Free* Tools for Aerial Light Painting

An exploration into how open source projects can be remixed and repurposed for art. Using MAVLink – a popular open source and military-adjacent UAV framework – for light painting.

Daniel Adelodun

September 2025

1 Abstract

This paper walks through my experience contributing to a popular open source UAV ('Unmanned Aerial Vehicle', drone) framework - in particular, it's communication protocol **MAVLink** - to simplify a particular artistic task: Aerial Light Painting. I build upon the underlying framework, creating hardware, software, and developer tools with the intention of sharing these with the wider community, and from which even more artistic uses can be realized and built upon.

I demonstrate the effectiveness of these new additions by producing light paintings myself, noting how my contributions simplify the overall process. Given the size and inertia of the underlying project, it is unlikely my contributions will be significant; nevertheless the following forms an interesting analysis of the framework, where it's size and popularity make it a representative case study on the realities of contributing to an open source project. Fitting with the theme of freedom and openness, I use many other open source tools while developing and testing, in particular KiCAD for PCB design, and blender for flight path planning.

2 Introduction

The importance of Free* (as in freedom) and Open Source tools to the digital ecosystem cannot be overstated. So much so that the phrase "open source" has lost some impact and meaning as it gets overused by marketing companies and commentators to apply to projects that do not fit the original intended class of community-driven, transparently developed, and freely modifiable software. There is a clear and growing societal recognition of the importance of openness, collaboration, and shared technological infrastructure, and as the lines between the digital and physical world continue to blur, it becomes increasingly important to all involved to ask how these projects can be guided towards producing the best outcomes for the communities who in turn maintain and expand the ecosystems they rely upon.

Figure 1: *As of November 2025, Grok is not Open Source*

The politics and history behind many open source projects can get interesting, and MAVLink is no exception. MAVLink (Micro Air Vehicle Link), created by Lorenz Meier, started off as a packet based serial communication protocol used by a few small DIY drone projects. Notably, PX4 (created by the same Lorenz Meier) and ArduPilot, both of which have a long history in DIY drones. Both of these projects still exist today and have become popular choices for hobbyists, and both still use MAVLink for communication. The relevance here is that I will be using PX4 as

the firmware running on the drone, and expanding the ecosystem of community built tools around it and MAVLink to accomplish our goal of streamlining our creative use case.

The politics here go beyond typical online-community drama, and into the world of international relations. Drones are increasing used in war, in particular in the war in Ukraine. The United States has recognized MAVLink as an alternative to the Chinese hardware which dominates the market. As a result, much of the development of MAVLink is with military applications in mind. There aren't many military applications for multi-colour LEDs - the dream would be that as more development is done on these tools and applications, the project and hobby can become less associated with war.

3 Past Work

Drones are increasingly common tools of choice in areas beyond the military, for example in cinematography and surveying. This particular use case is already fairly well supported in MAVLink via a subset of messages related to controlling gimbals and cameras. They were introduced to the main project after first being suggested by a single hardware vendor, who also created camera gimbals which can properly interpret these messages. Over time, features and UI elements were added to the tools used to control the UAVs that would send and receive these messages to interoperate with the gimbal hardware.

OlliW's storm32 gimbals and the subset of MAVLink messages created by him are what formed the model for my own contributions. I will go into more detail later on what MAVLink messages look like, and what the end-user tools are, but I based my contributions around the same model: create messages for controlling LEDs, hardware that supports these messages, and modify the GCS tools to make use those messages.

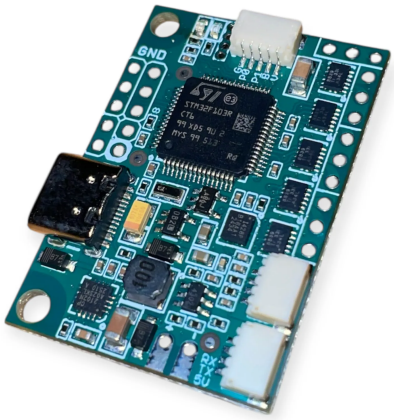


Figure 2: *OlliW's Storm32 Brushless Gimbal Controller board.*

Using UAVs and coloured lights for creative tasks is also a common use case - in drone shows and light paintings. One of the earliest and most impressive examples of this being Marshmallow Laser Feast's 'Meet Your Creator' (2012) which used multiple LED-strapped drones, mirrors, and spot lights in a choreographed show.

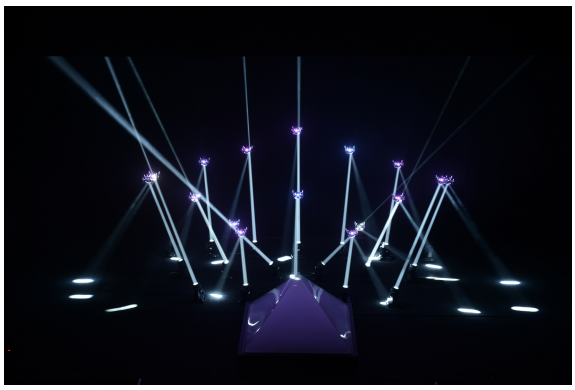


Figure 3: *A still image of Marshmallow Laser Feast's 2012 'Meet Your Creator' performance.*

Most drone shows today use upwards of 100+ drones carrying bright LEDs to create complex and morphing geometric shapes. MAVLink drones are very often used for these shows, but even then a lot of custom hardware and software is needed to make it all work.

Figure 4: *Studio Drift's 2025 'Wind Of Change' performance. 2000 drones where used in this show.*

These shows take a lot of time, effort, money and custom software, largely as a result of the number of drones involved. I would like to make things easier for people who want to accomplish a similar but less expensive goal, by using a single drone (and a camera capable of long exposure photography) for light painting. Instead of many drones working together to produce an image, a single drone traces a path whose trail is picked up while the camera is set to capture the image. Software does exist to help people create drone shows with MAVLink drones, but these are targeted towards those who want to put on big shows with many drones. My target demographic is myself and other people like me (poor students and hobbyists).

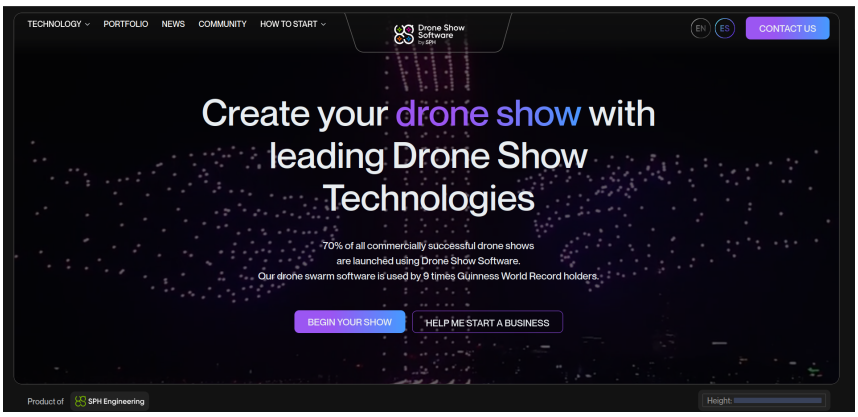
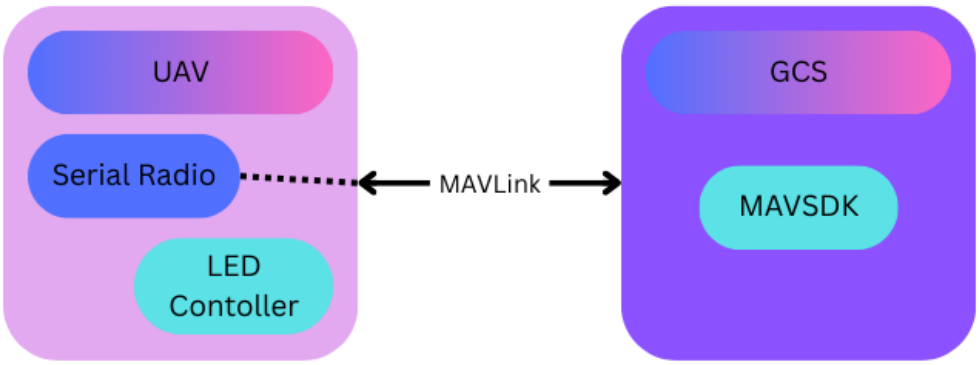


Figure 5: *Drone Show Software: A very popular GCS and planning tool for Drone Light Shows. Compatible with MAVLink.*

Figure 6: *An alternative and open source drone show planning tool. It works with drones running a custom version of ArduPilot/ArduCopter.*

4 Landscape

At a high level, there are 3 components of our system: 1) the UAV and it's hardware, including the LED controller. 2) The Ground Control Station/Software, which should have the ability to control the LEDs and guide the drone to produce interesting light paintings. And 3) MAVLink, to facilitate communication between all the different parts.



4.1 UAV Hardware

First, the hardware for the drone. Building DIY drones is a very satisfying and popular hobby, and so there is a lot of information online about how UAVs and quadcopters work, how to choose an appropriate drone, and the various components needed to build one to fit your style or goals. For our purposes, most of that doesn't matter. The important thing is that the drone we use is one designed to be used with PX4/MAVLink, and as such, has the connectors and interfaces needed to communicate with other MAVLink devices.

To accomplish our goal, connected to UAV will be hardware designed to power and control LEDs midflight. This hardware will need to communicate with the rest of the system and to our GCS to allow for producing mulitcoloured light trails.

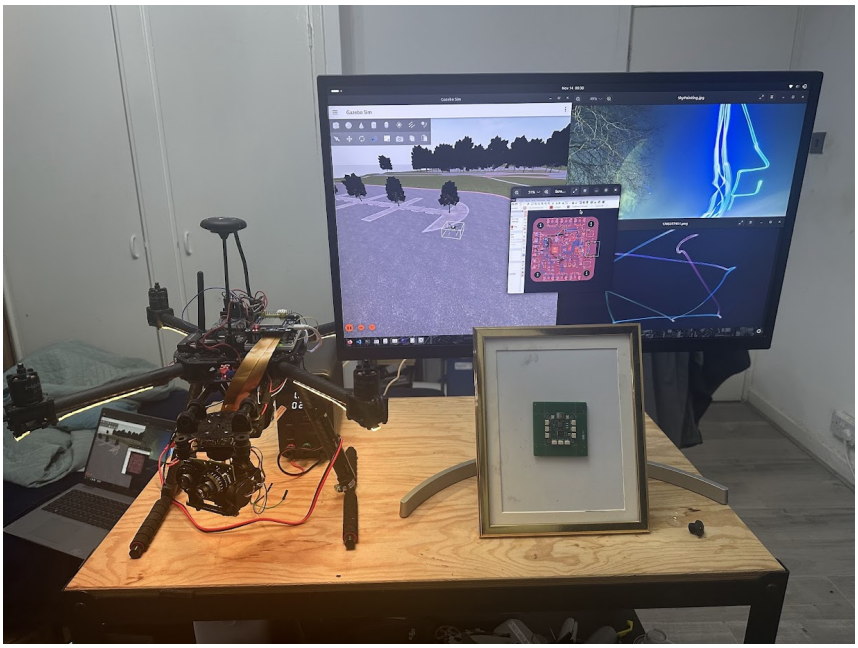


Figure 7: *X500 Drone running PX4 (left). LED Controller (right).*

4.2 Communication

The second component is the communication protocol itself. A drone may use hardware created by one vendor (Holybro, for example) that itself uses sensors and chips manufactured from another (STM, Bosch, etc.), then motors from somewhere else, all controlled using open source software (PX4, Ardupilot, etc.) maintained by industry partners and volunteers. MAVLink facilitates communication between these different components. Whilst relatively simple to understand and expand, it is what determines the capabilities of the overall system and the potential use cases, so we begin here when attempting to expand these capabilities. "If you control language, you control thought" - Orwell, probably.

4.3 GCS Software

Last is the ground control station software - this is how we control the drone when it's in the air. There's a wide range of command and control options for Radio Controlled Robotic Systems. In our case, we are using a serialized and packetized communication protocol, which let's us send arbitrary data (i.e. text data, and not just timed pulses or a set of predetermined signals) to and from our drone. As such, what we can tell the drone to do from the ground is limited only by the set of text messages both our drone and ground station can send and understand. We will need to expand both our ground station and drone to understand and respond appropriately to our new set of messages. In other words, we need to teach our ground station the new MAVLink phrases we have invented.

5 Methodology

Now, lets take a look at the specific tools we will need to use and modify.

5.1 UAV Platform

The drone I'll be using will be a holybro X500 with a Pix35v5 running PX4. Again, the specifics do not really matter here outside of the LED controller, but this is a nice choice as it is one of the "standard" test development platforms for PX4, and so the simulation tools built by that community come with the X500 as an option. This is very useful for testing and development, as real batteries only last a certain amount of time and drones do not like rain or high winds.

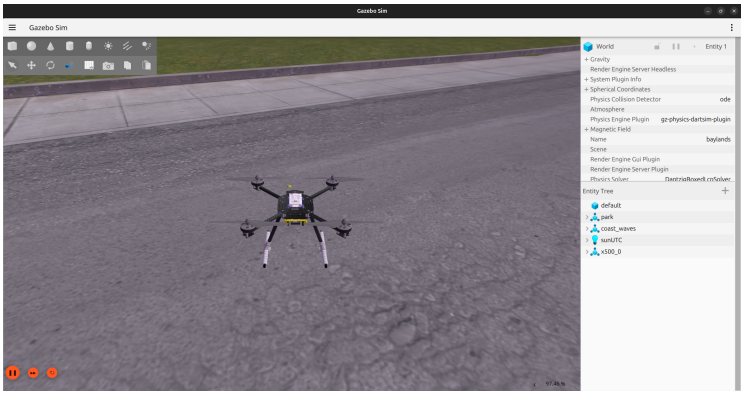


Figure 8: *X500 drone in a simulated environment*

5.2 LED Controller

The LED Controller is the most important part - it is what enables us to change the color and intensity of the LEDs on the drone midflight via MAVLink. It is designed to be mounted using the standard mounting style of DIY consumer drones: 4 M4 mounting holes spaced 30.5mm apart, used to hold the board in place inside the drone frame with appropriately sized nuts and bolts. Connected to the LED controller will be up to 4 WS2811 LED strips, attached using the de-facto standard connector of DIY RC robots, the JST-GH connector. Each one is given an id from 0-3. The individual LEDs on each strip are addressable, so each gets an index number starting from 0. Thus an individual LED can be changed using the strip id and LED index. The board should have a few extra features, including being able to detect and indicate the current flight mode of the drone if asked to, and convert battery power (14V-26V) into the 5V needed for the LEDs.

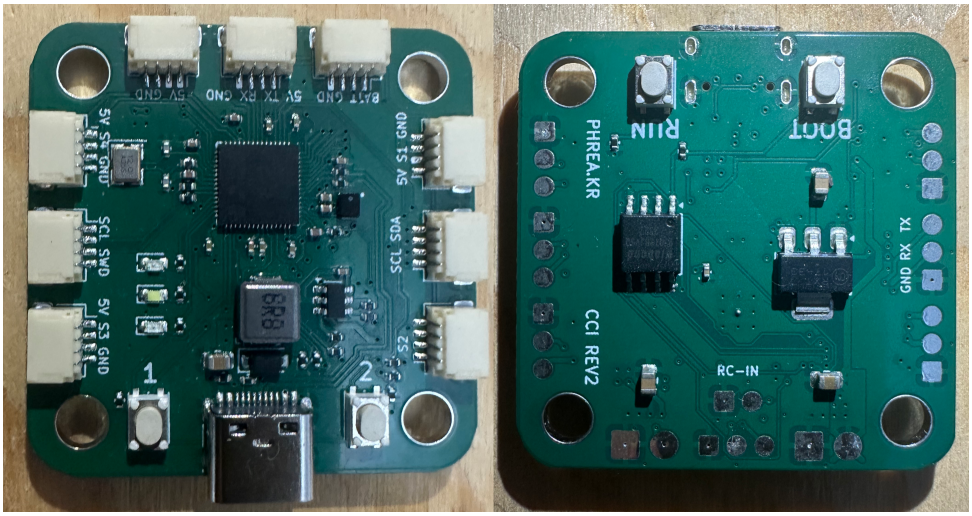


Figure 9: *Front (left) and back (right) of the LED Controller*

The hardware needs to be loaded with firmware to drive WS2811 LED strips, and to interpret the MAVLink messages we created for it.

5.3 Ground Control and MAVSDK

To control our LEDs from the ground, we modify the tools developed by PX4 and the wider MAVLink community so that they can send and receive our messages. The main tools for Ground

Control for MAVLink drones are the GUI programs QGroundControl and MissionPlanner. - both of these projects proved too complex to attempt to use directly, and in the end I was not able to create a GUI application for controlling our LEDs.

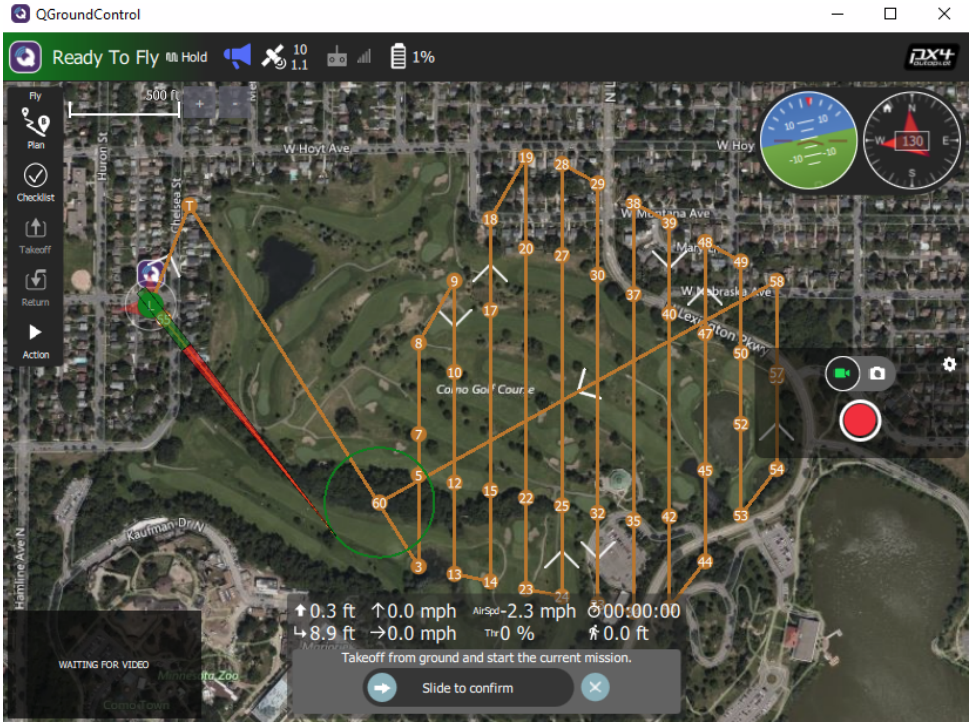


Figure 10: *QGroundControl User Interface. Here we see a flight plan being prepared to be sent to the UAV.*

That said, it was possible (and much easier) to modify the SDK that QGroundControl uses under-the-hood, **MAVSDK**. MAVSDK is a collection of libraries for various programming languages, used to interface with MAVLink systems such as drones, cameras or ground systems. It simplifies tasks like opening a communication channel between two MAVLink systems (for example over WiFi of or a serial port), listening for certain messages or sequences of messages, performing actions like landing and traveling to a certain position, and so on. It uses a plugin structure, and so making new additions and adding new features can be done without completely changing the entire library. In the end, I create a plugin for MAVSDK to control our lights, and then use Python scripts that include the modified SDK with the additional LED plugin for control.

5.4 Flight Planning and Blender

Since it was not possible to directly modify QGroundControl to support LED control, I still needed a way to plan and visualize the flights. To simulate the planned flight once we have created one, I can use ROS2/Gazebo and tools provided by PX4. But to visualize and plan the paths in the first place, I attempted to follow the path set out by one of the drone show planning software options mentioned before: SkyBrush. Skybrush - a tool for planning drone shows - works as a blender add on, allowing users to set and see where each drone will travel as an animation through blender's 3D viewport. You can then export those paths as flight plans for each drone. I did not create a full add-on, but I do use blender scripts to create paths and visualise them, and then I export the the flight paths as a CSV file of points in space over time. That CSV file can then be used by the MAVSDK python scripts to complete the flight.

I did attempt other ideas, including using Unreal Engine along with photogrammetry tools and 3D mapping tools like Cesium to help plan flights in a digital twin version of the target location. This proved too difficult, but I think would be the next step in creating a fully complete tool.

6 MAVLink

Figure 11: *Each MAVLink message becomes a series of bytes when transmitted. The PAYLOAD contains the data needed complete the specified message type, set using the MSG ID.*

I have mentioned MAVLink many times so far - as the communication protocol between all the components, it effectively limits the capabilities of the overall system. MAVLink messages contain fields which determine the message type, the source & target system, the source & target component in that system, and any parameters needed for the message. Each message is serialized into a single MAVLink packet, and that data is sent were it needs to go via whatever communication link is available, usually either WiFi/IP/UDP/TCP or a direct serial RS232-like connection. A good example is the heartbeat message: it has message ID 0, and will broadcast information about the source system to anyone listening.

MAVLink messages are defined using XML files. XML allows for setting for rules for the allowed attributes and file structure, making it possible to validate correctly defined messages automatically. Inside the XML file, message fields are given names and their sizes and data types are defined. Also inside the XML file, names are given to special integer values that are used by the messages - these are called *enums* (enumerators) aka named constantans.

```
<?xml version="1.0"?>
<mavlink>

    <include>common.xml</include>
    <!-- <version>9</version> -->
    <dialect>8</dialect>

    <enums>
        <!-- Enums are defined here -->
    </enums>

    <messages>
        <!-- Messages are defined here -->
    </messages>

</mavlink>
```

For example, **heartbeat** messages are defined as follows:

```
<messages>
  <message id="0" name="HEARTBEAT">
    <description>
      ....
      This microservice is documented at https://mavlink.io/en/services/heartbeat.html
    </description>
    <field type="uint8_t" name="type" enum="MAV_TYPE">
      Vehicle or component type....
    </field>
    <field type="uint8_t" name="autopilot" enum="MAV_AUTOPILOT">
      Autopilot type / class.
      Use MAV_AUTOPILOT_INVALID for components that are not flight controllers.
    </field>
    ....
    <field type="uint32_t" name="custom_mode">
      A bitfield for use for autopilot-specific flags
    </field>
  </message>
```

```
</messages>
```

Shown above is a stripped down version of the heartbeat message definition. We can see that it has fields with the names `type`, `autopilot` and `custom_mode`, using data types `uint8_t`, `uint8_t`, and `uint32_t` respectively. The `autopilot` field takes values from the `MAV_AUTOPILOT` enum. Let's look at how the names of constants in this and other enums are defined:

```
<enums>
....
  <enum name="MAV_AUTOPILOT">
    <description>
      Micro air vehicle / autopilot classes.
      This identifies the individual model.
    </description>
    <entry value="0" name="MAV_AUTOPILOT_GENERIC">
      <description>
        Generic autopilot, full support for everything
      </description>
    </entry>
      ....
    <entry value="20" name="MAV_AUTOPILOT_REFLEX">
      <description>
        Fusion Reflex - https://fusion.engineering
      </description>
    </entry>
  ....
  </enum>
....
</enums>
```

I won't state here all the rules for defining for defining messages (found in MAVLink, 2025), but 2 are especially important: 1) messages have a maximum allowed size, and 2) each message field must be a basic data type, or an array of basic types. This means we need to figure out what we need of the LED messages, how to represent that using the basic data types, and how to fit it all in one message.

We need to set a limit on the number of LEDs each message targets to ensure the message is not too long. Each message will target a specific LED strip on the controller - this will be set using the `strip_id` field. Then we'll further limit the number of the LEDs set to 8. For flexibility, the `length` field can be used to specify a smaller number. Which 8 LEDs on the strip to set by a partucular message is chosen using the `index` field.

We only have a small set of data types to work with. To represent LED colours, I'll use a common format which takes up 4 bytes per LED colour - 1 byte for each of the colour channels RGB, plus one for pure white. These 4 bytes are concatenated to form a single 32bit value, so the hex value `0xWWRRGGBB` will tell you brightness of the white `(0xWW)`, red `(0xRR)`, green, `(0xGG)` and blue `(0xBB)` channels on a scale of `0` to `0xFF` (255). We're setting 8 LEDs per message, so we'll use an array of 8 `uint32_t`.

So, the message we'll use to configure our LEDs looks like this:

```
<message id="52600" name="LED_STRIP_CONFIG">
  <description>
    ....
  </description>
  <field type="uint8_t" name="target_system"></field>
  <field type="uint8_t" name="target_component"></field>
  <field type="uint8_t" name="mode" enum="LED_CONFIG_MODE">
    How to configure LEDs.
  </field>
  <field type="uint8_t" name="index"></field>
  <field type="uint8_t" name="length"></field>
  <field type="uint8_t" name="id"></field>
  <field type="uint32_t[8]" name="colors"></field>
</message>
```

The `mode` field, which uses the `LED_CONFIG_MODE` enum, determines how to interpret the message. This gives us the option to set the LEDs as described above, or use the same message to clear the LEDs on a particular strip, set all the LEDs to a single value, or have the LEDs indicate the current flight mode instead of being controlled manually. To do this, we use the following named constants:

```
<enum name="LED_CONFIG_MODE">
  <description>
    ....
  </description>
  <entry value="0" name="LED_CONFIG_MODE_ALL"></entry>
  <entry value="1" name="LED_CONFIG_MODE_INDEX"></entry>
  <entry value="2" name="LED_CONFIG_MODE_FOLLOW_FLIGHT_MODE"></entry>
  <entry value="3" name="LED_CONFIG_MODE_CLEAR"></entry>
</enum>
```

We now have MAVLink messages we can pass around to different components of our system for controlling LEDs. To use these messages, we need to generate code that takes our required fields as inputs, and outputs a string of bytes in the format shown in Figure 11. How to use the `mavgen` tool to do this is described in detail on the MAVLink website. We will be interested in generating serialization code for 2 languages: `C` for our hardware LED controller, and `Python` for the MAVSDK scripts that we will use for controlling the drone. MAVSDK will also generate `C++` code for itself under-the-hood. Once we've got code to generate the data we need, we need to hardware to pass it around.

7 Hardware

I chose this topic out of an appreciation of open source software. One commonly used example of open source software that has grown to, in some ways, surpass closed source and propriety alternates is the EDA software tool *KiCAD*. Electronic Design Automation (EDA) tools are used to design printed circuit boards and give instructions for how manufacture them and what components to use. Since that is one of our goals - making hardware - I chose to use KiCAD for the task.

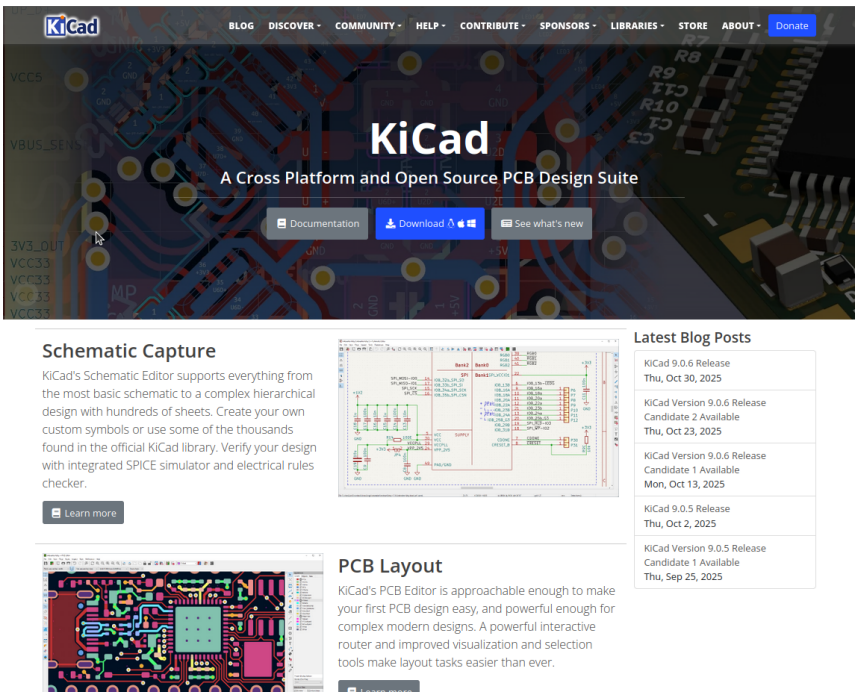


Figure 12: *KiCad Homepage*

This was the most interesting and most time consuming part of this entire process. It was also the most expensive. It was my first time both designing hardware and using KiCAD specifically,

but after following some online video tutorials, using the KiCAD documentation as a guide, and following the hardware design guide provided by Raspberry Pi (and the provided KiCAD project files) I was able produce a working prototype.

This highlights the usefulness of open source software - while a professional might choose to use an non-free EDA tool with more features and support, and may have learned to use the tool either at work or on an "official" course, open source tools benefit from being as accessible as possible, which often makes them good alternatives for hobbyists and beginners. I'd like my LED controller to be similarly accessible, and so it was designed with that constraint in mind.

The first step of this entire process was creating a prototype. To do this, I used the Raspberry Pi Pico 2 development board. This board uses the same MPU chip (the RP2350) that we will use in our LED controller. I go into detail on how I programmed the RP2350 in the next section, but I used the Pico 2 to test the code. This is useful for testing, but there are 4 important features missing from the development board that need to will need to be incorporated into our LED controller.

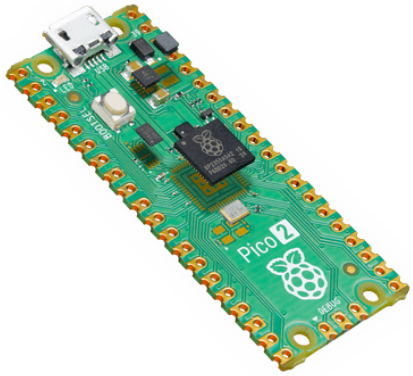


Figure 13: *The Raspberry Pi Pico 2, with the RP2350 MCU. This is what we will base our own hardware on.*

7.1 Connectors

The first is the connectors. A typical WS2811 LED strip has 3 inputs: 5V and GND for power, and a single wire for sending data and changing the colors of the LEDs. This means that our LED controller needs 3 sets of outputs for each strip - 5V, GND and data. I would have then liked to use four 3-pin JST SH connectors on the LED controller, so that an LED strip with the appropriate connector could be easily attached. These are easy to make using a wire crimping set, and since JST connectors are popular choice for hobbyist hardware, some pre-made options exist. Unfortunately, when ordering the components for the LED controller, all 3-pin connectors of the type I wanted were sold out. So I chose to use a 4 pin connector, with 2 pins wired to ground. I also need JST SH connectors for the MAVLink port (using the RP2350s UARTs), for an optional LCD display, for debugging ports, and finally power. For connecting to a host computer for programming, I use a USB-C connector, as I like those better than the micro-usb connector suggested in the hardware guide, and it is more durable.

Figure 14: *JST SH connector socket and plug*

7.2 Power

For the prototype I needed to "borrow" 5V and GND from servo outputs built into the drone, so that LEDs strips were powered using the drone's buck converter. This is a little risky, as it's not guaranteed that the buck converter used for the servos is able to produce enough current to power LEDs. In fact, it might cause the flight controller to lose power during flight if too much power is drawn. I would like to build a high current buck converter into the LED controller, capable of taking battery power (typically between 14 and 26 volts) and turning it into 5V. This was probably the most difficult aspect of the entire project, given that I have no background in power electronics. After much consideration I chose to use the LMR51430 SMPS chip from Texas Instruments.

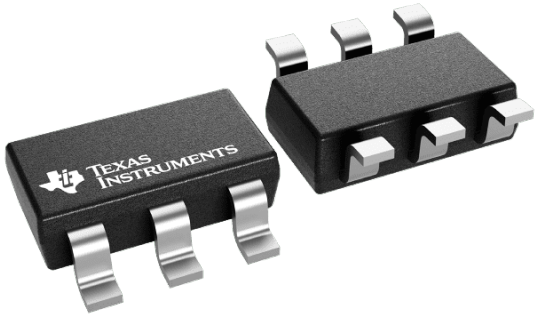


Figure 15: *LMR51430 Buck Converter*

7.3 Mounting

The third issue with the Pico 2 development board is difficultly mounting it. I would like the LED controller to be the same size as many other drone components, and to have the same "standard" mounting options, so I add instructions for the PCB manufacture to drill the appropriate holes. The last thing I'll need to do is add some on-board status LEDs and buttons.

7.4 Status and Control

I would like to be able to tell what the LED controller is doing (looking for the flight controller, responding the LED configuration message, etc.) and enable / disable certain features without always having to send data to the device (e.g. use radio control/servo inputs to control the LEDs instead of our beloved MAVLink messages).

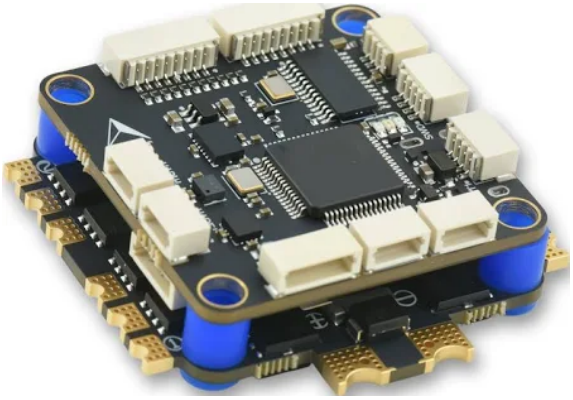


Figure 16: *Example 30x30 Flight Stack. The mounting holes are spaced 30mm apart from each other.*

Thankfully, these are all straightforward demands. We do not need many additional active components beyond what the Pico 2 already has, nor do we need to do much complex electrical engineering work. We can base our design largely off of the hardware design guide supplied by Raspberry Pi for the RP2350, and the schematic provided for the Pico 2.

First, we give out PCB an outline, with the standard shape and size. The board is a rounded square with dimensions 40x40mm. The mounting holes are designed for M4 bolts and screws, and so will have an inner diameter of 4.3mm. They will be spaced 30.5mm apart.

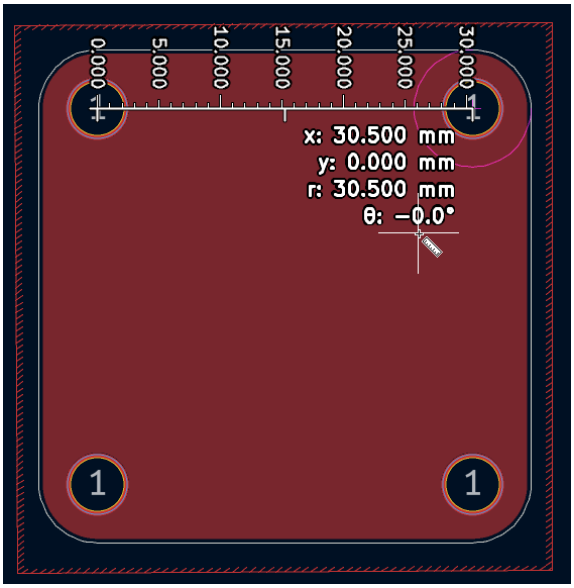


Figure 17: *PCB Outline and Mounting Holes in KiCAD.*

Next are our connectors. We'll use JST GH connectors, the same ones seen in the example flight stack shown in Figure 16. Each of the 4 connector for the LEDs will need to be routed to a GPIO pin on the RP2350 chip, as well as to 5V and Ground for power. We'll also need a few extra connectors for battery power, our MAVLink serial port and some other considerations. The USB-C receptical will also need to be added, with the D+/D- differential pair pins on the receptacle routed to the corresponding same pins on the RP2350.

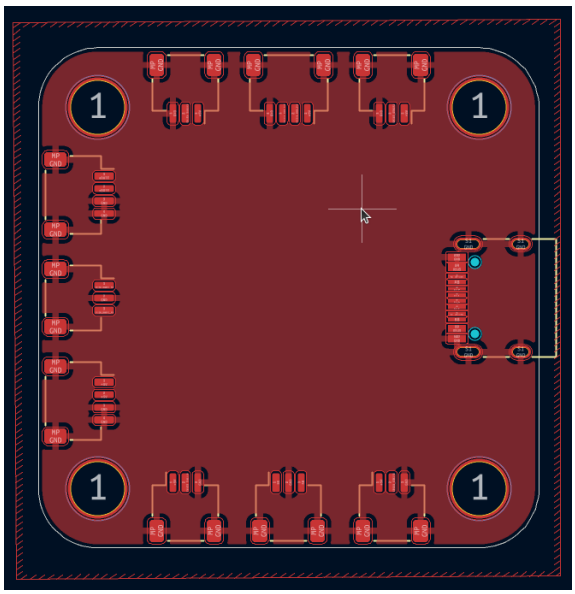


Figure 18: *JST connector placement in KiCad. Note that the 3 pin connectors had to swapped for 4 pin connectors in the end, as the 3 pin connectors where out of stock.*

Opting for a USB-C connector almost proved to be a mistake, as I had initially tied certain 'unused' connectors to ground via the same resistor. After reading through some forum posts, I found out that the RaspberryPi foundation made the same mistake in some early models of the RaspberryPi 4, breaking compatibility with certain chargers. Luckily I fixed this (hence the 'CCI REV2' label), and managed to convince JLCPCB to update the board files prior to production so I wasn't charged twice(!).

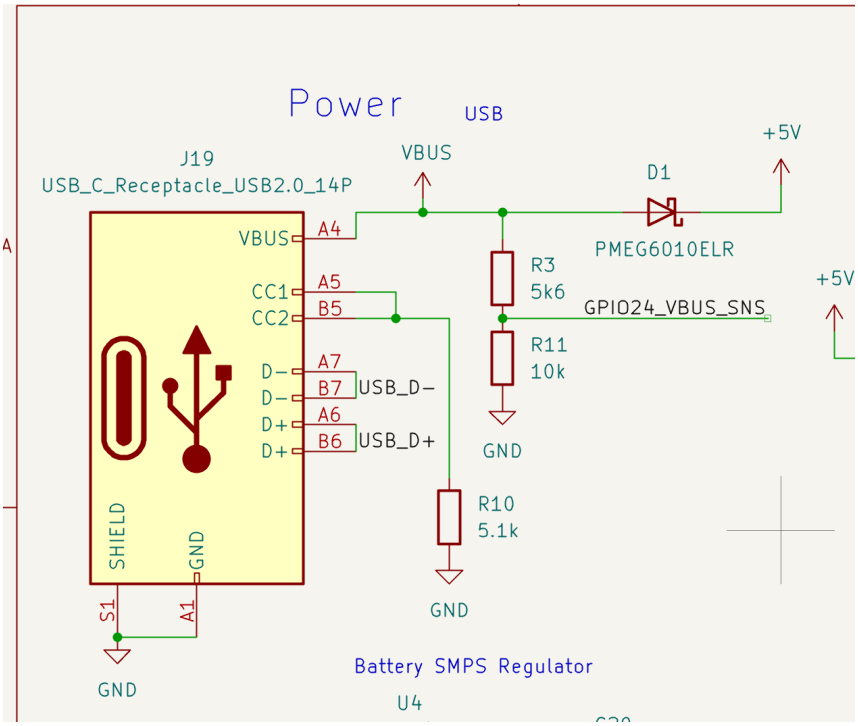


Figure 19: *REV1*

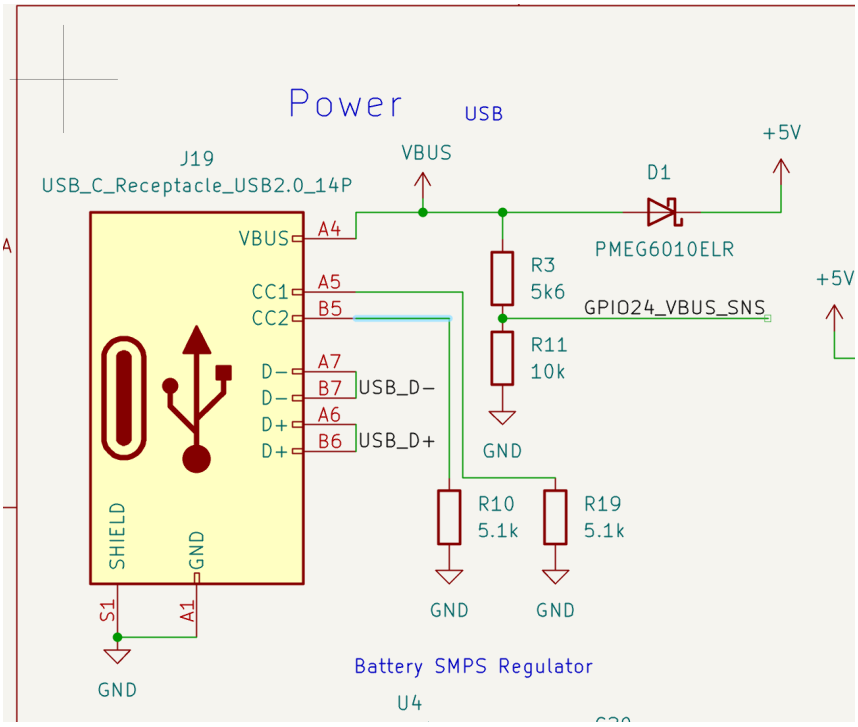


Figure 20: *REV2*

For power, we are using the LMR51430 SMPS to step down our battery voltage to 5V, in addition to the NCP1117 suggested by Raspberry Pi in their minimal hardware guide for stepping down the 5V to the 3.3V needed by the MCU. I use diodes in the appropriate places so that for example power can't flow from the the LMR51430 output back into the the USB port, meaning we have the option of powering the board either from battery, directly via a 5V power supply, or over USB. I chose the LMR51430 as it being a SMPS it has high efficiency and can supply up to 3A continuously. Long LED strips can use a lot of power, and so I wanted to be able to power as many as possible while keeping the 40x40mm PCB size. The downside of using a SMPS over a Linear Power Supply (for example using the 5V version of the NCP1117) is that it is a little more complex to wire. We follow the guide given in the LMR51430 datasheet to figure this out.

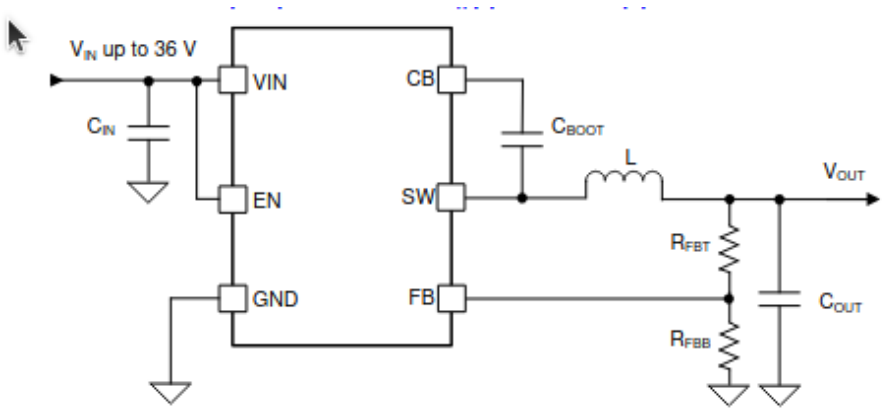


Figure 21: *We need to add some resistors, capacitors and inductors around the LMR51430, with values chosen so that it outputs the required 5V.*

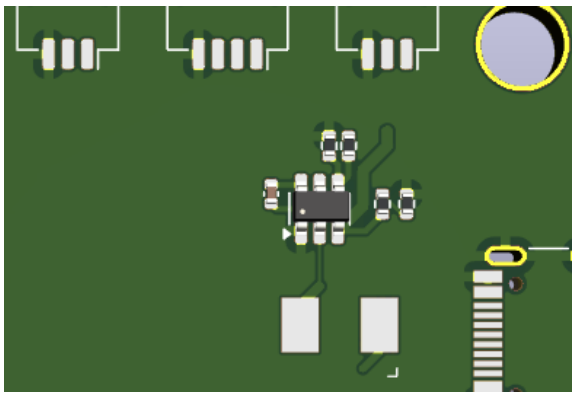


Figure 22: *KiCad 3D view of LMR51430, along with the needed capacitors and inductors*

Last comes the LEDs and buttons. We have 4 buttons: one which puts our board into boot mode by shorting the QSPI_SS pin on the RP2350 to GND; one which resets the board using the RUN pin, and 2 user input buttons. There are 3 LEDs, one which will show the power state of the board, one which will blink when there is some kind of activity, and one which will what 'mode' our LED controller is in (are we setting the LED colours manually, using the flight mode of the drone, or using Servo/RC input instead of MAVLink messages).

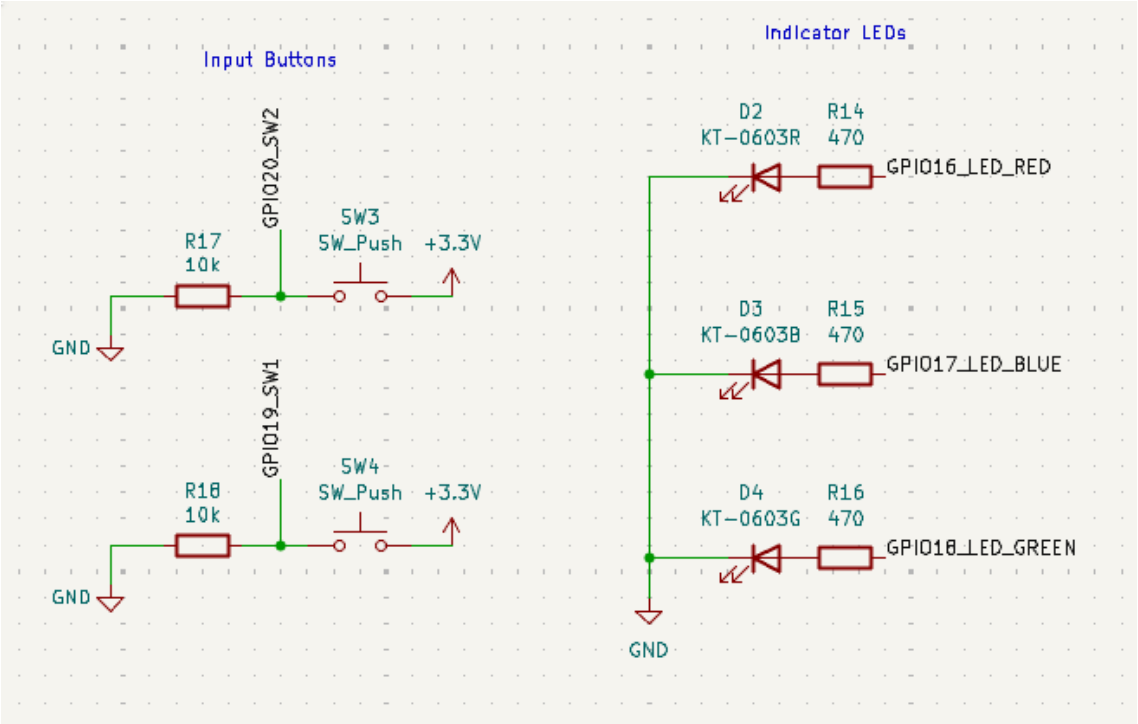


Figure 23: *KiCad Schematic for the Buttons and LEDs*

Beyond this, I just need to follow the hardware design guide to determine where to put bypass capacitors for filtering, inductors for clocks and any other necessary components. The hardware design guide provides an example KiCad project, and we use that as a reference when completing the our board.

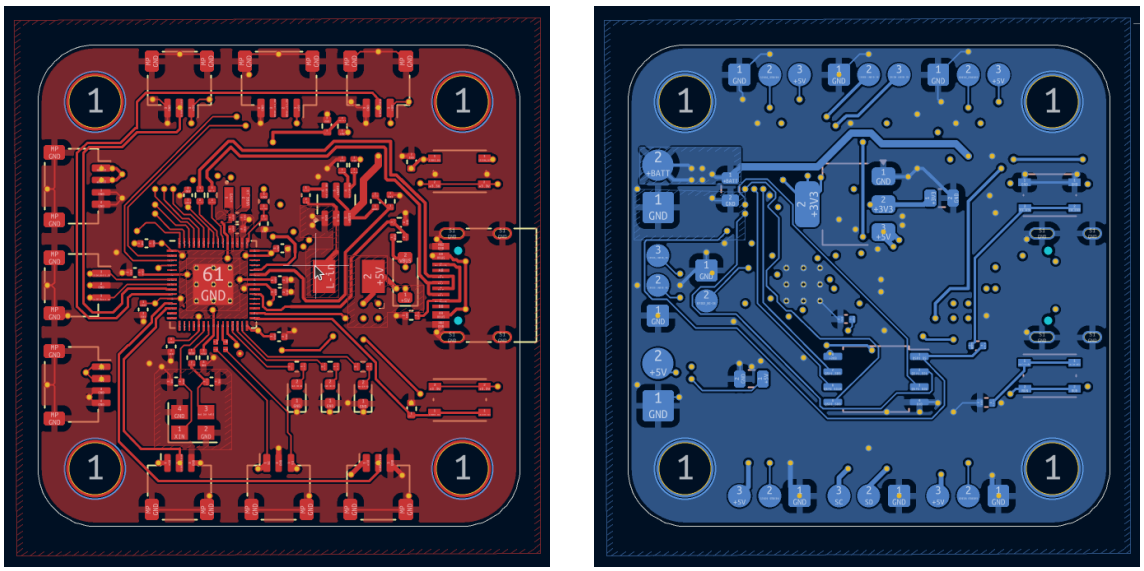


Figure 24: *Front (left) and back (right) of the LED Controller Board*

8 Firmware and the Raspberry Pi RP2350

Programming the RP2350 is similar to programming any ARM based MCU (the MCU will have features/peripherals that can accomplish certain tasks, an to use those features we write and read appropriate values from given memory locations), except that Raspberry Pi provides a higher level SDK that provides APIs which make using each peripheral much easier. We will need to use the UART peripherals for MAVLink and Debugging, I2C for the LCD display, and low-level GPIO control for the buttons and LEDs. How exactly to use the SDK is explained in detail in the books *Getting Started with Raspberry Pi Pico* and *Raspberry Pi Pico-series C/C++ SDK*. More detail on the peripherals and IO memory map in the RP2350 is found in the *RP2350 Datasheet*.

We'll follow the guides to set up our developer environment. I like the SWD debugging feature of ARM MCUs so I set up VS code and my environment appropriately, using a second Raspberry Pi Pico as a debugger. Once this is done, we can begin writing and testing our program.

There are 3 primary tasks that our LED controller needs to accomplish: 1) Reading from the UART and Parsing MAVLink messages, 2) Unpacking message fields and parameters, 3) Controlling the LEDs based on those messages. We make use of the MAVlink messages we defined earlier by generating code that can parse and serialize raw MAVLink messages, including our custom additions. Our firmware is written is C, so we need to generate C code and then use that in our own program. The **mavgen** tool generates header files based on our message definitions, and so to use them in our project we only need to make sure they are #included.

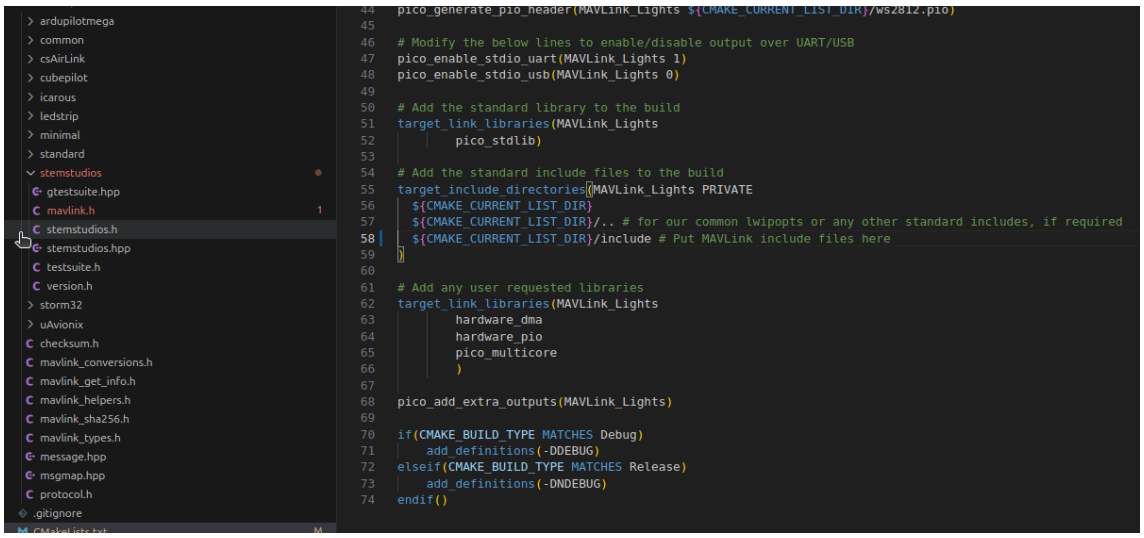


Figure 25: *CMake files for including MAVLink Headers in our RP2350 project*

The RP2350 is a 32-bit dual core chip, meaning we can easily run two separate tasks simultaneously. Because of this, I decided to split our tasks into two groups. The first thread reads characters from our UART, parses valid MAVLink frames into message structs, and enqueues the messages we are interested in (specifically, **heartbeat**, **mode_change**, and our newly created **led_config** messages). The second thread dequeues each message and unpacks the individual parameters. Then, depending on the message type and the current state of the LED controller, either configures the LED strips as requested, or changes the state of the controller to watch for changes in the flight mode and change colours based on that.

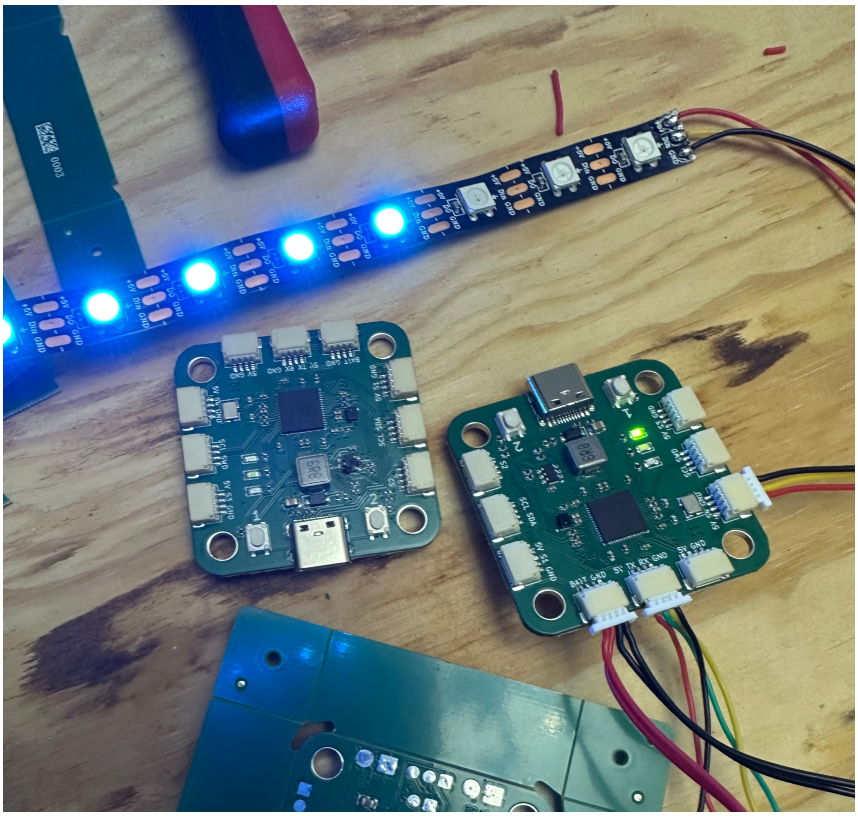


Figure 26: *Testing the Hardware and Firmware on the bench...*

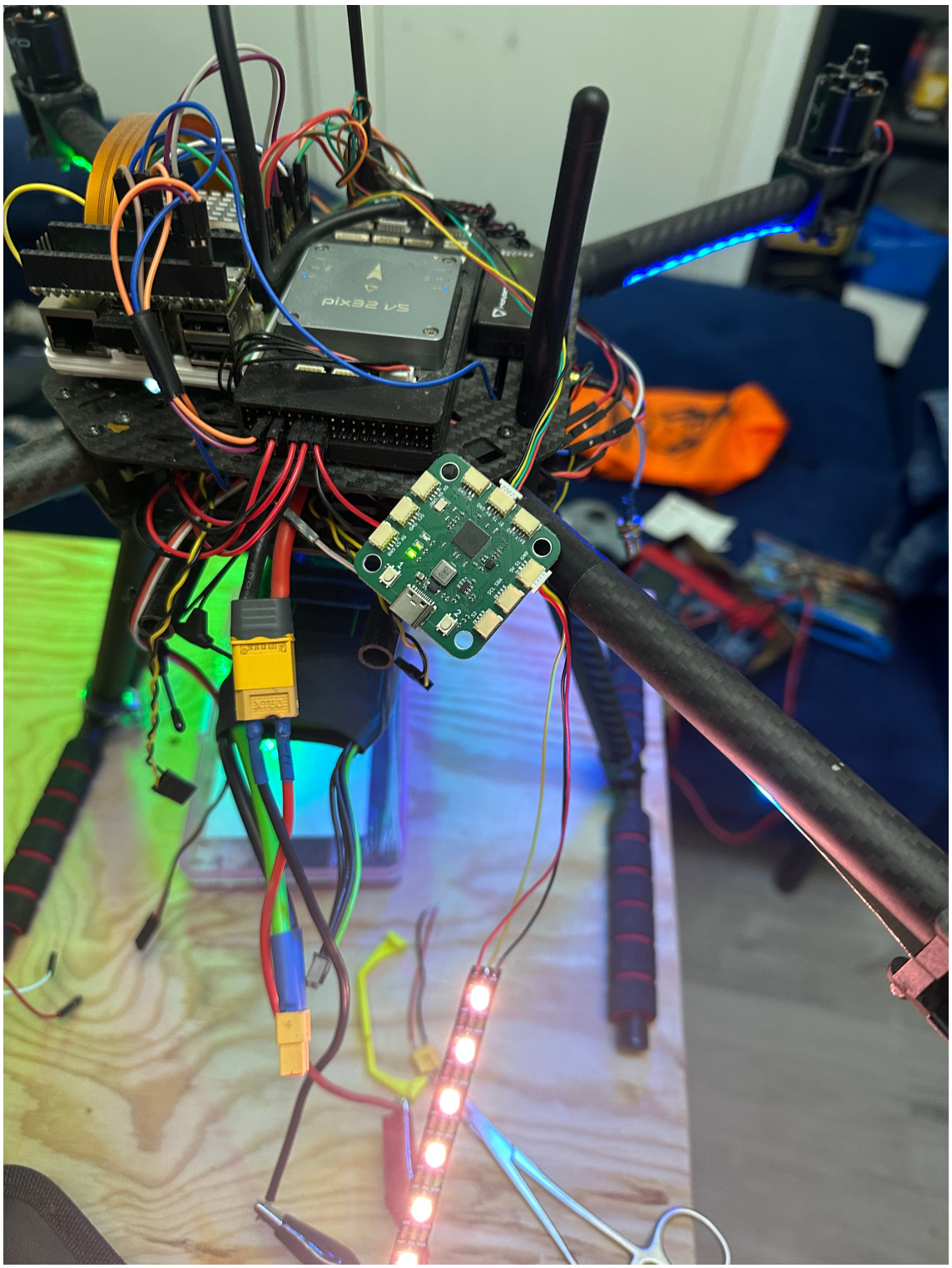


Figure 27: *...and in the drone*

9 MAVSDK

Once we have hardware that can interpret our mavlink messages, we'll need software on the ground to send messages and control that hardware. Ideally, we'd like things to be intuitive enough that the end-user tasks we hope to accomplish - light painting and drone shows - become straightforward. Looking at tools that accomplish this same purpose, it's clear that having a UI that allows for graphical flight planning and pre-visualization would be the ideal. This is what is done in SkyBrush - you are put in inside blender's 3D view port, and can draw paths for multiple drones to follow while setting colors for each drone at chosen locations or times.

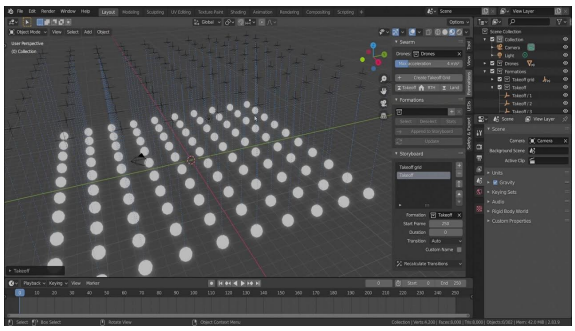


Figure 28: *SkyBrush UI*

I tried this, starting with using a combination of Unreal Engine and 3D scans of real world locations provided by Cesium. I got as far as visualizing paths in an unreal engine, but drawing those paths intuitively and then outputting those paths as flight commands proved too difficult.

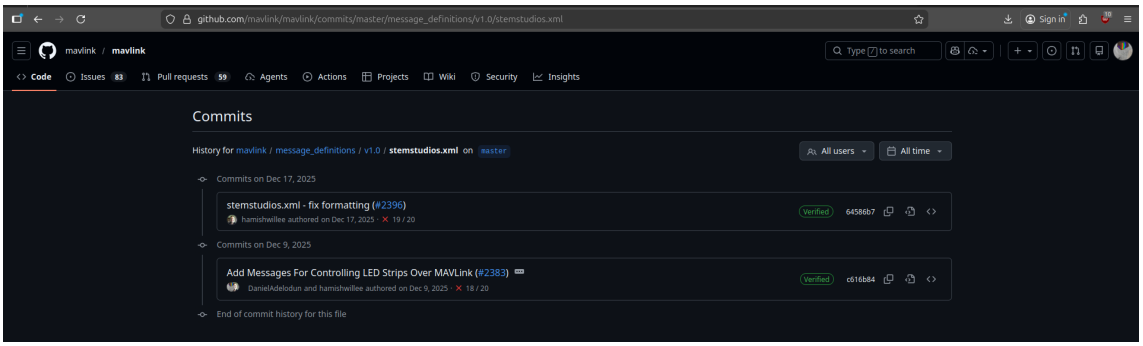
In the end, I settled on only making additions to the Python module developed by PX4 for use with MAVLink systems. Unfortunately, this meant there was no way to visualise the paths or LED colours, outside of simulating the flights after everything was planned out.

MAVSDK-Python works by having different 'plugins' specalised for different tasks, so I had to create a new plugin for LED controls, which I called 'Lights'. With this new modified version of MAVSDK-Python, we can write a script that both sets the target locations and the the desired LED colour at each location. It depends on the C++ SDK MAVSDK, and so the full process for creating a new plugin involves first implementing the logic (specfically, what MAVLink commands and parameters should be sent when calling certain functions) in C++, and then letting the Python module pick up those changes.

While this is not as intuitive as a real graphical UI, MAVSDK-Python is the entry point for a lot of newcomers into programming MAVSDK systems, and so I'd hope this still would make this sort of thing relatively accessible for hobbyists and individuals looking to make use of these commands.

10 Reflections & Aerial Light Painting

At the end of it all, using the hardware created, our newly expanded MAVSDK-Python module, and a little bit of blender to help create paths, I managed to produce some 'Aerial Light Paintings'. While I do not think I fulfilled my original goal of making something truly accessible to anyone, the additions I made to the MAVLink communication protocol were eventually pulled into the upstream branch of MAVLink. Hopefully this means that anyone wanting to expanding on this shouldn't have to start from scratch.



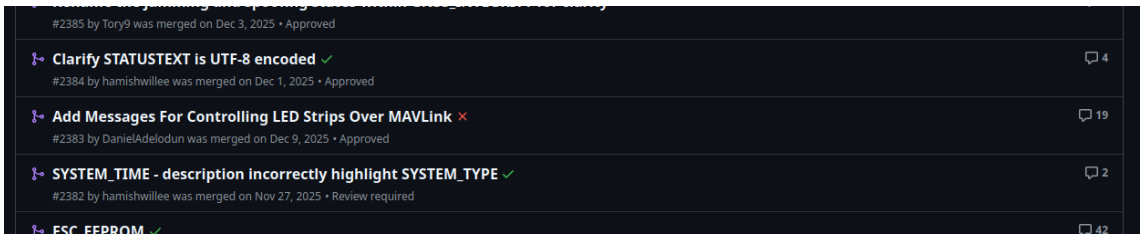
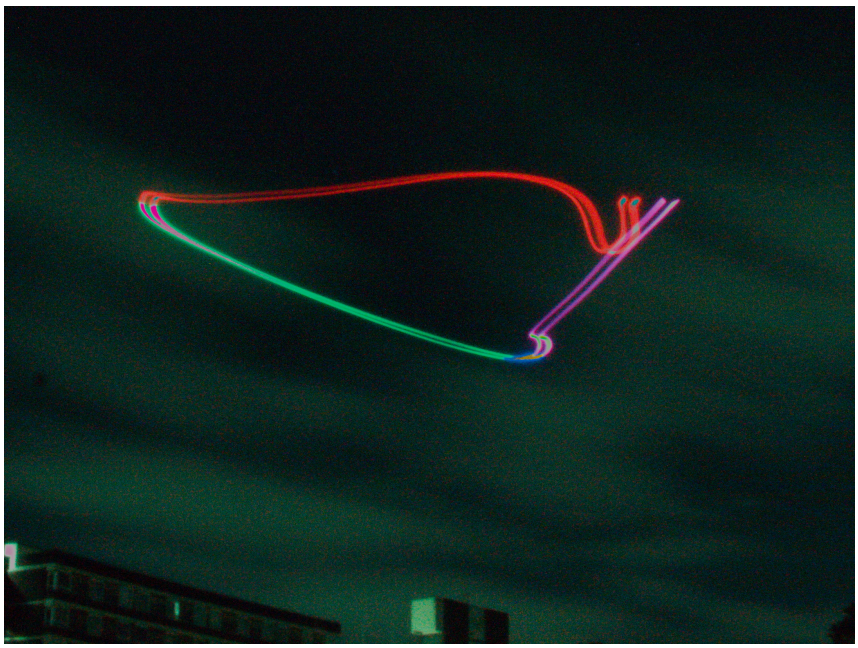


Figure 29: *MAVLink commits in the upstream MAVLink repository*

If I were to make any extra additions or expansions, I would focus on trying to create a proper graphical user interface, and expanding support for other kinds of LEDs besides WS2811 LED strips. I would also attempt some more interesting light painting, in a more appropriate location and with the help of other people.

I used all of these tools to create some light paintings shown below.



References

KiCad Developers Team (2025). *KiCad EDA Homepage*. Accessed: 2025-11-01. URL: https://www.kicad.org.

Marshmallow Laser Feast (2012). *MEET YOUR CREATOR – Quadcotor Show (YouTube Video)*. Video recording of Marshmallow Laser Feast's "Meet Your Creator" show. YouTube. URL: https://www.youtube.com/watch?v=JLAKXJG1trU.

MAVLink (2024). *MAVLink Micro Air Vehicle Communication Protocol Homepage*. Accessed: 2025-11-01. URL: https://mavlink.io.

MAVSDK Developers (2025). *MAVSDK Guide Documentation*. Official guide for MAVSDK, a set of MAVLink SDK libraries for drone communication and control. MAVSDK. URL: https://mavsdk.mavlink.io/main/en/index.html.

OlliW (2025). *STorM32-BGC Wiki – Main Page*. Online documentation for the STorM32 brushless gimbal controller project. STorM32-BGC Wiki. URL: https://www.olliw.eu/storm32bgc-wiki/Main_Page.

Raspberry Pi Ltd (2021). *Getting Started with Raspberry Pi Pico*. Accessed: 2025-11-01. Raspberry Pi Press. URL: https://datasheets.raspberrypi.com/pico/getting-started-with-pico.pdf.

Skybrush Team (2025). *Skybrush Suite: Open-Source Drone Show Software*. Open-source software suite for designing, visualizing, and managing drone light shows. Skybrush. URL: https://skybrush.io.

Studio Drift (2025). *Wind of Change*. Luminous aerial performance with 2000 drones tracing wind patterns and forces shaping the world. Studio Drift. URL: https://studiodrift.com/work/wind-of-change/.