Glowpucks

By Araam Zaremehrjardi

GitHub Repository: https://github.com/araamz/GlowPucks

Overview

The project will go over the following topics of the Glowpuck Device being overall methodology, technology integration, and conclusions drawn from development of the embedded product.

Abstract

Status	Feature Category	Feature Description
Completed	Category 1	LED Lighting for Drivers
Modified	Category 1	Bluetooth Communication
Completed	Category 1	PIR Motion Sensor
Completed	Category 2	Lighting Control System Application
Completed	Category 2	Communication Software
Unfinished	Category 3	Enhanced Visual Interfacing for Drivers with LED Screen

Table 1: *List of features progress that were denoted within the initial project report.*

Purpose

Glow Pucks are meant to harness the potential that conventional street reflectors normally have while being redesigned as an embedded system with LED lighting and WiFi connectivity. The Glow Puck have two main features of being active features such as Illumination Interfacing and passive features like Vehicle Awareness sensory. Active features that directly interface with drivers are the lighting effects that modify the color and behavior of the Glow Puck and a LED screen that allow bitmaps be displayed upon the road device itself.

Goals

1. Create a device that can visually communicate to drivers.

2. Enhance changing road conditions to allow drivers to best utilize traffic options.

3. Analyze road usage via motion sensing used by applicable parties.

Design - Hardware Overview

Type	Device Name	Feature Usage
MCU	Arduino UNO	Used to control and use peripheral devices.
MCU	Espressif ESP32C3	Used as a MQTT Client for UNO.
General Purpose Computer	Raspberry Pi 3 Model B	Used as a MQTT Broker and Access Point to orchestrate published and subscribed messages.
Peripheral	Mini Basic PIR Sensor – BL412	Used to sense motion from moving vehicles.
Peripheral	Adafruit NeoPixel Ring – 16 x 5050 RGB LED	Used to provide an illumination interface to drivers.
Peripheral	Red LED	Used to signify errors relating to operation of either TCP transport or MQTT transport.
Peripheral	Blue LED	Used to signify that the ESP32C3 connected to the MQTT Broker.
Peripheral	Green LED	Used to signify that the ESP32C3 connect via TCP to the MQTT Broker.
Peripheral	Arduino UNO BUILTIN LED	Used to signify that the device is detecting motion.
Peripheral	Arduino UNO EEPROM	Used to save the number of cars detected even when powered off.

Table 2: *List of devices used to accomplish the purpose of the project.*

Design - Hardware Design

Glowpuck Embedded Design Notes

1. The communication between the ESP32C3 and UNO requires a voltage divider for transmission of data from UNO (TX – 5V) to ESP32C3 (RX – 3.3V).
2. UNO is required for to drive Adafruit NeoPixel Ring due to requiring a 5V signal. The ESP32C3 can only output 3.3V signals.
3. Adafruit NeoPixel Ring and PIR Motion Sensor is connected to UNO.
4. LED Lights are used by ESP32C3 to denote TCP Transport Status.

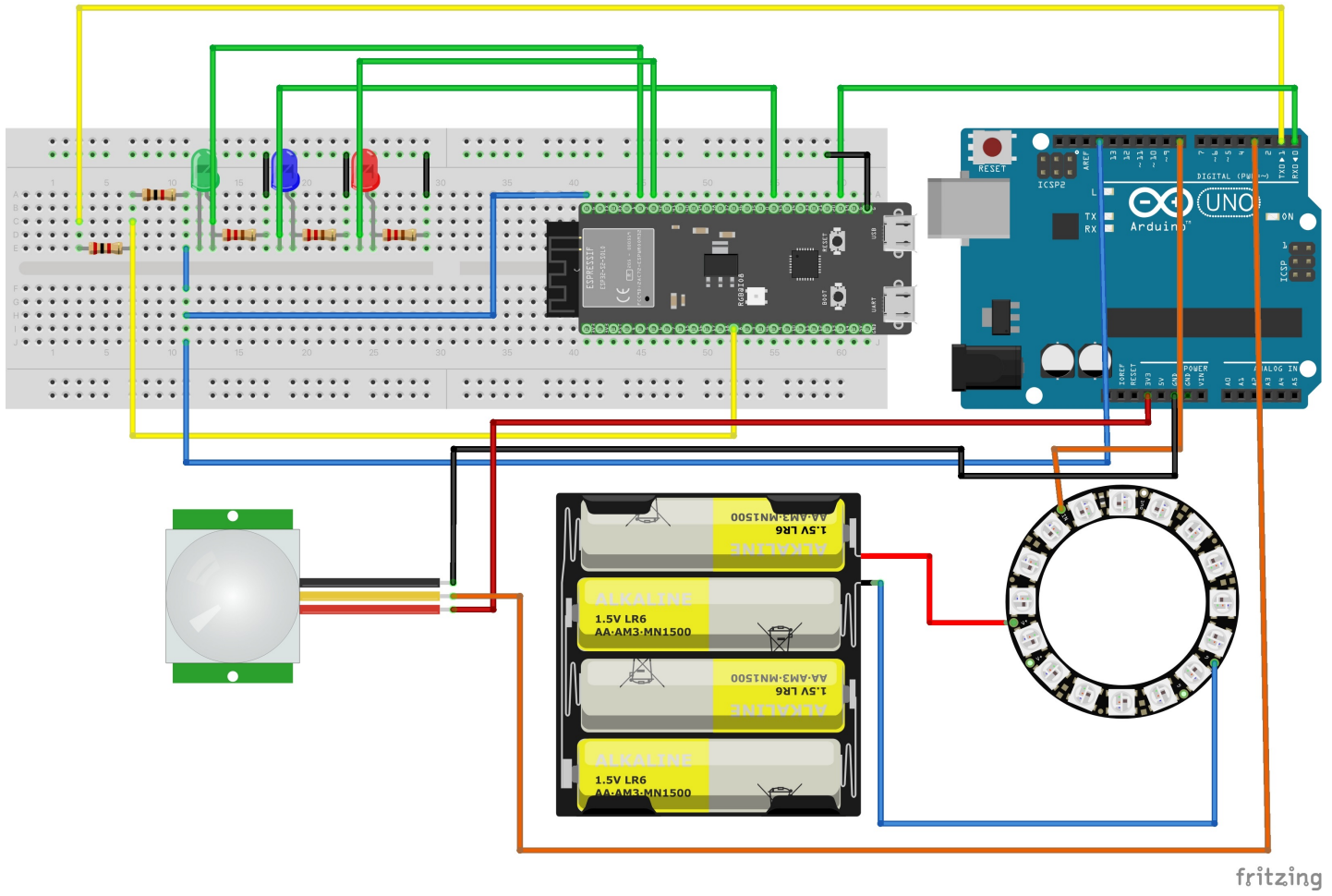


Figure 1: Configuration of the embedded system of a single Glowpuck. Diagram not fully accurate due to the ESP32C3 being substituted for ESP32S2. GPIO Pins still correlate to real circuit.

Design – ESP32C3 Pin Layout

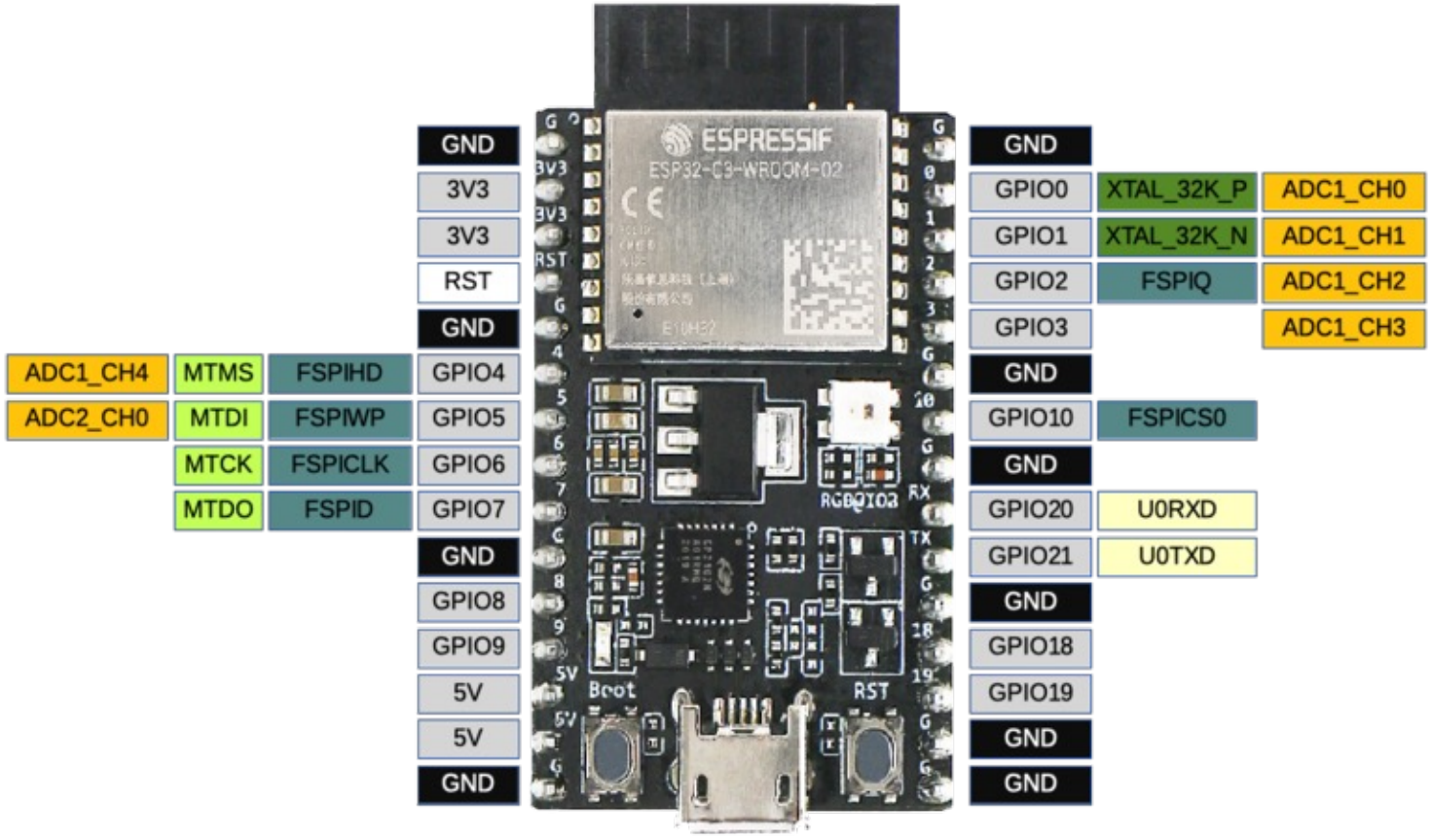


Figure 2: ESP32C3 Pin Layout.

Design – ESP32C3 Data Pin Usage

PIN #	Connected Peripheral	Peripheral Usage
GPIO 2	Red LED	Used for indication of TCP Transport error.
GPIO 0	Blue LED	Used for successful connection to MQTT Broker on Raspberry Pi.
GPIO 1	Green LED	Used for successful connection to MQTT Broker Access Point on Raspberry Pi.
GPIO 19	UART 1 TXD	Used to transmit serial data to UNO UART 0 RXD pin.
GPIO 18	UART 1 RXD	Used to receive serial data from UNO UART 0 TXD pin.

Table 3: *Description of peripherals connected to each pin on the ESP32C3.*

Design – UNO Pin Layout

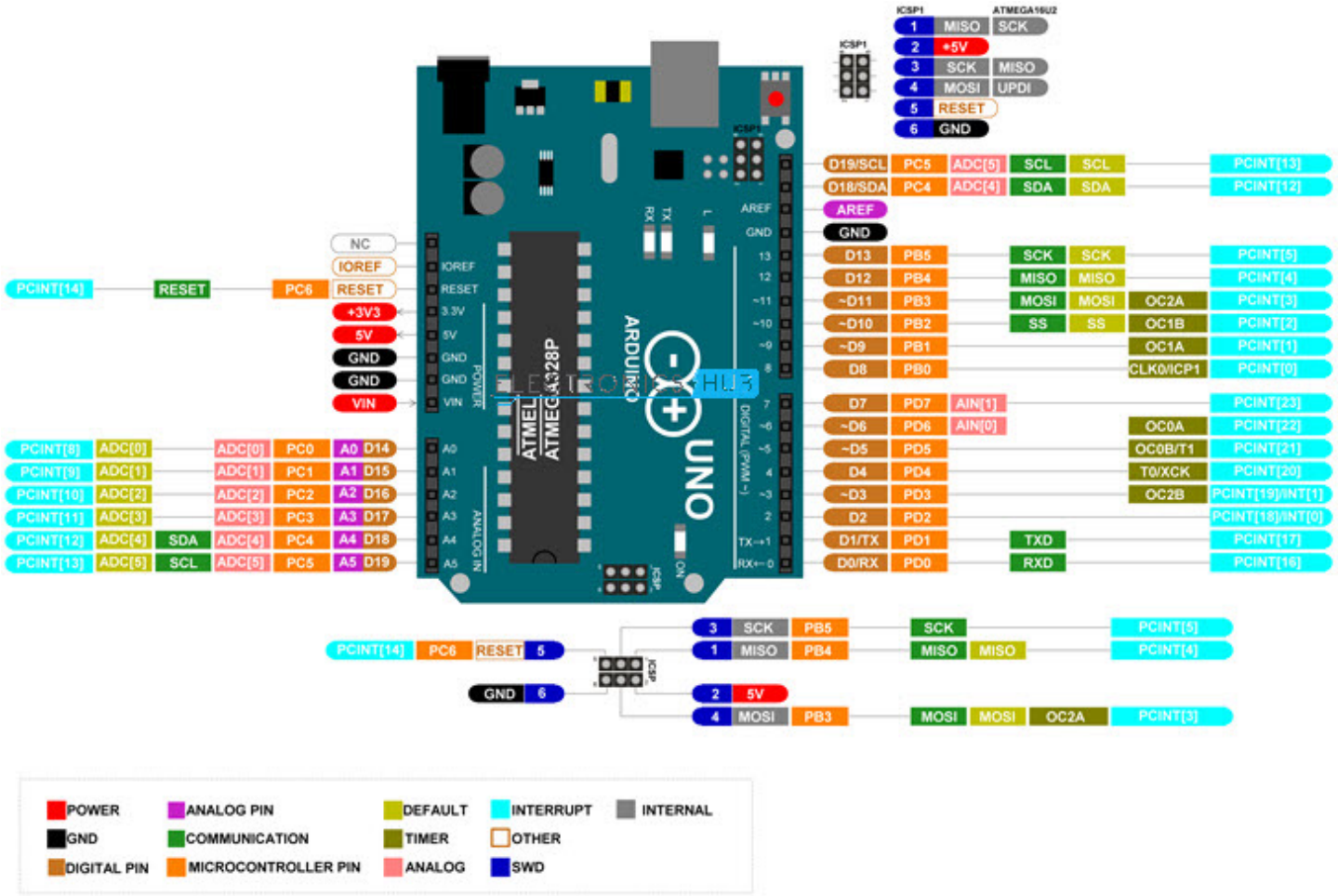


Figure 3: UNO Pin Layout.

Design – UNO Data Pin Usage

PIN #	Connected Peripheral	Peripheral Usage
GPIO 8	Adafruit NeoPixel Ring – 16 x 5050 RGB LED	Used to send RGB data to change color of NeoPixel LEDs.
GPIO 3	Mini Basic PIR Sensor – BL412	Used to capture the motion by using an interrupt for watching HIGH signals.
UNKNOWN	EEPROM	Used to save car count integer data at memory address 0.
GPIO 0	UART 0 RXD	Used to receive serial data from ESP32C3 UART 1 TXD pin.
GPIO 1	UART 0 TXD	Used to transmit serial data to ESP32C3 UART 1 RXD pin.

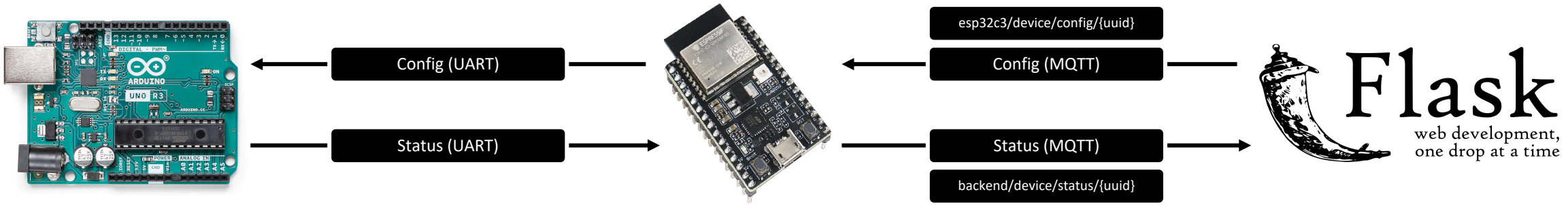
Table 4: *Description of peripherals connected to each pin on the UNO.*

Design – Software Technologies Used

Architecture Component	Technology Name	Technology Usage
Frontend	React	Used for creation of the web application giving access to dynamic components, application states, and routing.
	SocketIO	Used to stream live request and response data from the backend that uses the publisher/subscribe model from MQTT
Backend	Flask	Used as basis for a lightweight backend server.
	Flask-MQTT	Used to transfer MQTT messages from the broker to the backend processing.
	Flask-SocketIO	Used to pass MQTT messages and live stream the data to the frontend of the application.
Raspberry Pi	Mosquitto MQTT Broker	Used to facilitate a publish/subscribe model for messages between the backend and ESP32.
ESP32C3	Custom Firmware	Custom Firmware
UNO	Custom Firmware	Custom Firmware
	Adafruit NeoPixel LED Library	The official library for Arduino devices to control all NeoPixel products.

Table 5: *High-level overview of software used in the Glowpuck Technology Stack.*

Design – Software Data Pipeline



UART/MQTT Communication Protocol

`type\tlabel\tbrightness\tactive\tmode\tr1\tg1\tb1\tr2\tg2\tb2\tgroup_enable\tgroup_target\t car_clear\tcar_count`

Type Values	`{config = 0, status = 1, bootstrap = 2}`
Active Values	`{enable = 1, disable = 0}`
Mode Values	`{still = 0, flash = 1, alternate = 2}`
Group Enable Values	`{group_enable = 1, group_disable = 0}`
Car Clear Values	`{car_clear_enable = 1, car_clear_disable = 0}`

Figure 4: *High-level overview of UART/MQTT Communication Protocol.*

Implementation – Hardware

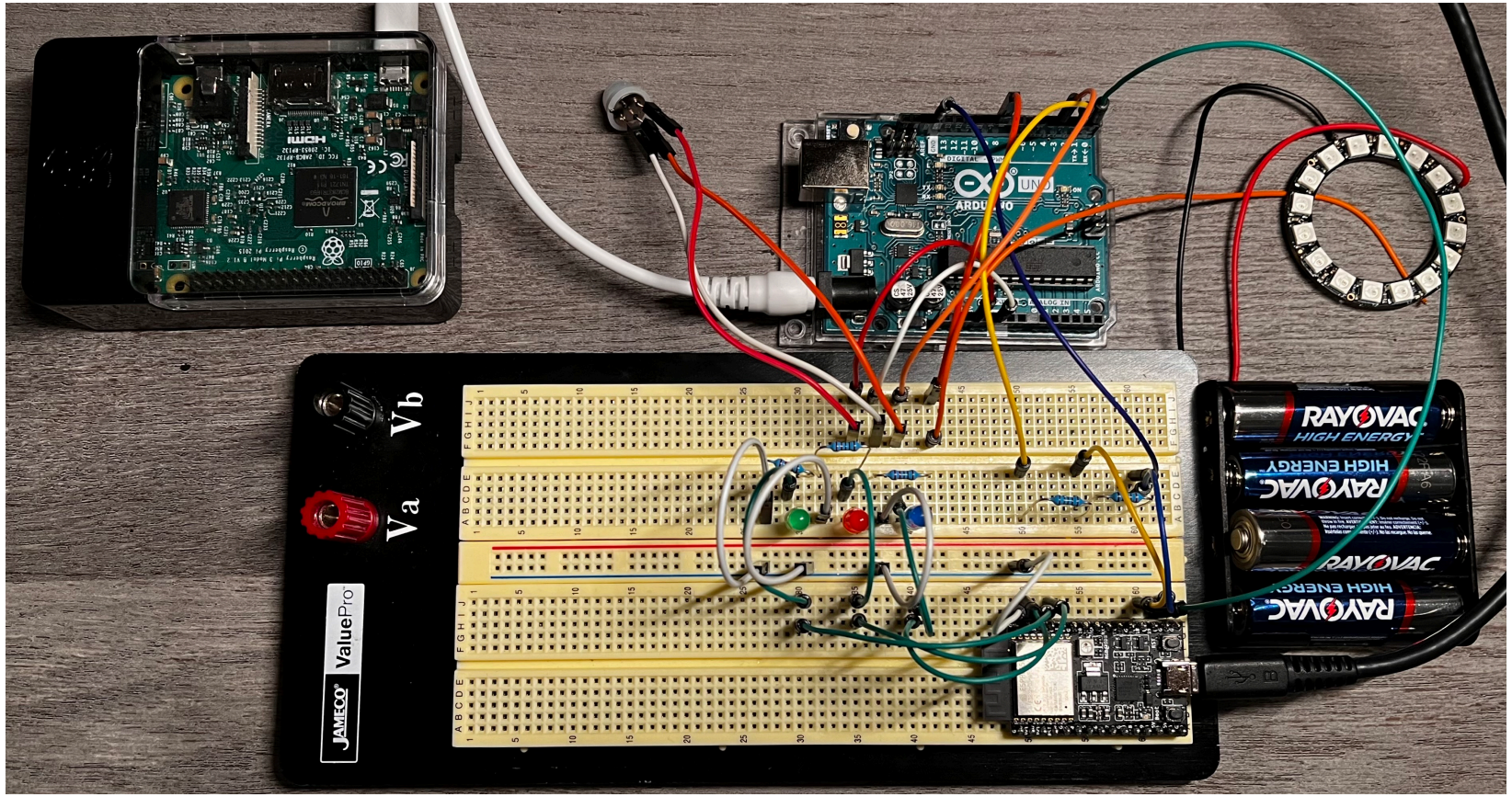


Figure 5: *Glowpuck hardware implementation (off).*

Implementation – Hardware

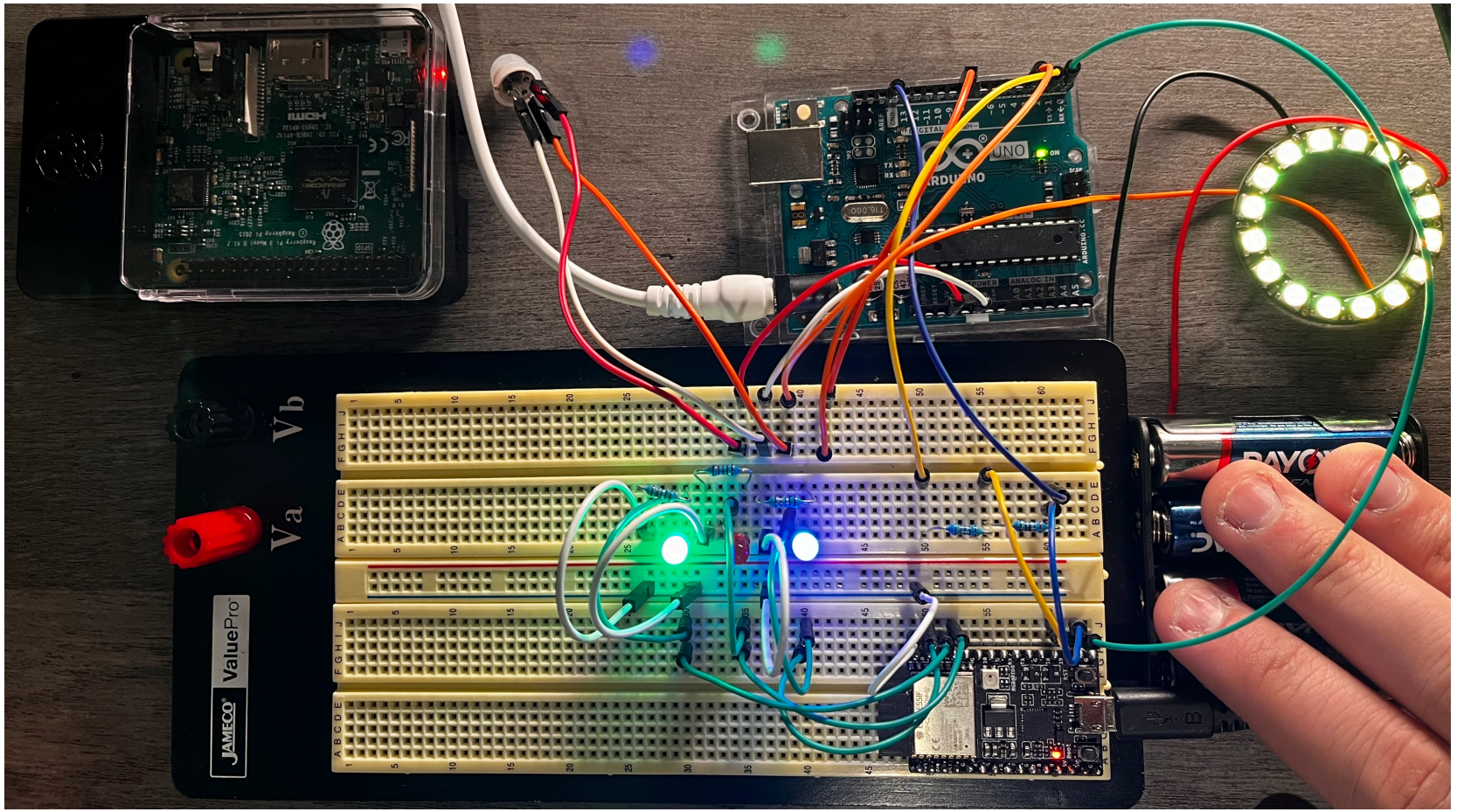


Figure 6: Glowpuck hardware implementation (on).

Implementation – ESP32C3 Functions

```
void parse_protocol_message(char *protocol_message, int *parsed_type,
char **parsed_label, int *parsed_brightness, int *parsed_active, int
*parsed_mode, int *parsed_r1, int *parsed_g1, int *parsed_b1, int
*parsed_r2, int *parsed_g2, int *parsed_b2, int *parsed_group_enable,
char **parsed_group_target, int *parsed_car_clear, int
*parsed_car_count);
```

Function Description (refer to link to view function code)

- Parameters: *The function takes pointer to predefined values to set them at their specific memory location allowing the function return void.*
- *Function: The function uses `strtok()` to parse tokens of the protocol message and at each certain index of the protocol format sets the necessary value. If a value at a certain index is integer, the `atoi()` function is called to transform the parsed string value (being a number) to an integer. Otherwise, if the token is a string as denoted at the index of the protocol format, the pointer for the respective is set to that string location.*

ESP32C3 Code GitHub Link: ESP32C3

Implementation – ESP32C3 Functions

```
void write_esp32c3_status_protocol_message(char
*status_protocol_message)
```

Function Description (refer to link to view function code)

- Parameters: *The function takes a pointer pre-allocated buffer to store the generated status message used for transmission.*
- Function: *The function creates a set of variables that correlate to the protocol format. The function uses the static variables defined at the top of the file (that stores the configuration for each value in the protocol format) to write into the created set of variables. Integer values are directly written into the created variables while memory is allocated for strings via* `malloc()` *to be stored. The function can determine if certain fields in the protocol format are not needed through a set of if-statements. For example, if alternate mode is not enabled then the variables* `message_r2, message_g2, and message_b2` *are filled in with a default value (in this case –1). Once variable setting is completed, the variables are printed into the allocated buffer using the* `sprint()` *function. The type of message (as per the protocol format) is set to* `1`*.*

ESP32C3 Code GitHub Link: ESP32C3

Implementation – ESP32C3 Functions

```
void write_esp32c3_config_protocol_message(char
*config_protocol_message)
```

Function Description (refer to link to view function code)

- Parameters: *The function takes a pointer pre-allocated buffer to store the generated config message used for transmission.*
- Function: *The function creates a set of variables that correlate to the protocol format. The function uses the static variables defined at the top of the file (that stores the configuration for each value in the protocol format) to write into the created set of variables. Integer values are directly written into the created variables while memory is allocated for strings via* `malloc()` *to be stored. The function can determine if certain fields in the protocol format are not needed through a set of if-statements. For example, if alternate mode is not enabled then the variables* `message_r2, message_g2, and message_b2` *are filled in with a default value (in this case –1). Once variable setting is completed, the variables are printed into the allocated buffer using the* `sprint()` *function. The type of message (as per the protocol format) is set to* `0`*.*

ESP32C3 Code GitHub Link: ESP32C3

Implementation – ESP32C3 Functions

```
int execute_protocol_message(char *protocol_message, enum communication communication_mode)
```

Function Description (refer to link to view function code)

- Parameters: *The function takes a pointer of read UART message being a protocol message. The function also takes an enum to denote the communication mode in which the behavior of the instructions executed changes depending upon a UART message (`0`) or MQTT message (1).*
- Function: *The function allocates space for parsed values (as per the protocol format) and then passes their memory addresses to the `parse_protocol_message()` function to parse the protocol message. Once completed, the function goes into an if-statement based upon the selected communication mode and then a switch statement (for each mode) in which the `parsed_message_type` determines the next course of action. For example, if the communication is in `0` (for UART) and the type is of `0` then the function `update_uno_config()` is called to update the UNO configuration (such as light values and light mode).*

ESP32C3 Code GitHub Link: ESP32C3

Implementation – ESP32C3 Functions

```
void update_esp32c3_config(char *new_label, int new_brightness, int
new_active, int new_mode, int new_r1, int new_g1, int new_b1, int
new_r2, int new_g2, int new_b2, int new_group_enable, char
*new_group_target, int new_car_clear)
```

Function Description (refer to link to view function code)

- Parameters: *The parameters are pointers to new values that are to be set to the static configuration variables of the ESP32C3.*
- Function: *The function for each variable of the ESP32C3 takes in the respective variable to the static configuration variable of the ESP32C3 and sets it. For integers, they are simply set while strings use the `strcpy()` to copy variables into the static string variables.*

ESP32C3 Code GitHub Link: ESP32C3

Implementation – ESP32C3 Functions

```
void bootstrap_uno()
```

Function Description (refer to link to view function code)

- Parameters: *(none).*
- Function: *The function allocates a buffer to store the configuration of the UNO config protocol message in which is used to send to the UNO via UART. Using the* `write_esp32c3_config_protocol_message()` *function, the protocol message is generated in which is then used to send (via UART) to the UNO using the* `uart_write_bytes()` *function. This function is used when the ESP32C3 receives a message of type 2 in which the ESP32C3 bootstraps the UNO in the event of the UNO losing power and needs a configuration.*

ESP32C3 Code GitHub Link: ESP32C3

Implementation – UNO Functions

```
void wait_for_uno_bootstrapping()
```

Function Description (refer to link to view function code)

- Parameters: *(none).*
- Function: *The function blocks the UNO until the static* `bootstrap_complete` *is set to* 1. *During blocking, the function checks the UNO UART buffer where it is read as string and then processed by the* `execute_protocol_message()` *function. After that, the UNO is delayed for 2 seconds to process a bootstrap message from the ESP32C3.*

UNO Code GitHub Link: UNO

Demonstration

1. Overview the frontend and see the default configuration being loaded.

2. See the data being live-streamed in near real-time.

3. See changes reflected on the Glowpuck when a configuration update is sent.

Outcomes

1. Created an alpha state software that can be used as a proof of concept for a new tool on roads for traffic control and analytics.

2. Learned the use of a proper protocol and disadvantages of implemented protocol used between the UART and MQTT messages.

Discussion – Future Improvements

1. Consolidate embedded hardware architecture into one device rather than two separate devices.

2. Implement a PIR Motion sensor that has a faster record time lower than the current 2 seconds.

3. Implement secure measures such as MQTT username and password authentication to client and broker.

4. Retool frontend software to be more expendable for custom lighting effects.

5. Refactor the ESP32C3 code to be less prone to memory leaks (in which cause the device to reset).

Conclusion

- Successfully created an embedded device that provides a proof-of-concept for a digital “usher” that can be used to direct drivers while providing minimal analytics using motion sensing to count moving vehicles.

- Was able to successfully design a data pipelines that were both real-time and on-demand to control devices on the MQTT protocol.

- Was able to successfully design a hardware stack that took full use of all UARTs, WiFi, and other protocols.