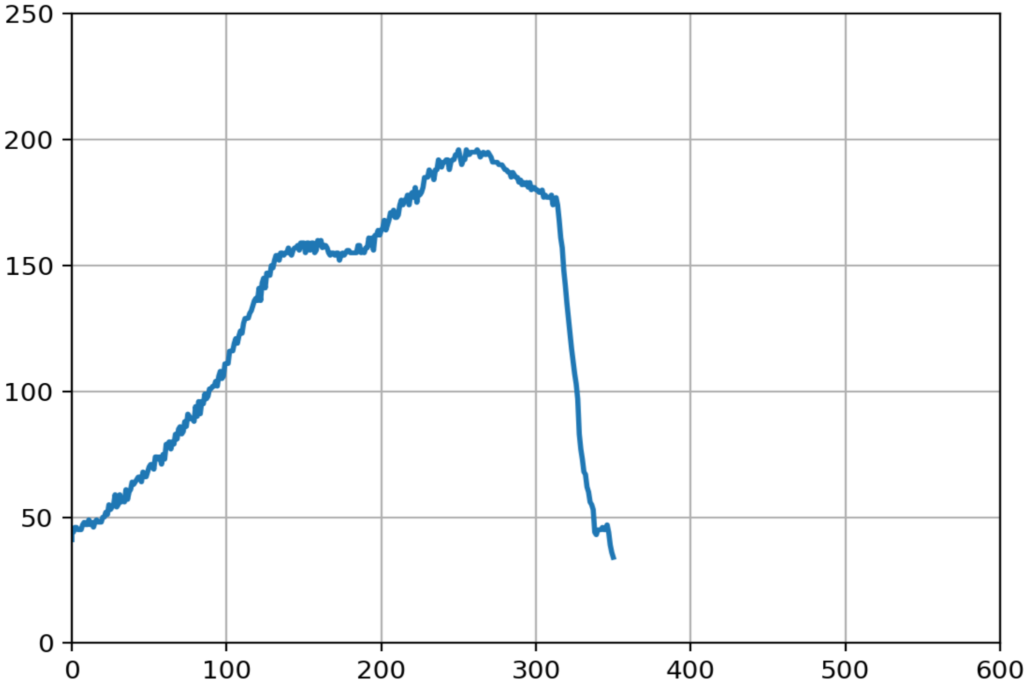


Figure 9. Temperature Plot from code values.