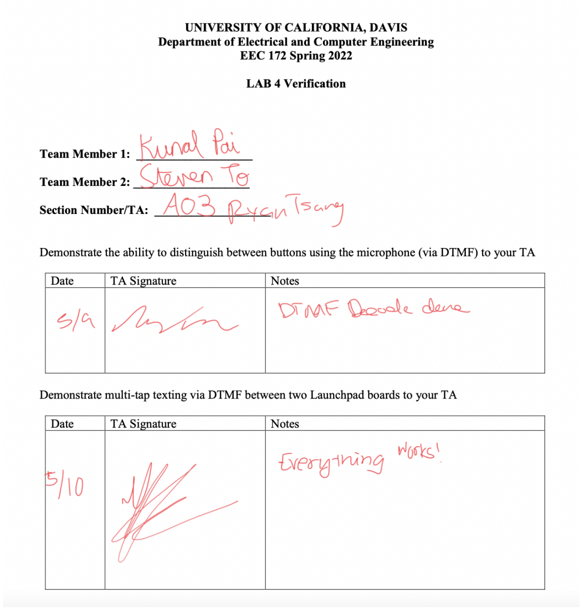

UNIVERSITY OF CALIFORNIA, DAVIS
Department of Electrical and Computer Engineering
EEC 172 Spring 2022

LAB 4 Verification

Team Member 1: Kunal Pai
Team Member 2: Steven To
Section Number/TA: A03 Ryan Tsang

Demonstrate the ability to distinguish between buttons using the microphone (via DTMF) to your TA

Date	TA Signature	Notes
5/9	[illegible]	DTMF Decode done

Demonstrate multi-tap texting via DTMF between two Launchpad boards to your TA

Date	TA Signature	Notes
5/10	[illegible]	Everything works!

Contribution:

Steven did the circuit configurations and the adjustment of the multi-tap texting to meet the scheme of Lab 4.

Kunal did the signal detection and decoding of the DTMF tones and implementation of board-to-board texting via UART.

Introduction:

The objective of this lab was very similar to that of lab 3. Rather than decoding the waveforms produced from an AT&T IR remote control, we had to decode dual-tone multi-frequency (DTMF) audio signals coming from our phones. To obtain these signals, a microphone had to be configured in a circuit to pick up sounds and produce corresponding analog signals. These analog signals could then be converted to digital signals using an ADC, which can then be analyzed to determine if a button was pressed and what specific button was pressed. After implementing the detection and decoding of these DTMF tones, the same board-to-board texting application from Lab 3 can be replicated with some slight adjustments to circuit configuration and software to accommodate for noise and a different signal processing scheme.

Background:

SPI is a serial communication protocol that allows the transmission of data between a controller and one or more peripheral devices. It involves the use of four data lines: MOSI which allows the controller to transmit data to the peripheral, MISO which allows the peripheral to transmit data to the controller, SCLK which transmits the clock signal from controller to peripheral, and CS which allows the controller to select which peripheral to communicate with.

UART is a serial and asynchronous communication protocol that allows for bidirectional transmission of data between two devices. For data to be transmitted, it is sent through the Tx pin from one device which is then received by the Rx pin of another device. Because UART is asynchronous, it does not require a clock to synchronize the transmission of data bits. Instead, it depends on start and stop bits to indicate when to start reading data and when to stop. The rate of data transfer is based on the baud rate which must be the same for both devices.

The OLED is a 1.5” display consisting of 128x128 RGB pixels, with each pixel having 16 bits of resolution to allow for high contrast and a wide range of vivid colors. It uses a SSD1351 driver chip that manages the display through SPI. Adafruit also provides the OLED with an API of functions that can be used to display patterns and text (Because it was originally written for Arduino, it needs to be ported to the CC3200 through the modification of low-level functions).

The Saleae logic analyzer is a tool that allows for the debugging and verification of waveforms and signals of digital circuits and systems. This is especially useful for ensuring that the data bits transmitted for SPI are correct.

The MAX9814 is a microphone amplifier with automatic gain control and low-noise microphone bias. It is a circuit component that captures DTMF tones from our phones and produces an analog signal depending on the tone.

The MCP3001 is a 10 bit analog-to-digital converter with an SPI serial interface. It is a circuit component that takes the analog signals produced from the MAX9814 and converts them into digital signals. These digital signals can be transmitted to the CC3200 using SPI ports MISO, CLK, and CS.

The LM1086 is a low dropout voltage regulator. It is a circuit component that separates the power sources for the OLED and the MAX9814 microphone, since the OLED produces significant noise which can corrupt the microphone's output.

DTMF tones are sounds generated by a phone when certain buttons are pressed. Each button can be distinguished by different DTMF tones consisting of two frequencies. Using something like Goertzel's algorithm, these tones can be detected and decoded to match with their respective commands.

Goals:

- To set up the LM1806 voltage regulator with capacitors to provide the OLED its own +3.3 V power source
- To set up the MAX9814 microphone accordingly to capture audio sounds and produce their respective analog signals
- To set up the MCP3001 ADC accordingly to convert analog signals to digital ones, which can be sent to the CC3200 using SPI
- To explore how Goertzel's algorithm works and use it to develop software that detects and decodes DTMF tones, with few adjustments made to the example code provided
    - 16 kHz sampling frequency
    - Collecting 410 samples
    - SPI bit rate of at least 400 kps
    - Using only fixed-point arithmetic for efficiency
    - Extracting the right bits from the MCP3001 output
    - Removing the DC-bias from the microphone signal before it is processed
- To interface the CC3200 launchpads with the MCP300 ADCs and OLED displays using SPI
- To connect two CC3200 launchpads using UART for bidirectional data transmission
- To develop an application that uses DTMF tones coming from a phone to compose text messages on the OLED that can be sent back and forth between two CC3200 launchpads

- The application must support multi-tap texting, text color change, spaces, and character deletion
- The transmission of text messages will be achieved through UART interrupts
- Incoming messages will appear at the bottom of the OLED, while messages in composition will appear at the top

Methodology:

Part 1:

To get the coefficient array, we put every value of the frequency array (*{697, 770, 852, 941, 1209, 1336, 1477, 1633}*) into this formula:

coeff[i] = (2 * cos (2 * π * (f_tone[i] / 16000.0))) * (1 << 14);

And thus, we got our coefficients as follows:

long int coeff[7] = {31548, 31281, 30951, 30556, 29144, 28361, 27409};

For part 1, we used the Goertzel algorithm as given in the lab manual, except, we changed it to accept only the coefficients, as the sample size and the sample array were declared globally. Also, we changed all the datatypes to long long, and typecasted the resultant power to long, and returned that value.

We used the post_test() function as is, but changed the row_col array to remove ‘A’, ‘B’, ‘C’ and ‘D’ because our DTMF dialer did not have these special keys. We also changed our bound for the column power to be 100000000, which we found experimentally.

We also used a transferSignal() function, which transfers the Chip Select using SPITransfer. After transferring, we return (data[0] << 5) | ((0xf8 & data[1]) >> 3).

We collected the signals in the sample array in a timer interrupt that activates every 5000 ticks, and on collecting 410, a flag variable is changed to one, indicating that the result can be decoded using post_test();. We also removed the DC bias by subtracting 372 from the value returned from the transferSignal function.

Part 2:

For part 2, we built upon the implementation of part 1, and used the Adafruit files to draw characters on the OLED. Firstly, we initialized all the clocks and the peripheral for SPI communication. Then, we defined global strings of the input sequence for every button. For example, “abc” for button 2, “def” for button 3 and so on. After that, we changed the function that decodes the input. We added a variable that stores the previous input and another that marks the index of the input sequence string. If the previous input and the current input are the same, the index increases by 1, which outputs the next character in the input sequence. To prevent out of bounds errors, we used a modulo arithmetic function so that it remains within the bounds of the string sequence. For the colors, we created a 2D array of unsigned integers using the color macros defined in the Adafruit files, and cycle through them when 1 is pressed, similar to the logic behind the button presses. The current color is stored in another variable and applied whenever printing of the character is supposed to happen. If a zero is pressed, since it is a space, we just concatenated a whitespace to the resultant string, and for delete, we moved back an index and changed the character to a null escape sequence (“\0”). We also have a Timer Interrupt running through, and if another button is not pressed within it, the character is locked in and the user can enter the next character.

For the UART communication, we initialized the pins for UART1 RX and TX and initialized the UART. We created an interrupt for UART that keeps accepting characters using the UARTCharGet function until they are available. Once it reaches a terminator, which we defined as ‘/0’, the bottom half of the screen is filled to black, and the message is either sent via UART or is printed out into the OLED. Similarly, on pressing LAST, a flag variable is changed to indicate that a message needs to get sent. While sending the message, we used the UARTCharPut function, which generates the interrupt in the first place.

Discussion:

In this lab, we were exposed to a lot of errors.

Firstly, we got an error for tBoolean not being defined. As a result, any ulStatus for our timer interrupts could not be initialized. We fixed this error by changing the order of the #include statements. Our hypothesis is that tBoolean is defined in ”hw_types.h” and therefore, should be included first to avoid any errors.

Later, we got a connection error, because the Code Composer Studio could not find a “target”. We theorized that the reason could be the PinMuxConfig() and therefore redefined it in the TI Pin Mux Tool a few times, but to no avail. Finally, we fixed the error by using the same pin_mux_config files from the previous lab, and it seemed to work.

Next, we found that SPI waveforms did not appear on the Logic Analyzer’s capture of the waveforms unless an erroneous printf statement was added in the implementation. We realized that we were disabling another Timer that did not exist, and as a result, the program was entering a fault and was hanging. Disabling the timer we were using for capturing the samples (at 5000 ticks) solved this bug.

On running the post_test() function, we noticed that our program was entering the FaultISR() due to an out-of-bounds index access of our power array. This happened due to garbage values being stored in rows and columns because the powers were negative, and therefore, could not be greater than 0, which was the initialized max_power value. This was because our power values were in the billions and were way larger than the bounds of a long variable. Changing the datatypes in the goertzel algorithm to long long and then typecasting the resultant power to long gave us positive power values.

Our OLED did not initialize initially, but we fixed that error by initializing the OLED before the UART. Due to some reason, our

UART1 was not working, but this was fixed by using another board.

Conclusion:

In this lab, we were exposed to the DTMF signals, and using the Goertzel Algorithm and Fourier transforms to extract the peaks and use the powers to identify the signals. We also learned more about UART in general, and how to initialize communication using its API. We used timer interrupts and the Timer_If API for the sample collection. We were exposed to using the SPI API again to communicate with the OLED. We also used the same OLED interface we built in the previous lab to achieve what we wanted in this lab.