Individually Programmable LEDs:
Creating an API to connect to the SP108E LED controller and send custom bytes

Team members

Samarth, Mihika, Sritan, Vishal, Daniel

Machine Learning II, Period 6
March 13, 2024

1 Introduction

1.1 Problem

While our classmates worked on code to triangulate the 3D locations of LEDs strung on a Christmas tree, we worked to find a way to individually program each of the LEDs, using our classmate's commands to directly control the lighting patterns of the LEDs. Our goal was to create an integrated system that could receive code as input and accordingly change the colors of the LEDs to match the specified commands. We wanted to create a system in which each of the lights could be individually controlled, at the behest of the user.

1.2 Equipment Information

Going into our project, we had an array of possible controllable LEDs that we could choose from. Doing research online, we came across a video by the YouTube channel Stand-Up Maths, which tackled the same problem we were currently faced with. The only problem with this setup was the use of the Raspberry Pi, which conflicted with our interests of keeping our project minimally integrated with external devices. Doing more research, we found that the lights used in the video showed promise of controllability with a portable WiFi controller, so we decided to follow through with these lights.

Our setup consisted of the WS2811 LED lights along with a SP108E WiFi controller. Following is a description of each. The descriptions are derived from the following manuals for the WS2811 and SP108E.

The WS2811 LED lights are a type of controllable light which come in clusters of 100. Each cluster can be connected to another cluster through a three-pin port, enabling control of multiple clusters simultaneously. There are three relevant channels which the LED clusters use to interact with one another. The first two are the GND and VCC, while the third is a data pin that enables the transfer of instructions through the lights in the system. These LED Strips operate on 12V.

The SP108E WiFi controller is a controller that can control the lights through wireless instructions sent by a computer. The SP108E has a maximum voltage of 24V, so there is a necessity for a voltage adapter if working with LEDs of 5V or less (which is not the case with the WS2811 LEDs). There are a total of six ports on the SP108E, VCC (2), CLK (1), DAT (1), and GND (2). By connecting the GND, VCC and DAT ports using the corresponding wires on the WS2811 LED lights, we can connect and send instructions to the lights. To effectively connect these wires, we borrowed wire strippers from the electronics lab and screwed the wires in so that a proper connection could be established.

Sending instructions to the controller can be done in a series of ways. The WiFi controller supports two different modes, a routing mode (AP) and LAN connection mode (STA). First, we must connect to the WiFi network associated with the SP108E controller and connect with the password "12345678". Now, the most simple method to send instructions is to use the LED Shop App that comes with the controller. With this app, it is possible to send a variety of instructions including changing the colors, changing the brightness, and displaying preset patterns. Despite this functionality, we were not able to find a way to make custom patterns on the LEDs, or reverse engineer the application to allow for custom control.

Connecting to the application via an external device is a more demanding task. First, we must connect to the IP address and open port of the WiFi controller via a coding interface, (for example, the socket library in Python). The IP address associated with the SP108E WiFi controller is 192.168.4.1, and the port number is 8189. From here, we can directly send instructions to the WiFi controller, the format of which we will cover in a later section.

1.3 Methodology

We used the WS2811 LED strip lights [1], which come in groups of 100 lights each and include 3 input channels. We also used the SP108E LED WiFi controller. This WiFi controller receives commands in the form of a byte array, sending signals to each of the LEDs based on the bytes it receives. The process of our work can be broken down into the following steps:

1. Controlling the LEDs with the built-in LED Shop mobile app
2. Connecting to the WiFi controller and pushing code to the LEDs
3. Determining the format of the byte commands to the LEDs
4. Developing an API to send byte commands to the LEDs

We will further elaborate on our progress, with respect to each of these steps, in the later sections. In the later sections, we will describe our design process, challenges faced, conclusions drawn, and ideas for future work.

2 Our Process

We started off by first trying to get the lights working. We figured out that we needed a power adapter, which we ended up getting from the Robotics lab, and connected the WiFi controller to the LED Shop mobile app.

However, connecting to the WiFi adapter using anything but the app turned out to be a struggle. We found multiple cases of online documentation of connecting to a WS2811 controller, but none of them seemed to work until we found a GitHub repository [2] with PHP code that managed to connect to the lights. This code was often buggy though, and we would need to run it multiple times to get a secure connection. In addition, the code seemed to have trouble doing anything other than changing the brightness; we were not able to come close to programming any light individually.

We ended up finding another GitHub Repository [3] that could individually program the lights, but it was done based off a screenshot taken with Linux tools and failed to produce any novel functionality outside of what we could already accomplish via the LED Shop App. Through reading these two GitHub repositories, we learned that the WS2811 can be connected using socket in python, and uses a certain byte format that the aforementioned GitHub reverse-engineered to transmit data. Using this knowledge, and after many hiccups, we managed to create python code to individually program lights to our liking.

One problem we encountered was that the patterns would repeat every 60 lights, which we found was the default size per segment, despite us sending data for many more lights.

Furthermore, it was unclear (and from our tests, convoluted) how it was choosing which subsection of the information we sent to control those lights. We found the byte commands to modify the number of lights per segment, however, the computer would simply disconnect. It turns out, we had to wait for responses from the computer, and for the commands that did not include responses, we had to add a delay. We managed to modify the number of lights per segment up to 300. This was because the LED Shop App sent packets of 900 bytes and the wifi controller refused to accept more than that amount, so 3 bytes per LED meant 300 individually programmable LEDs per controller.

Figure 1: Single bunch of WS2811 LEDs

3 Results and Conclusions

We were finally able to construct our API. We made use of `commands.py`, a file found in Hamish Coleman's GitHub repository. This is a video in which we demonstrate an alternating blue and green pattern on the RGB LED strip. Currently, we are able to control up to 300 lights using our WiFi controller. Furthermore, we have access to various commands, enabling us to change the brightness, color, and various other properties pertaining to each LED on the strip.

In the future, a valuable area to continue exploring would be the configuration of more than 300 LEDs so that we can make our API more generalizable, perhaps by integrating both wifi controllers together. More importantly, we are now able to develop a user-facing application (targeted towards fellow students in Dr. Gabor's ML class) in which users are able to send instructions for LED patterns in an intuitive manner (beyond our API). Our API can also be integrated with the LED triangulation algorithms built by our peers. As a final product, we envision an app where users can string these LEDs onto a tree, take a few calibration photos to triangulate the 3D position of each light, and immediately be able to run both built in and custom patterns based on the 3D positioning of the lights on the tree.

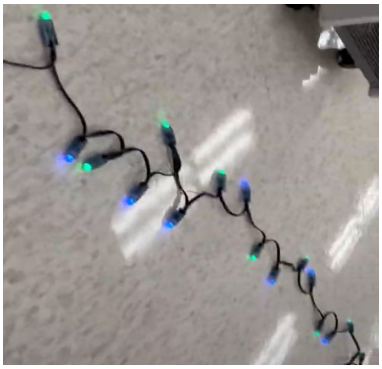

Figure 2: Custom alternating pattern which we set up with our API.

4 Acknowledgments

We would like to thank Dr. Peter Csaba Gabor for his guidance and mentorship throughout this project, as well as the resources he provided. We are also grateful to the robotics and engineering labs for providing us with necessary hardware and equipment.

5 Appendices

5.1 Appendix A: Commands File

```
CMD_CUSTOM_EFFECT = 0x02
CMD_SPEED = 0x03
CMD_MODE_AUTO = 0x06
CMD_CUSTOM_DELETE = 0x07
CMD_WHITE_BRIGHTNESS = 0x08
CMD_SYNC = 0x10
CMD_SET_DEVICE_NAME = 0x14
CMD_SET_DEVICE_PASSWORD = 0x16
CMD_SET_IC_MODEL = 0x1c
CMD_GET_RECORD_NUM = 0x20
CMD_COLOR = 0x22
CMD_CUSTOM_PREVIEW = 0x24
CMD_CHANGE_PAGE = 0x25
CMD_BRIGHTNESS = 0x2a
CMD_MODE_CHANGE = 0x2c
CMD_DOT_COUNT = 0x2d
CMD_SEC_COUNT = 0x2e
CMD_CHECK_DEVICE_IS_COOL = 0x2f
CMD_SET_RGB_SEQ = 0x3c
CMD_CUSTOM_RECODE = 0x4c
CMD_GET_DEVICE_NAME = 0x77
CMD_SET_DEVICE_TO_AP_MODE = 0x88
CMD_TOGGLE_LAMP = 0xaa
CMD_CHECK_DEVICE = 0xd5

# if we know this command, record if we expect a response
response = {
    CMD_BRIGHTNESS: False,
    CMD_CHECK_DEVICE: True,
    CMD_COLOR: False,
    CMD_DOT_COUNT: False,
    CMD_GET_DEVICE_NAME: True,
    CMD_MODE_AUTO: False,
    CMD_MODE_CHANGE: False,
    CMD_SEC_COUNT: False,
    CMD_SET_IC_MODEL: False,
    CMD_SPEED: False,
    CMD_SYNC: True,
}

CMD_FRAME_START = 0x38
CMD_FRAME_END = 0x83
```

```
# These modes all have a single color, set by CMD_COLOR
MODE_METEOR = 205
MODE_BREATHING = 206
MODE_STACK = 207
MODE_FLOW = 208
MODE_WAVE = 209
MODE_FLASH = 210
MODE_STATIC = 211
MODE_CATCHUP = 212
MODE_CUSTOM_EFFECT = 219

# Auto sequence through all the multi-color modes
MODE_AUTO = 0xfc

# TODO
#IC_MODEL_xxx = yyy # noqa


def frame(cmd, data):
    """Return the bytes needed for this command packet"""
    if data is None:
        data = b'\x00\x00\x00'
    elif len(data) < 3:
        data += bytes(3-len(data))
    elif len(data) > 3:
        raise ValueError("data length max is 3")

    return bytes([CMD_FRAME_START]) + data + bytes([cmd, CMD_FRAME_END])


def _call0(cmd):
    """Generic command packet with no parameters"""
    return frame(cmd, None)


def _call1(cmd, param1):
    """Generic command packet with one parameter"""
    return frame(cmd, bytes([param1]))


def speed(speed):
    """Set the speed of the programmed effect"""
    return _call1(CMD_SPEED, speed)


def get_device_name():
    """Request the current device name"""
```

```
    return _call0(CMD_GET_DEVICE_NAME)


def check_device(challenge):
    """Request a device check"""
    return frame(CMD_CHECK_DEVICE, challenge.to_bytes(3, 'little'))


def mode_change(mode):
    """Request a display pattern mode change"""
    # I dont know why they didnt let you simply MODE_CHANGE to MODE_AUTO
    if mode == MODE_AUTO:
        return _call0(CMD_MODE_AUTO)

    # TODO
    # - the screen freezes when given an invalid mode, maybe validate here?

    return _call1(CMD_MODE_CHANGE, mode)


def sync():
    """Request a status response"""
    return _call0(CMD_SYNC)


def set_ic_model(model):
    """Set which LED protocol is needed"""
    # TODO
    # - the screen freezes when given an invalid model, maybe validate here?

    return _call1(CMD_SET_IC_MODEL, model)


def color(rgb):
    """Set the color to be used for the single-color patterns"""
    return frame(CMD_COLOR, rgb.bytes)


def brightness(value):
    """Set the color to be used for the single-color patterns"""
    return _call1(CMD_BRIGHTNESS, value)


def dot_count(value):
    """configure the number of pixels in each segment"""

    # TODO
```

```
    # - range is 1 - 0x697, outside of which it transparently resets to 0x32
    # - simple testing suggests that 300 pixels is the max
    # should we check value here?

    return frame(CMD_DOT_COUNT, value.to_bytes(2, 'little'))


def sec_count(value):
    """configure the number of sections in the display"""
    return frame(CMD_SEC_COUNT, value.to_bytes(2, 'little'))

def rgb_ordering(value):
    """configure the rgb ordering [0-5]. Untested."""
    return _call1(CMD_SET_RGB_SEQ, value)
```

5.2 Appendix B: Final Lights API

```
### LIGHT CONTROLLER API ###

import socket
import time
from timeit import default_timer
import commands


class LightController:
    def __init__(self, lights_per_segment=300, num_segments=2048):
        """Initialize the connnection with the SP108E."""

        self.host_ip = "192.168.4.1" # might need to change the last digit
        self.port = 8189
        self.s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.s.connect((self.host_ip, self.port))
        self.n = min(300, lights_per_segment)
        self.num_segments = min(2048 // self.n, num_segments)

        MIN_SLEEP = 0.1

        self.s.send(commands.sec_count(self.num_segments))
        time.sleep(MIN_SLEEP)

        self.s.send(commands.dot_count(self.n))
        time.sleep(MIN_SLEEP)

        self.s.send(commands.rgb_ordering(0))
        time.sleep(MIN_SLEEP)
```

```
        self.s.send(b'\x38\x00\x00\x00\x24\x83')
        self.s.recv(10)

        self.bytes = bytearray(self.n*3)
        for i in range(self.n*3):
            self.bytes[i] = 0

        self.send_colors()

    def set(self, i, r, g, b):
        """Set the ith light to a desired r g b value. You need to call
           send_colors to have the change reflect onto the lights."""

        self.bytes[3*i] = r
        self.bytes[3*i + 1] = g
        self.bytes[3*i + 2] = b

    def set_frame(self, colors):
        """Set the first len(colors) lights to desired colors. You need to call
           send_colors to have the change reflect onto the lights."""

        if (len(colors) > self.n):
            colors = colors[:self.n]

        for i, c in enumerate(colors):
            self.bytes[3*i] = c[0]
            self.bytes[3*i + 1] = c[1]
            self.bytes[3*i + 2] = c[2]

    def send_colors(self, colors=None, delay=0.0):
        """Send the lights information to the controllers."""

        start_time = default_timer()

        if colors is not None:
            self.set_frame(colors)

        self.s.send(self.bytes)
        res = self.s.recv(10)

        end_time = default_timer()
        tot_time = end_time - start_time

        time.sleep(max(0, delay - tot_time))

        return res
```

```
### EXAMPLE USAGE ###


if __name__ == "__main__":
    lc = LightController(250, 3)

    frame1 = [[255, 0, 0], [0, 255, 0]]*10
    frame2 = [[0, 255, 0], [0, 0, 255]]*10
    frame3 = [[0, 0, 255], [255, 0, 0]]*10

    while True:
        lc.set_frame(frame1)
        res = lc.send_colors()

        lc.set_frame(frame2)
        res = lc.send_colors()

        lc.set_frame(frame3)
        res = lc.send_colors()
```

References

[1] SP108E WiFi LED Controller Operating Instructions, 2021. URL `https://ledlightinghut.com/files/sp108e-instructions.pdf`. Online; last accessed 13 March 2024.

[2] H. El-sewify. SP108E Controller GitHub, 2019. Online; last accessed 13 March 2024.

[3] H. Coleman. LED SP108E GitHub, 2021. Online; last accessed 13 March 2024.