Andrew Scott
E155
Lab 5
October 12, 2015

Lab 5: Digital Audio

Introduction

In this lab, I wrote a C program to play music. The way this works is that the code generates a square wave on a GPIO pin from the Raspberry Pi, and then that signal passes through an audio amplifier into a speaker that makes the appropriate sound.

Design and Testing Methodology

Most of the focus in this lab was writing the C code to control the Raspberry Pi, leaving a fairly simple breadboard circuit without too many design decisions to be made. I followed a diagram in the data sheet for the LM386 amplifier without resistors and capacitors to make the amplification circuit, but there was a really loud buzzing when I turned on the power. After adding a capacitor between the amplifier and the speaker, as well as a pull down resistor between the GPIO pin and the amplifier, the buzzing decreased to a manageable level. However, I later realized that the source of the buzzing was the fact that I was powering my Pi from a wall outlet and not from the same 5V supply that is powering the amp. After fixing this issue, the buzzing disappeared. I tested the circuit by running it, which showed that it was properly playing the music from the Pi.

For the software, there were several design decisions to be made. The decision of how to implement the functions to set up GPIO pins and then read and write to them was made easy due to the fact that the class worked them out together. The same can be said for the sleep function, which causes the program to wait and do nothing for a specified period of time. The one part of the lab that we didn't work out in class was how to generate a square wave on one of the GPIO pins (which would translate to a musical note when it went thorough the speaker). I decided to do this by writing a 1 to the GPIO pin, sleeping for half a cycle (according to the Hz of the note), and then writing a 0 to the GPIO pin and again sleeping for half a cycle. Doing this required knowing how many times to do the on/off cycle in order to achieve the desired length of the note, which I was able to calculate with the formula $\frac{Hz \times millis}{1000}$. This works because it calculates

$$\frac{cycles}{seconds} \times milliseconds \times \frac{1 second}{1000\ milliseconds} = cycles$$

I then used a for loop that ran for the calculated number of cycles, spending half of each cycle on and half of each cycle off. I tested this code by using some print statements to make sure that the number of cycles were being calculated correctly and then by running the code while hooked up to the speaker circuit, which showed that the code was producing the appropriate signals since the music sounded right.

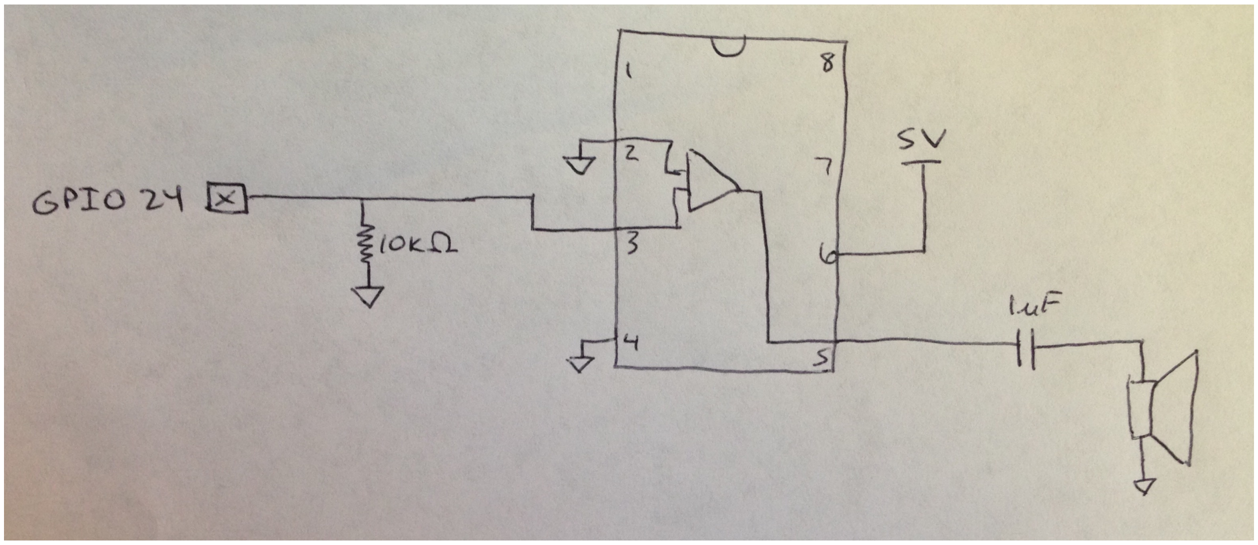
Results and Discussion

I was able to finish this entire lab. The music played clearly by the time I finished without any buzzing, which was a relief to finally get rid of.

Conclusion

I spent about 5 hours on this lab. I spent awhile trying to figure out the source of the buzzing which was frustrating, but overall everything went well.

Breadboard Schematic



C Code

```
#include <sys/mman.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

///////////////////////////////////////////////////////////////////////
// Constants
///////////////////////////////////////////////////////////////////////

#define LEDPIN 21

// GPIO FSEL Types
#define INPUT  0
#define OUTPUT 1
#define ALT0   4
#define ALT1   5
#define ALT2   6
#define ALT3   7
#define ALT4   3
#define ALT5   2

#define GPFSEL   ((volatile unsigned int *) (gpio + 0))
#define GPSET    ((volatile unsigned int *) (gpio + 7))
#define GPCLR    ((volatile unsigned int *) (gpio + 10))
#define GPLEV    ((volatile unsigned int *) (gpio + 13))
#define INPUT  0
#define OUTPUT 1

// Physical addresses
#define BCM2836_PERI_BASE       0x3F000000
#define GPIO_BASE               (BCM2836_PERI_BASE + 0x200000)
#define BLOCK_SIZE (4*1024)
#define SYS_TIMER_BASE          (BCM2836_PERI_BASE + 0x3000)

// Pointer that will be memory mapped when pioInit() is called
volatile unsigned int *gpio; //pointer to base of gpio

// Pointer that will be memory mapped when pTimerInit() is called
volatile unsigned int *sys_timer; //pointer to base of sys_timer

///////////////////////////////////////////////////////////////////////
```

```c
// Rasperry Pi Helper Functions
/////////////////////////////////////////////////////////////////////

void pioInit() {
    int  mem_fd;
    void *reg_map;

    // /dev/mem is a psuedo-driver for accessing memory in the Linux filesystem
    if ((mem_fd = open("/dev/mem", O_RDWR|O_SYNC) ) < 0) {
        printf("can't open /dev/mem \n");
        exit(-1);
    }

    reg_map = mmap(
         NULL,                   //Address at which to start local mapping (null means don't-care)
         BLOCK_SIZE,             //Size of mapped memory block
         PROT_READ|PROT_WRITE, // Enable both reading and writing to the mapped memory
         MAP_SHARED,             // This program does not have exclusive access to this memory
         mem_fd,                 // Map to /dev/mem
         GPIO_BASE);             // Offset to GPIO peripheral

    if (reg_map == MAP_FAILED) {
        printf("gpio mmap error %d\n", (int)reg_map);
        close(mem_fd);
        exit(-1);
    }

    gpio = (volatile unsigned *)reg_map;
}

void pTimerInit() {
    int  mem_fd;
    void *reg_map;

    // /dev/mem is a psuedo-driver for accessing memory in the Linux filesystem
    if ((mem_fd = open("/dev/mem", O_RDWR|O_SYNC) ) < 0) {
        printf("can't open /dev/mem \n");
        exit(-1);
    }

    reg_map = mmap(
         NULL,                   //Address at which to start local mapping (null means don't-care)
         BLOCK_SIZE,             //Size of mapped memory block
         PROT_READ|PROT_WRITE, // Enable both reading and writing to the mapped memory
         MAP_SHARED,             // This program does not have exclusive access to this memory
         mem_fd,                 // Map to /dev/mem
         SYS_TIMER_BASE);        // Offset to SYS_TIMER peripheral

    if (reg_map == MAP_FAILED) {
        printf("sys_timer mmap error %d\n", (int)reg_map);
        close(mem_fd);
        exit(-1);
    }

    sys_timer = (volatile unsigned *)reg_map;
}

/*
Function to set the mode of a pin. Function is the new GPFSEL value for the
specified pin.
*/
void pinMode(int pin, int function)
{
    unsigned int offset, shift;
    if (pin > 53 || pin < 0) {
        printf("bad pin, got pin %d\n", pin);
        return;
    } else if (function > 7 || function < 0) {
        printf("bad function, got function %d\n", function);
    }
    offset = pin / 10;
    shift = (pin % 10) * 3;

    // AND gpio[offset] with all 1s and function at the proper location
    gpio[offset] &= ~((~function & 7) << shift);
    // OR gpio[offset] with all 0s and function at the proper location
    gpio[offset] |= function << shift;
}

/*
Function to write val to a pin. Val can be either 0 or 1.
*/
void digitalWrite(int pin, int val)
{
    unsigned int set, clr;
    if (pin > 53 || pin < 0) {
        printf("bad pin, got pin %d\n", pin);
        return;
    }

    if (val){
        set = pin < 32 ? 7 : 8;              // select the proper set address
        gpio[set] = 0x1 << (pin % 32);       // write to the set address
    } else {
        clr = pin < 32 ? 10 : 11;            // select the proper clear address
        gpio[clr] = 0x1 << (pin % 32);       // write to the clear address
    }
}

int digitalRead(int pin)
{
    int out;
    if (pin > 53 || pin < 0) {
        printf("bad pin, got pin %d\n", pin);
        return 0;
    }

    // read from the proper address (depends on the pin number)
    if (pin < 32) {
        out = (gpio[13] >> pin) & 1;
    } else {
        out = (gpio[14] >> (pin - 32)) & 1;
    }
    return out;
}

void sleepMicros(int micros)
{
    if (micros == 0) {
        return;
    }
    sys_timer[4] = sys_timer[1] + micros;    // C1 = CLO + micros
    sys_timer[0] = 0b0010;                   // clear M1
    while (!!(sys_timer[0] & 0b0010) == 0);  // wait for M1 to go high again
}

void sleepMillis(int millis)
{
    sleepMicros(1000 * millis);     // sleep 1000 microseconds for each millisecond
}

//////////////////////////////////////////////////////////////////////
// Lab 5 Specific Code
//////////////////////////////////////////////////////////////////////

// Notes for happy Birthday
const int notes[][2] = {
    {294, 125},
    {294, 125},
    {330, 250},
    {294, 250},
    {392, 250},
    {370, 500},

    {294, 125},
    {294, 125},
    {330, 250},
    {294, 250},
    {440, 250},
    {392, 500},

    {294, 125},
    {294, 125},
    {587, 250},
    {494, 250},
    {392, 250},
    {370, 250},
    {330, 250},

    {523, 125},
    {523, 125},
    {494, 250},
    {392, 250},
    {440, 250},
    {392, 1000}
};

// Notes for fur elise: Pitch in Hz, duration in ms
/*const int notes[][2] = {
    {659,   125},
    {623,   125},
    {659,   125},
    {623,   125},
    {659,   125},
    {494,   125},
    {587,   125},
    {523,   125},
    {440,   250},
    {0, 125},
    {262,   125},
    {330,   125},
    {440,   125},
    {494,   250},
    {0, 125},
    {330,   125},
    {416,   125},
    {494,   125},
    {523,   250},
    {0, 125},
    {330,   125},
    {659,   125},
    {623,   125},
    {659,   125},
    {623,   125},
    {659,   125},
    {494,   125},
    {587,   125},
    {523,   125},
    {440,   250},
    {0, 125},
    {262,   125},
    {330,   125},
    {440,   125},
    {494,   250},
    {0, 125},
    {330,   125},
    {523,   125},
    {494,   125},
    {440,   250},
    {0, 125},
    {494,   125},
    {523,   125},
    {587,   125},
    {659,   375},
    {392,   125},
    {699,   125},
    {659,   125},
    {587,   375},
    {349,   125},
    {659,   125},
    {587,   125},
    {523,   375},
    {330,   125},
    {587,   125},
    {523,   125},
    {494,   250},
    {0, 125},
    {330,   125},
    {659,   125},
    {0, 250},
    {659,   125},
    {1319,  125},
    {0, 250},
    {623,   125},
    {659,   125},
    {0, 250},
    {623,   125},
    {659,   125},
    {623,   125},
    {659,   125},
    {623,   125},
    {659,   125},
    {494,   125},
    {587,   125},
    {523,   125},
    {440,   250},
    {0, 125},
    {262,   125},
    {330,   125},
    {440,   125},
    {494,   250},
    {0, 125},
    {330,   125},
    {416,   125},
    {494,   125},
    {523,   250},
    {0, 125},
    {330,   125},
    {659,   125},
    {623,   125},
    {659,   125},
    {623,   125},
    {659,   125},
    {494,   125},
    {587,   125},
    {523,   125},
    {440,   250},
    {0, 125},
    {262,   125},
    {330,   125},
    {440,   125},
    {494,   250},
    {0, 125},
    {330,   125},
    {523,   125},
    {494,   125},
        {440,   500},
        {0, 0}};*/

    void playNote(int hz, int millis)
    {
        if (hz == 0) {
            sleepMillis(millis);
        } else {
            unsigned i;
            unsigned cycles = hz * millis / 1000;   // number of cycles to achive desired note length
            unsigned cycleLength = (unsigned) (1000000 / (double) hz);      // cycle length in microseconds
            for (i = 0; i < cycles; i++) {
                // generate a square wave by going high for half a cycle then low for half a cycle
                digitalWrite(24, 1);
                sleepMicros(cycleLength / 2);
                digitalWrite(24, 0);
                sleepMicros(cycleLength / 2);
            }
        }
    }

    int main()
    {
        pioInit();
        pTimerInit();
        pinMode(24, OUTPUT);
        unsigned i;
        for (i = 0; i < sizeof(notes) / sizeof(notes[0]); i++) {
            playNote(notes[i][0], notes[i][1]);
        }
        return 0;
    }
```