Telemetry, Control Systems, and Sensors on an Autonomous Aircraft

Maxwell Bradley, Kodiak North, Grady Watkins, and Zane Bradley

Abstract—This paper describes the subsystems onboard an autonomous aircraft that detects wildfires. The first subsystem is the telemetry system, which continuously transmits information about the aircraft to a ground station. The second is the control system, which navigates the plane between predefined way points, and keeps it on course in the event of wind or thermals. The third system is the fire detection system, which surveys the area beneath the aircraft, determines if there is a fire present, and communicates through the telemetry system in the event of a fire.

Index Terms—Aircraft, Airplane, RC Airplane, UAV, Autonomous, Wildfire, Wildfire Detection, Search and Rescue, Solar Power

1 Project Introduction

FOREVER Flight is a persistent aerial monitoring system to detect wildfires in fire-prone areas. It will consist of a plane with a mounted IR camera to detect fires below and a flight controller capable of autopilot, guidable with GPS waypoints sent from a laptop computer on the ground. This project was named 'Forever Flight' for its goal of never having to recharge its batteries, staying aloft and performing fire detection while the sun is up, charging back up from solar energy and our power regenerative mechanisms.

This project was inspired by the recent California wildfires like the Camp Fire, a 2018 fire in Northern California that caused $16.5 billion in damage and claimed 86 lives [1]. The reason that these fires were so deadly is because there was no early warning system that detected the fire before it started ripping through towns like Paradise, California. The team hopes that Forever Flight will prevent fires like this.

2 Data Transfer

2.1 Introduction

There is a large amount of information coming out of the aircraft at any time, transferred over telemetry. This transfer contains a huge amount of different kinds of data, such as fire detection stats, fire detection processing, GPS coordinates, and more. Getting this kind of information out of the plane can be challenging because of the flight controller's running overhead and varying transmission protocol versions. I will focus this section of the paper on the transfer of data between the in-flight aircraft and the ground control station.

2.2 System Components

Many different components make up the data transmission path between the plane itself and the ground station. I will now discuss each of those components in detail.

2.2.1 Pixhawk Flight Controller

The flight controller is the main brain of the entire plane. It receives IR information from the downward facing IR sensor through a serial communication port (using the I2C protocol). The Pixhawk is where the majority of the data transfer work occurs in the second and third versions of the data transmission system design. It uses an overall system scheduler to decide when to call the MAVLink packet sending function. Within that MAVLink sending function, one of a large group of packets is selected and sent over whatever telemetry port MAVLink is configured to use.

2.2.2 MAVLink

MAVLink is a protocol commonly used between drones and ground stations. Most autopilots use MAVLink to both send and receive packets to and from the ground station. Its packet structure is described in the following table.

Field Name	Byte Index	Purpose
start of frame	0	Denotes the start of frame transmission
payload length	1	length of the payload, in bytes. Let this value be called 'N'.
packet sequence	2	counter which increments with each message, used to detect packet loss
system ID	3	Identifier for the message sender, so that the receiver can differentiate between multiple senders
component ID	4	Additional identifier for the message sender
Message ID	5	Identifies the message, so that the receiver knows how to parse the payload.
Payload	6 to N+6	The message-dependent data. For example, an attitude message will include pitch, roll, and yaw in its payload
CRC	N+7 to N+8	A 2 byte Cyclic Redundancy Check of the entire packet, to catch transmission errors

The start frame transmission is the hex character set 0xFE, which tells the ground station which MAVLink protocol version is being used. The next byte tells the receiver how many bytes in the payload to expect. The next set of bytes are for identifying the system sending the message, the component sending the message, and the packet sequence.

MAVLink message definitions exist in the common.xml file, where the message title and its payloads are all declared.

The most important part of this packet is the message id, the 5th byte in the index. This byte signals which particular MAVLink message the incoming data corresponds to. Every message coming in indexes to one of the messages in the common.xml file. For instance, several times a second a 'heartbeat' message is sent from the plane to the ground station. This message is declared in the common.xml file as having ID 0. When the ground station receives this message, it reads that xml file, looking for a message definition that matches the incoming ID. When it finds it, it interprets the payload attached to the MAVLink message as the attributes associated with the message definition in the xml file. The autopilot that was chosen for this project uses the MAVLink protocol to communicate with the ground station. During my time as the data transmission engineer, I have become very familiar with this protocol.

2.2.3 ArduPilot

After looking at all the free and open source autopilots on the market, we selected ArduPilot. This firmware can be run on a variety of different platforms: there is an ArduSub, an ArduCopter, an ArduTractor, and obviously an ArduPlane. The flight controller for this project uses the ArduPlane version.

This autopilot is written in C++, and has a relatively small code base for the ArduPlane itself. Its most important parts consist of a main plane scheduler, a MAVLink sending module, and a huge header file that includes all the important modules from the libraries and from the plane folder itself.

The majority of the substance for the ArduPlane module is contained in the libraries. These libraries contain code which all different vehicle versions that run ArduPilot (ArduPlane, ArduSub, ArduTractor, etc.) use. More importantly, they provide functionality for sending MAVLink packets that can be used between vehicles, hardware abstraction layers, control systems, and plenty more.

2.2.4 Telemetry Module

The team landed on using the 3DR Radio Telemetry Kit. This set of transmitters and receivers use the frequency 915MHz. The flight controller uses this link to send flight data. The ground station sends GPS coordinates for the plane to go to on over this link as well. A picture of this module can be found in figure 1.

FPV Radio Telemetry Air Module
FPV Radio Telemetry Ground Module

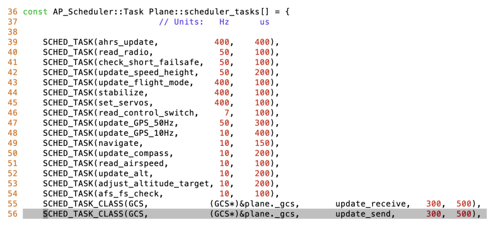
Fig. 1. 100mW 915MHz telemetry modules [5].

2.3 Results and Discussion

This section of the paper will be about the overall set up of the system, the design process, and the problems that were encountered on the way.

2.3.1 *System Set Up*

Our data transmission system starts in the Pixhawk scheduler code within the main file called ArduPlane.cpp. The flight controller is essentially a microcontroller without an operating system, which means that it needs to implement a scheduling system that figures out which process out of hundreds for the processor to run. The scheduler code can be found in figure 2.

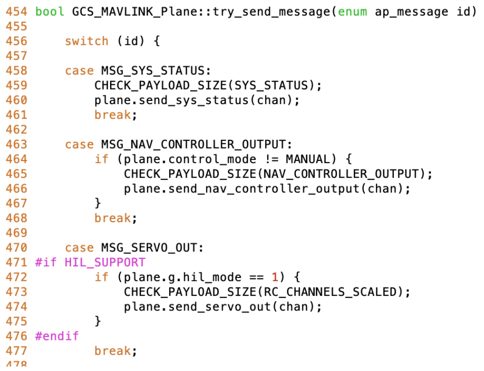


```
const AP_Scheduler::Task Plane::scheduler_tasks[] = {
                           // Units:   Hz      us

    SCHED_TASK(ahrs_update,           400,    400),
    SCHED_TASK(read_radio,             50,    100),
    SCHED_TASK(check_short_failsafe,   50,    100),
    SCHED_TASK(update_speed_height,    50,    200),
    SCHED_TASK(update_flight_mode,    400,    100),
    SCHED_TASK(stabilize,             400,    100),
    SCHED_TASK(set_servos,            400,    100),
    SCHED_TASK(read_control_switch,     7,    100),
    SCHED_TASK(update_GPS_50Hz,        50,    300),
    SCHED_TASK(update_GPS_10Hz,        10,    400),
    SCHED_TASK(navigate,               10,    150),
    SCHED_TASK(update_compass,         10,    200),
    SCHED_TASK(read_airspeed,          10,    100),
    SCHED_TASK(update_alt,             10,    200),
    SCHED_TASK(adjust_altitude_target, 10,    200),
    SCHED_TASK(afs_fs_check,           10,    100),
    SCHED_TASK_CLASS(GCS,            (GCS*)&plane._gcs,       update_receive,   300,  500),
    SCHED_TASK_CLASS(GCS,            (GCS*)&plane._gcs,       update_send,      300,  500),
```

Fig. 2. Ardupilot scheduler

The highlighted line of code in figure 2 schedules the GCS sending module, which calls functions to use the MAVLink link between the ground station and the plane. Diving deeper into the code base in the plane folder, there is the GCS_MAVLink.cpp file that contains the functions to actually go about sending the message. This code can be found in figure 3.

As a backup to the function in figure 2, ArduPilot also implemented a function

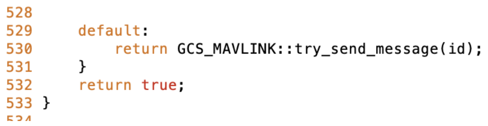


```
bool GCS_MAVLINK_Plane::try_send_message(enum ap_message id)

    switch (id) {

    case MSG_SYS_STATUS:
        CHECK_PAYLOAD_SIZE(SYS_STATUS);
        plane.send_sys_status(chan);
        break;

    case MSG_NAV_CONTROLLER_OUTPUT:
        if (plane.control_mode != MANUAL) {
            CHECK_PAYLOAD_SIZE(NAV_CONTROLLER_OUTPUT);
            plane.send_nav_controller_output(chan);
        }
        break;

    case MSG_SERVO_OUT:
#if HIL_SUPPORT
        if (plane.g.hil_mode == 1) {
            CHECK_PAYLOAD_SIZE(RC_CHANNELS_SCALED);
            plane.send_servo_out(chan);
        }
#endif
        break;
```

Fig. 3. Vehicle specific MAVLink sending function

that all vehicles under the ArduPilot umbrella use. If none of the IDs passed into the try_send_message function matched, it would fall into the try_send_message that was defined in the libraries of ArduPilot. This was good for the overall project structuring, but made figuring out the actual implementation of MAVLink transmission very challenging.

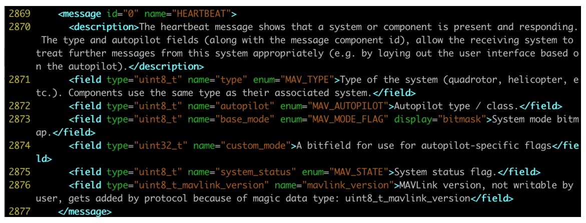
Figure 4 is the fallback function at the end of the try_send_message function that links the plane MAVLink implementation to the common MAVLink packets that all ArduPilot vehicles use. This function is contained in the library files that all ArduPilot vehicles have access to.



```

    default:
        return GCS_MAVLINK::try_send_message(id);
    }
    return true;
}
```

Fig. 4. Sending function common to all Ardupilot vehicles

By editing code in either the library try_send_message function or the ArduPlane specific try_send_message function, I as able to make changes to the MAVLink messages that were being sent to the ground station. I changed existing MAVLink definitions rather than create new ones because of the difficulty in changing the MAVLink protocol version from version 1.0 to version 2.0 (please view the problems section for more information on this issue). I started by selecting messages to overwrite in the common.xml file. An example of the XML code can be found in figure 5.



```
    <message id="0" name="HEARTBEAT">
      <description>The heartbeat message shows that a system or component is present and responding. The type and autopilot fields (along with the message component id), allow the receiving system to treat further messages from this system appropriately (e.g. by laying out the user interface based on the autopilot).</description>
      <field type="uint8_t" name="type" enum="MAV_TYPE">Type of the system (quadrotor, helicopter, etc.). Components use the same type as their associated system.</field>
      <field type="uint8_t" name="autopilot" enum="MAV_AUTOPILOT">Autopilot type / class.</field>
      <field type="uint8_t" name="base_mode" enum="MAV_MODE_FLAG" display="bitmask">System mode bitmap.</field>
      <field type="uint32_t" name="custom_mode">A bitfield for use for autopilot-specific flags</field>
      <field type="uint8_t_mavlink_version" name="mavlink_version">MAVLink version, not writable by user, gets added by protocol because of magic data type: uint8_t_mavlink_version</field>
    </message>
```

Fig. 5. XML definitions of MAVLink packets

2.3.2 Design Process

I began the design process by trying to prevent the flight controller from having to handle any of the data transmission at all. Instead, my original plan was to have a Teensy microcontroller serve as the gateway between the ground station on the ground and the plane up in the sky. The first thing that I did was to set up a UART link between the Teensy and the flight controller, sending over information that would eventually be relayed to the ground station. The overall structure can be found in figure 6.

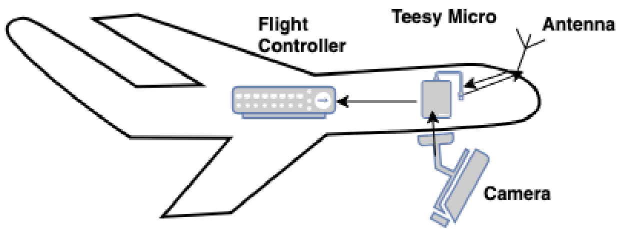


Fig. 6. Diagram of dual microcontroller setup

However, this original set up became less than optimal when the incredible wealth of the already existing ground control software became apparent. I discovered that the ground control software, Mission Planner, could be used to send GPS waypoints to the plane which the plane would then track to without any extra code. The interface between the flight controller and the ground control station would only work if it could receive bytes from the ground station. This required the flight controller to have a direct link to the ground station which it was not configured to do in the first design.

It didn't make sense to spend a lot of time designing a system where the Teensy acted as a go-between for the flight controller and ground station. In addition, the small and underpowered Teesny with a processor clocked at 48 MHz could not hope to keep up with a flight controller clocked at over 3 times that. The Teensy was then removed from the picture, letting the flight controller bear a greater computational load. A diagram of the second design can be found in figure 7.

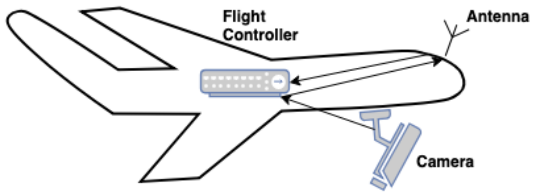


Fig. 7. Updated design without Teensy microcontroller

While this new set up removed complexity from the system, it placed a lot more stress on the flight controller. Instead of just flying the plane and keeping the control systems in check, it had to do image processing work for the IR camera. To reduce the stress on the flight controller, we decided to place the Teensy micro back on board. It would be in charge of performing all fire detection. The flight controller would listen on a serial port for a signal from the Teensy for whether a fire was detected. The flight controller's only added load would then only be to send out one extra MAVLink packet and listen on a serial port. This is far less stressful than trying to squeeze image processing into an already tightly scheduled system. The final design layout can be found in figure 8.

After the two computing devices had their roles clearly set, the design process began anew. The first important thing to do was to get a good understanding all of the underlying code for data transmission in the ArduPilot framework. After tracking down the control flow through the different files which eventually ended in in the library file that all ArduPilot

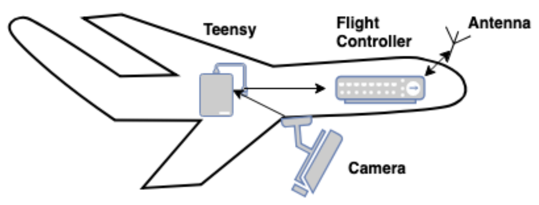


Fig. 8. Final design

vehicles used, the next step was to add a particular message to the common.xml file which contained the MAVLink message definitions.

Due to a parameter problem that I will cover in-depth in the problems part of this discussion, I found it necessary to overwrite one of the existing MAVLink definitions. The HIL_STATE message with ID 90 was selected because it had been replaced by another message and was no longer in use. This message was overwritten with a custom packet that contained fire detection data. This packet would only be sent when the sensing apparatus detected a fire. The simple control flow of this setup can be found in figure 9.

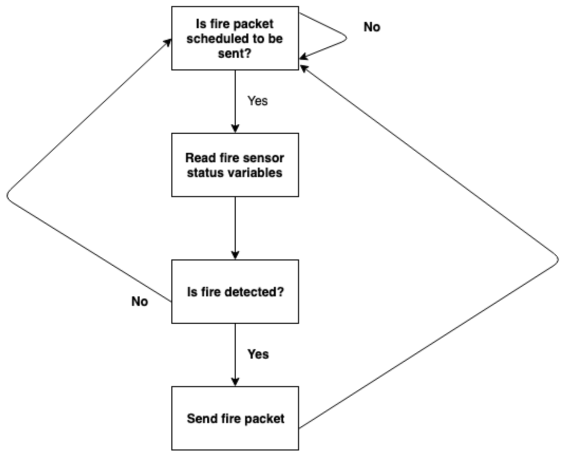

Is fire packet scheduled to be sent?
No
Yes
Read fire sensor status variables
Is fire detected?
No
Yes
Send fire packet

Fig. 9. Control flow of fire packet transmission algorithm

I wrote a Python script on the ground station side that used the Python package PyMavlink. This script read the serial port incoming messages would enter on and translate the incoming data into a human readable form. The associated data was then used to make plots, such as the one in figure 10 that plots latitude and longitude from the incoming GPS data packets.

This allows for the collection of real time flight data which can be very different from the data read from a wind tunnel in the lab.

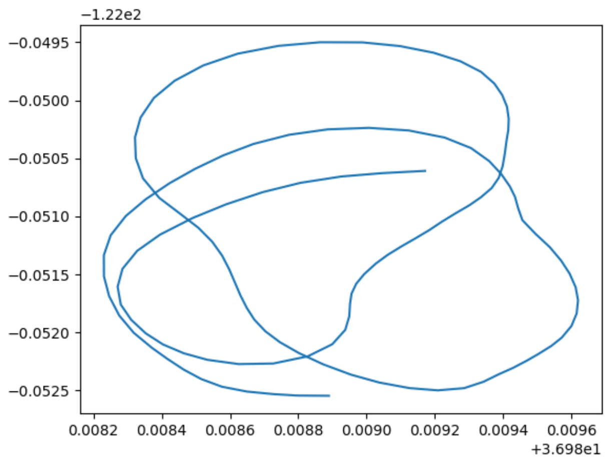


Fig. 10. Plot of the aircraft's maiden flight using GPS coordinates that were logged in real time via the telemetry link.

2.3.3 Design Problems

The biggest and most enveloping design problem I encountered during this project was changing the MAVLink protocol version. Please note the MAVLink byte structure referred to in the next few sentences is the first diagram of data transmission section of the report. The 5th byte of the packet designates the ID number that corresponds to the MAVLink message. This limits the total number of incoming messages to 256. The common.xml file contains message definitions up to 256. Due to the limited number of bits (8) that correspond to the incoming ID, there are only 256 possible messages that can be sent. This is why overwriting the already defined messages in the common.xml file is needed. Creating new messages with an ID value greater than 255 could not be sent by the current MAVLink protocol.

I realized this when I created a custom packet with an ID value of 11065 and tried to send it. However, nothing was sent, not even garbage. This along with the packet header of 0xFE made it very clear that the first version of the MAVLink protocol was used rather than the updated and more flexible MAVLink version 2. Overwriting the message was a necessary evil to send custom packets.

The changing of this protocol is badly documented on the ArduPilot website. Eventually, I attempted a manual change using the MAVProxy Python library (specifically changing the protocol version parameter of MAVLink), but MAVLink 1 packets continued to be sent. In the interest of time, I decided that simply overwriting the useless packets defined for MAVLink 1 was the best course of action.

An added problem of working with this flight controller is the time and processing constraints that adding a custom bit of functionality to an already extremely overtasked system come with. However, by keeping subroutines short, it was possible to add a good amount of data collecting and transmission software to an already built system.

2.4 Data Transfer Conclusion

Data transmission software is essential to pulling real time data that closely reflects and explains the events that occurred during flight. Even though the open source autopilot is incredibly difficult to parse through and overall not very well written by industry computer science standards, it can be manipulated to add custom MAVLink functionality. Once the difficultly of working with different transmission protocols over several different programming languages is overcome, data transfer's extreme value can be truly realized.

3 Control System

3.1 Introduction

This section will assist readers in building their own autonomous aircraft. Starting with choosing an airframe, they will learn why it is important to pick a flying style before buying the first plane at the store. Readers will find joy when learning how simple it is to configure a manually controlled plane, and then they will tackle the task of connecting the autonomous control platform.

3.2 Choice of Aircraft Frame

For our UAV, the team decided to go with a glider style aircraft over a delta-wing style. We originally wanted to use a delta-wing because these planes are quick to build and durable. However, we realized that these characteristics should not be at the top of our list when designing a solar powered aircraft. More important plane specifications include weight, wingspan, and wing area. This criteria directly relates to how easily and slowly the plane will fly; useful for an autonomous fire surveillance system. Note that wingspan and wing area also pertain to the amount of solar cells that can be mounted to the wing. With more area, more solar cells are integrated into the array, which generates more power, and increases our chances of completely sustaining flight. It is clear that a power efficient sailplane is the best choice. Such a plane will utilize updrafts and thermals to gain altitude for free, thus increasing its flight time. The team first built a homemade glider but did not have much luck flying it. The aircraft had unstable pitch oscillations despite the center of gravity being perfectly balanced (see Figure 11). It ended up crashing, but served as a good tool to learn how to connect all of the electronics. Fortunately around the time of the crash, we received funding and were able to purchase a well designed foam airplane kit; the Radian XL. It had a 2.6m wingspan to generate plenty of lift, and it has an optimal amount of area for solar cell placement. The Radian has no pitch oscillations and flies very well.

3.3 Connecting the Electronics

The first step in connecting the control system is to verify that all electronics are working properly before inserting them into the plane. This involves testing the servos and motor in a hobbyists' RC configuration without any autopilot, then connecting them to the autopilot and ensuring manual control can be taken with the flip of a switch, and finally integrating the wireless data transmission system. Refer to Appendix A for complete instructions.

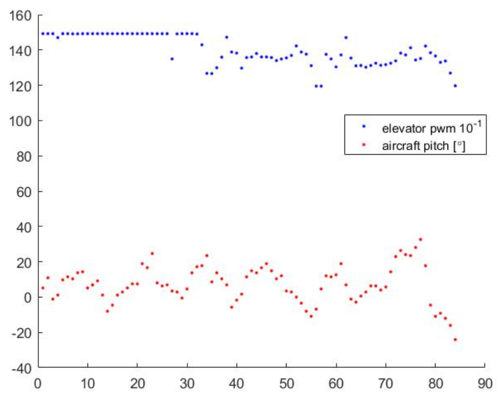


Fig. 11. This plot if of aircraft pitch (red) and elevator deflection (blue) in PWM over the flight duration. A decrease in elevator PWM equates to the control surface deflecting upward, i.e. it causes the aircraft to pitch up. Notice that only pitch up elevator was applied throughout the entire flight, however the plane pitched up and down. This made it clear that the plane was not flying properly. Rather than spending time debugging the issue, we purchased a new plane that was guaranteed to fly well.

3.4 Mounting the Electronics in the Plane

Now with the electronics tried and tested, it is time for the moment we have all been waiting for - to stuff everything inside of the plane. When doing this, it is important to mount the flight controller and GPS as close to the aircraft's center of gravity as possible. It is also important to keep other data transmission antennas (like the GPS and Telemetry link antennas) far away from another to reduce noise generated between the two. Keep the plane's center of gravity in mind when finding a nice place for heavier equipment. It must balance on two points 1/4 of the distance of the chord behind the leading edge of the wing, and preferably be slightly nose heavy. Failure to do this will cause the plane to fly improperly and will likely ensue a crash.

After mounting everything, power up the transmitter, connect the battery to the flight controller and test that all control surfaces move in the correct direction based on manual stick input. Then verify that the plane balances properly over its center of gravity. If it does not, shift something heavy (the battery works well) to correct this. Once all of these checks pass, the plane is ready for manual flight! Remember the Pixhawk can be used to log in-flight data even if it is unused in the first few flights.

3.5 Control System Conclusion

We now have a flight system implemented and ready to be simulated. We do not want to immediately jump into autonomous mode becuase if it does not work as we expect, the plane might crash, or fly far away and never return. For this reason, we are designing a Software in the Loop (SITL) simulation to detect any bugs in the autonomous code. One has been found already where if the aircraft is launched in the opposite direction from its first GPS waypoint, it will never turn around! The ArduPlane code tries to fly the plane around the entire world to hit the waypoint, but since our plane does not have enough battery life to make the journey, it would crash. Once we feel like everything looks good in the simulator, we will launch the plane on its maiden voyage.

4 Fire Detection

4.1 Introduction

The fire detection system onboard the aircraft scans a stream of images for signs of fire. These images are captured by an Infrared camera. A background on the inner workings of this camera will be given below. Infrared imaging is only one of many feasible techniques. An overview of other techniques explored by other researchers will also be given. Finally, this section will detail the fire detection system implementation, the advantages of our implementation, and the drawbacks of our implementation.

4.2 Comparison of Techniques

4.2.1 Human Observation

There are many tried and true methods for detecting wildfires. The oldest, and most simple method is to station human beings in watch towers in high elevation positions, such as the ridge lines of mountains. This method is not prone to false positives, as human beings are very good at identifying a fire once they see it. Human beings can detect fire through many senses, such as visible smoke, visible flames, burning smells, and changes in temperature or humidity. However, human vision has a limited range. A large number of watch towers and human watchers are necessary to monitor a small area. Also, the logistics of watch towers are complex. New watch towers take time to construct, the system is not mobile. The system also places human beings in danger. A fire could easily spread after detection but before being extinguished, and burn down the watch tower.

4.2.2 Manned Aerial Observation

A more recent technique uses human observers inside large manned aircraft. Unlike the watch tower method, this system keeps human beings out of harms way. This system also has the advantage of being mobile, as a plane can be deployed anywhere. The range of this method is also less limited, as a plane cover an area much larger than a human's vision in a single flight. Large enough planes can even be equipped to combat fires when they are detected.

The downsides of manned aircraft are cost and complexity. Flying a plane is an expensive activity. It also takes a highly trained pilot, and a team of people to maintain the vehicle. The fire department of a small rural town will lack the resources to use this method. Instead, they must rely on the state to provide the aircraft, pilot, and maintenance crew. Unfortunately, small rural towns are most affected by wildfires.

4.2.3 Satellite Imagery

Satellite imagery is another technique employed to detect wildfires. Satellites have an enormous upfront cost, but thankfully many space agencies allow small organizations to access their imagery. Many spaces agencies also allow organizations to place custom modules onboard their satellites, in the form of "Cube Satellites". However, the major drawback of using satellites to detect fires lies in the technical details. Satellite imagery has a very slow frame rate, on the order of days [13]. This delay is simply not suitable. In the span of a day, a fire can grow to an enormous size. With the right combination of weather and fuel, a wildfire can grow at a rate of 300 feet per second [18].

Satellite imagery has another crucial downside: low resolution. Satellites orbit at very high altitudes, on the order of hundreds and thousands of miles above earth's surface [13]. Advanced camera equipment would not be able to detect small fires. Satellite imagery would likely only capture large scale artifacts of fire, like smoke trails. However, these only develop after a fire has grown to a considerable size. If the goal of fire detection is to catch fires early, then satellites are not the right tool for the job.

4.2.4 Optical Camera Imagery

Many real world fire detection systems employ optical cameras. These operate in much the same way as a human observer, and are positioned in high elevation positions. They look horizontally over a region, and the resulting images can be digitally processed, or manually monitored for signs of smoke and fire. A major downside of this approach is the horizontal camera orientation. In densely forested areas, a fire may be obscured by trees while it is small. Optical imagery also cannot be used at night, as smoke is nearly impossible to distinguish in the dark.

The figure below demonstrates an optical image processing algorithm produced by Cruz et al. [12]. For each pixel of an input image, the algorithm computes the probability that the pixel is part of a fire.

4.3 Infrared Sensor Background

An infrared sensor is a specific type of spectrometer, which is a device capable of converting electromagnetic radiation into an electrical signal. Spectrometers can be tuned to be sensitive to different wavelengths of radiation. An infrared sensor is tuned to produce the maximum response signal when it receives radiation in the Infrared Region.

According to Planck's Law, all matter radiates energy in proportion to its temperature. The hotter an object is, the more energy it radiates. Plank's Law also tells us that the wavelength at which the most energy is produced decreases with temperature. In other words, hotter objects radiate higher frequency

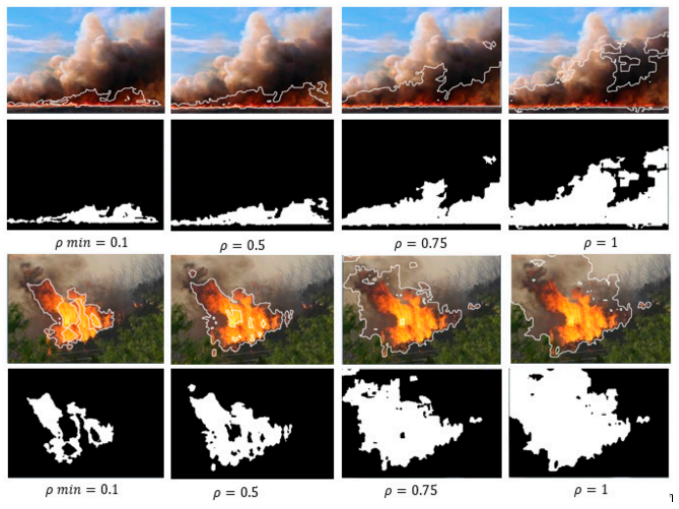


Fig. 12. The Forest Fire Detection Index algorithm with ρ values of 0.1, 0.5, 0.75, 1, and 1.25. Note that the region considered as being ”on fire” grows in size with larger values of ρ. For values above 0.75, the algorithm mistakenly classifies smoke as part of the fire.

energy, and more total energy than colder objects. The figure below shows various Planck curves, which relate energy, temperature, and wavelength.

There are several wavelength bands within the Infrared Region. The Thermal Infrared Region occupies wavelengths of 8-15 μm. The Medium Wave Infrared Region occupies wavelengths of 3-5μm. A peak in the Thermal Infrared Region range indicates an object that is roughly 248K. The Thermal Infrared Region is not appropriate for our application, because wood burns at 573-973K [16]. Instead, we will use the Medium Wave Infrared Region. A peak in this region indicates an object that is between 600K and 900K.

4.3.1 *Infrared Imagery*

Infrared imagery uses a grid of infrared sensors to create an image. Each sensor corresponds to a single pixel, and varies its electrical resistance in response to infrared radiation that collides with the sensor's surface. Infrared imagery has a key advantage over optical imagery, because IR radiation can be detected in night or day, and can detect objects that are merely hot, and not actually on fire. IR radiation is also not obstructed by smoke or fog, which could

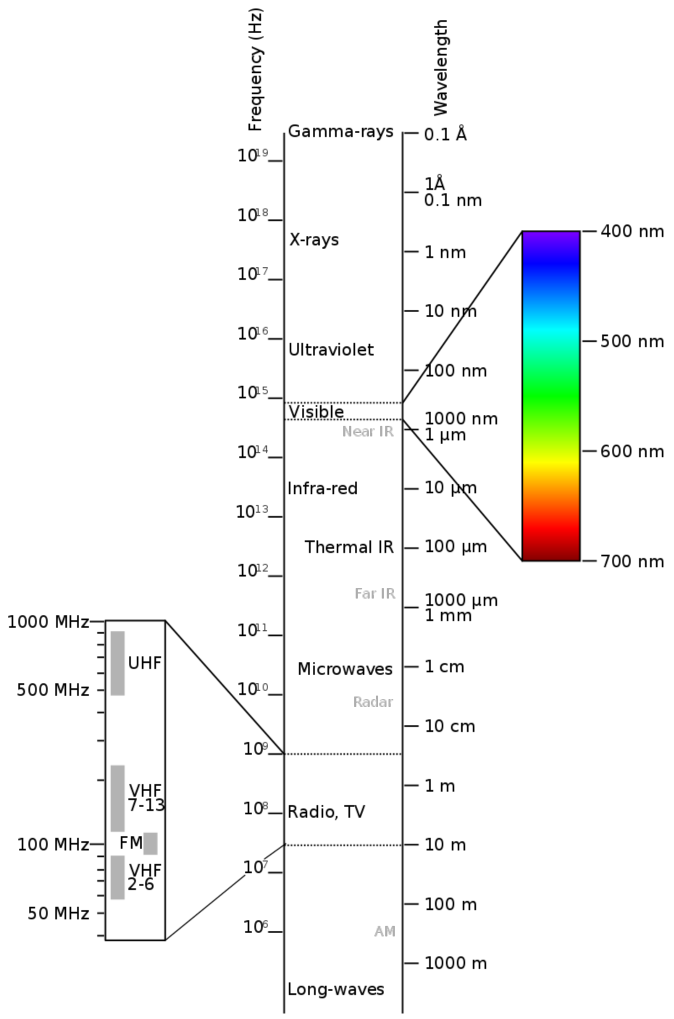


Fig. 13. The electromagnetic spectrum. Radiation between wavelengths of 1μm and 1000μm is classified as Infrared Radiation

obscure the sight line of an optical imagery fire detection system.

4.4 Fire Detection System Implementation

Our fire detection system makes use of an MLX90640, which is a low cost infrared sensor array. The MLX90640 produces images with a resolution of 32 horizontal pixels and 24 vertical pixels. The lens on the camera gives it a 55 degree horizontal field of view, and a 35 degree vertical field of view. The figure below shows the pyramid that encloses the sensor's field of view.

Each pixel represents a 13x10 square foot region, which is the smallest detectable fire size. Any fire smaller than this will not register on

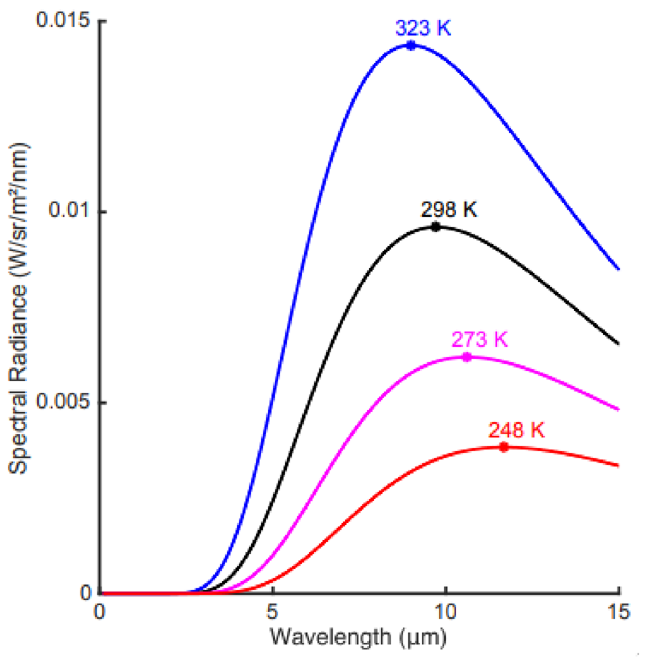


Fig. 14. Planck curves for temperatures 323K, 298K, 273K, and 248K. Notice that as temperature decreases, the peak wavelength increases and the total area under the curve decreases. This means that the temperature of an object can be detected, if we know the peak wavelength of its electromagnetic radiation.

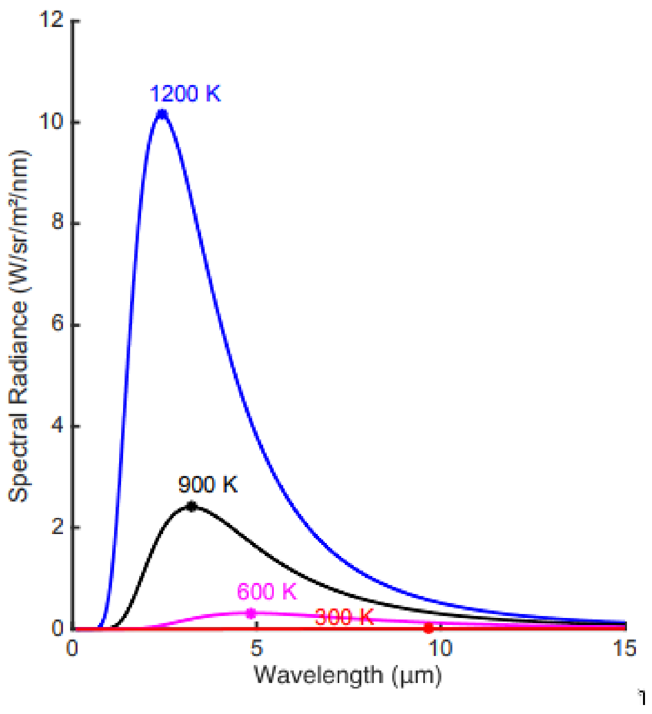


Fig. 15. Planck curves for temperatures 1200K, 900K, 600K, and 300K

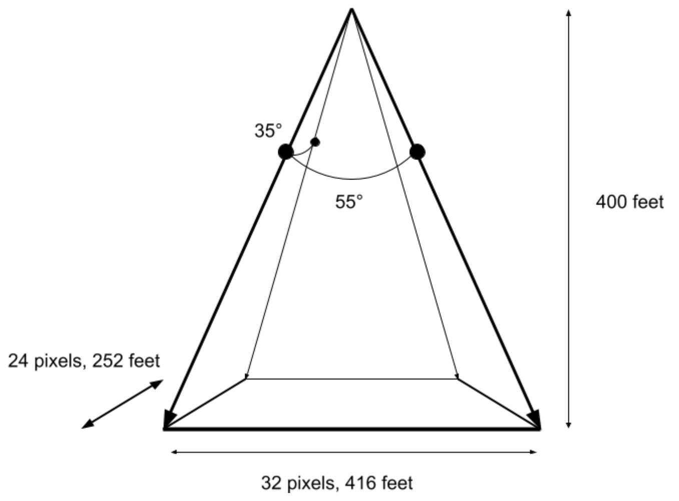


Fig. 16. From a height of 400 feet, images captured by the MLX90640 have a physical width of 416 feet and height of 252 feet. Each pixel represents a 13x10 square foot region.

the camera, and therefore cannot be detected by the system.

Image 17 describes the algorithm that backs the fire detection system.

Upon receiving a fire size message, the main microcontroller transmits the current GPS location, current time, and fire size value to the ground station.

4.5 Limitations and Problems

The minimum detectable fire size is a crucial metric of our system. The smaller it is, the smaller the fire will be when first responders arrive. The smaller it is, the easier it is to suppress the fire. This metric depends on the cruising altitude of the aircraft, and the resolution of the camera. This system was only tested at heights below 400 feet, due to FAA regulations. In practice, the cruising altitude should be higher, to mitigate collision with trees or terrain. This negatively affects the minimum detectable fire size. This can be compensated for with a higher resolution camera, however thermal cameras rapidly increase in price with increasing resolution. In short, using higher resolution thermal cameras is not a scalable solution. Using regular optical cameras is one solution, but these have many drawbacks, as discussed in section 4.2. Our fire detection system produces relatively simple information. It records its current GPS location and the current time whenever it flies over a fire. However, there are many more characteristics about a fire that are relevant to

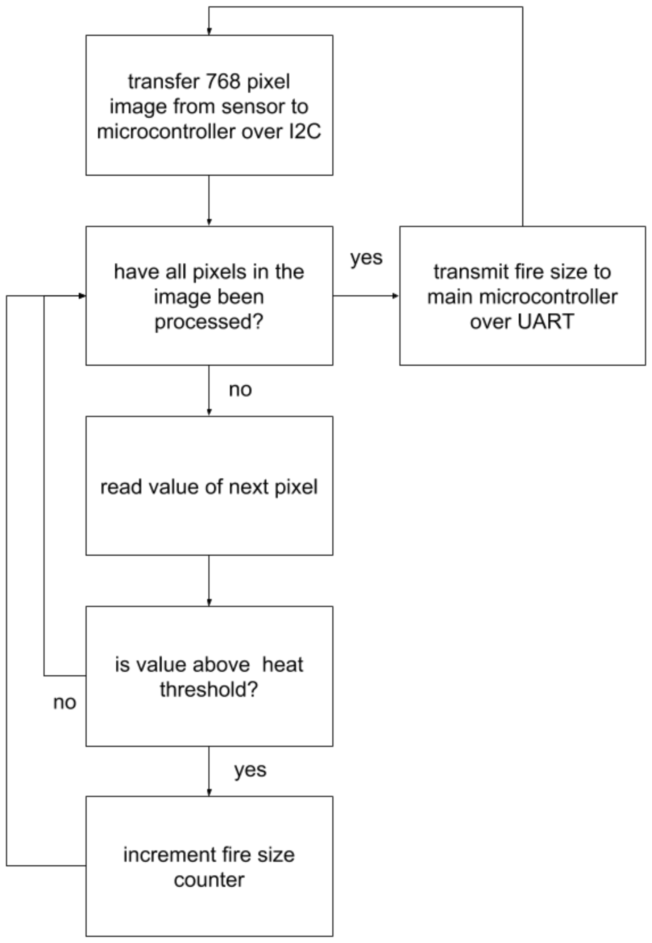

transfer 768 pixel image from sensor to microcontroller over I2C
have all pixels in the image been processed?
yes
transmit fire size to main microcontroller over UART
no
read value of next pixel
is value above heat threshold?
no
yes
increment fire size counter

Fig. 17. Fire detection algorithm

fire fighters.

Fire fighters are interested in modeling fires. The location of the starting point of a fire is nice to know, but it is equally important to know what the fire will do after it has started. Fire models attempt to describe the intensity, forward rate of spread, perimeter growth rate, area of perimeter, and shape of perimeter.

Intensity is defined as energy released per unit time, per unit length of the fire's perimeter. For example, a fire burning in a forest will typically have a higher intensity than a fire in a grassland. This is because the available fuel is much denser in a forest than in grasslands.

The forward rate of spread of a fire is the speed at which a particular point on the perimeter of the fire advances. This metric has units of distance per unit time. For example, the recent Paradise fire reached a forward rate of spread of 100 yards per second. The forward rate of spread is large when a fire moves uphill, and is positively affected by wind speeds.

The rate of spread, or rate of perimeter growth is self descriptive. It also has units of distance per unit time. An extremely large fire could have a small forward rate of spread, but a large rate of perimeter growth.

These models depend on information about the fire, such as location, the moisture content of the fuel, the wind speed, the wind direction, and the slope of the surrounding terrain. Figure 18, produced by F. Albini [16], shows different shapes of fire, depending on wind speed.

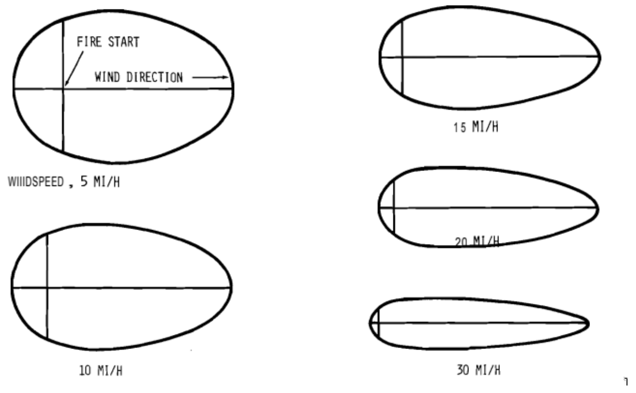


Fig. 18. Approximate fire shapes (not sizes, the scales are arbitrary) for windspeeds of 5, 10, 15, 20, and 30 mi/h [16]

It is important to keep in mind that models are only an approximation of true fire behavior, and are inherently inaccurate. Fires are also difficult and dangerous to reproduce in a laboratory setting. A large, realistic wildfire would never be purposely started for the sake of measurement. This means that data on wildfires is hard to come by, and of low quality. A model may also be inaccurate if certain assumptions are broken. Many models assume that the region on fire is composed of a single fuel source, which is uniformly dense.

4.6 Future Fire Detection Work

Given these additional fire characteristics, it is only natural that our project's scope should expand and provide additional information, beyond the location of a fire's starting point.

Image recognition techniques could be used to measure the area of land covered by a fire. Similarly, image recognition could be used to describe the shape of a fire. This information is useful to fire fighters, because it allows them to predict the rate of spread, and the direction of travel of a fire. It will likely take first responders a non-trivial amount of time to arrive at the scene of a fire, and this information gives them some idea of what to expect. A fire will grow in the meantime between detection and fire fighter response.

5 SOLAR

In the interest of maximizing flight time, solar panels are mounted on the wings of the plane. This will provide additional power to the motor, and result in less power being drawn from the battery for a given motor speed. To convert the sun's energy as efficiently as possible, we will power the motor directly with solar panels, connecting them in parallel with the power management board's output. Figure 19 shows a high level block diagram of our power system. As you can see from the power system block diagram, we will place a diode between the solar array and the motor input. This is to prevent current flowing from the battery back into the cells and potentially damaging them. We also include a computer controlled switch that can shut off current from the battery to save energy when we are getting enough power from the solar cells.

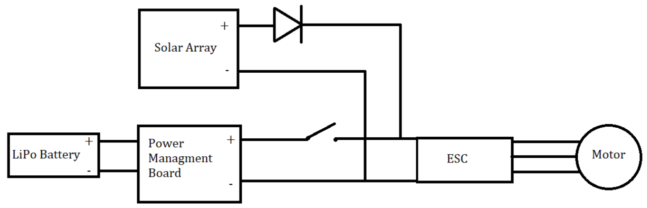


Fig. 19. Power System

5.1 Building the Solar Array

5.2 Aircraft Aerodynamic Issues

5.3 Solar/Battery Power Source Switching Circuit

A major component in the plane is a power switching circuit that chooses which source powers the motor based on inputs from a microcontroller. The idea is to switch into solar power whenever the plane has altitude to spare. It will remain in solar if array generates enough power to remain at cruising altitude. Otherwise it will use whatever power generated to slowly descend in a powered glide, and upon reaching a low altitude threshold, switch into battery power to climb back to a safe altitude. The first iteration of the circuit is in Figure 20. This

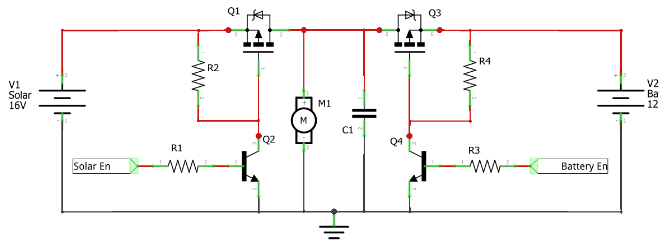


Fig. 20. Power source switch first iteration. Q1 and Q3 are SUP70101EL power MOSFETS, Q2 and Q4 are 2N3904 transistors. R1 is 180 $k\Omega$, R2 is 18 $k\Omega$, R3 is 180 $k\Omega$, and R4 is 12 $k\Omega$. C1 is a 300 μF capacitor that filters out noise when switching sources. Solar En and Battery En are 3.3V inputs from a microcontroller that enable solar or battery power via an open collector transistor configuration.

circuit worked in theory, but when tested, a major flaw was discovered due to the intrinsic body diode in Q3. When Solar En is high and Battery En is low, 16 volts is applied to the motor. If we focus our attention to Q3, we notice that 16 volts is applied to its drain, while 12.5 volts is applied at the source. Since the potential difference between 16 volts and 12.5 volts is enough to forward bias the body diode, Q3 is "turned on" even though the input is low! This shorts the two power sources and back drives the battery with a small amount of current. Back driving a battery is equivalent to charging it, but since we used lithium polymer batteries that have a high probability of combusting when overcharged, we opted to steer clear of uncontrollably charging the battery. A more problematic scenario would occur if battery power was on, and the solar array voltage was low enough to forward bias Q1's body diode. This would back drive the solar array and potentiall damage it. We migitagated this issue by adding transistors Q5 and Q6 in Figure 21. These were added to use body diodes

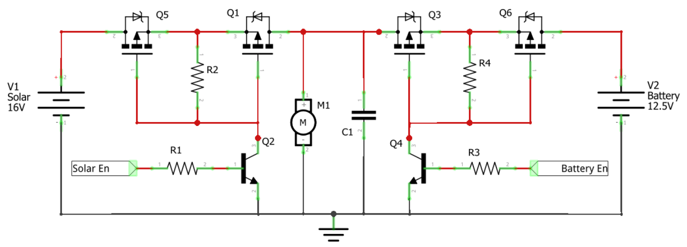


Fig. 21. Power source switch final iteration. Q1, Q3, Q5, and Q6 are SUP70101EL power MOSFETS; Q2 and Q4 are 2N3904 transistors. Note that the sources of Q1 and Q5; and Q3 and Q6 are tied together to prevent current backflow. R1 is 10 $k\Omega$, R2 is 18 $k\Omega$, R3 is 10 $k\Omega$, and R4 is 12 $k\Omega$. C1 is a 300 μF capacitor that filters out noise when switching sources. Solar En and Battery En are 3.3V inputs from a microcontroller that enable solar or battery power via an open collector transistor configuration.

to our advantage. By tying the sources of Q1 and Q5 (and Q3 and Q6) together, the body diodes opposed each other which effectively prevented any backflow of current. The first iteration also had an issue where transistors Q2 and Q4 did not operate deeply enough in the triode region. This caused one to blow during the testing phase. R1 and R3 were swapped with 10 $k\Omega$ resistors to alter the region these transistors operated in. See Figure 21's caption for the updates. This circuit was created using deadbug style soldering since the power lines needed to handle up to 30 amps of current. 12 awg copper wire and a large ground plane had more than enough capacity to handle the large current demands.

5.4 Flight Time Extension

5.4.1 Introduction

The circuit described above implements the switch that electrically connects the battery or solar panels to the motor. The circuit is driven by two inputs - Solar Enable and Battery Enable. This section will describe the software and hardware components of the system that drives these two inputs.

5.4.2 Altitude Control Algorithm

The heart of the system is a simple state machine which decides if the plane should be climbing or power gliding. This state machine is shown in the figure below.

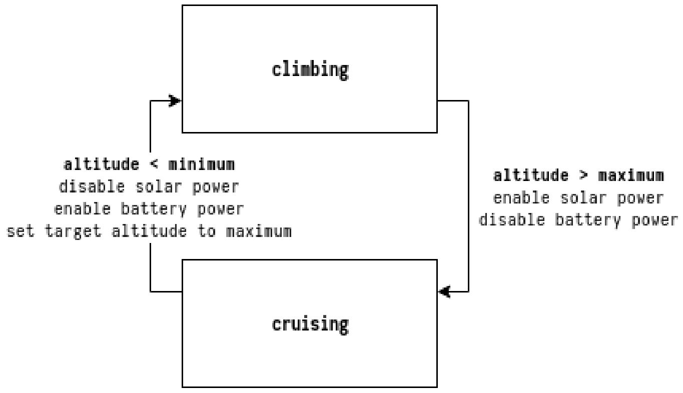


Fig. 22. While climbing, the plane exclusively uses battery power. The flight control system is commanded to fly to a target altitude. While cruising, the plane exclusively uses solar power. It will attempt to maintain the same target altitude, but will fall if insufficient solar power is available.

This state machine was implemented on a Teensy 3.1 microcontroller. The state machine drives two digital output pins on the Teensy, which are connected to Solar Enable and Battery Enable.

5.4.3 Measuring Altitude

The Pixhawk 4 Flight Controller (FC) comes with an internal barometer. However, the plane uses an external barometer to measure altitude. This was done because the Pixhawk's barometer was found to be extremely unreliable. The external barometer is mounted inside the plane, in a small slit cut into the foam of the plane's body. This prevents wind from affecting the altitude measurements. It is connected to the Teensy through I2C. The Teensy continuously polls the barometer for new altitude measurements. The Altitude Control Algorithm mentioned above transitions between states based on these measurements. Since the Pixhawk has a bad barometer, these measurements are also sent to the Pixhawk and replace the measurements of the internal barometer. The system which transmits altitude measurements to the Pixhawk is described in the section below.

5.4.4 Passing Altitude to Flight Controller

A UART link between the Teensy and Pixhawk allows the Teensy to set the Pixhawks's target altitude and as well as its estimated altitude. A state machine running on the Teensy constantly transmits these two altitudes, byte by byte, to the FC. A separate state machine running on the FC waits for target and estimated altitudes, and updates its internal navigation algorithm. Both altitude measurements are 32 bit floating point numbers, and are in units of millimeters. The transmission protocol is relatively simple, and only ensures that the Pixhawk and Teensy remain synchronized. It does not handle transmission failures, or re-synchronization in the event that the state machines are out of synchronization.

5.4.5 Results

The 3 plots in figure 24 show the results of a 20 minute test flight. The Altitude Control Algorithm was not used, and instead the solar and battery power sources were manually switched. The pilot of the plane manually climbed and power glided the plane as well. In effect, this was a test of the Altitude Control Algorithm. The flight time was greatly extended beyond what the battery was capable of providing on its own.

6 Looking Back

After spending roughly 6 months designing this system, there are a few things we could have done differently to get closer to reaching our goals.

First off, we should have begun our search for funding much earlier. We were self-funding everything at the start of our project, and trying to be as cheap as possible. This approach caused two major setbacks during our first 3 months; the homemade plane we built did not fly well, crashed, and was never used; and the first IR camera we purchased did not have anywhere near the range and resolution we needed to hit our minimum specifications. Had we received funding earlier, we could have purchased nicer equipment from the beginnig to save a few headaches.

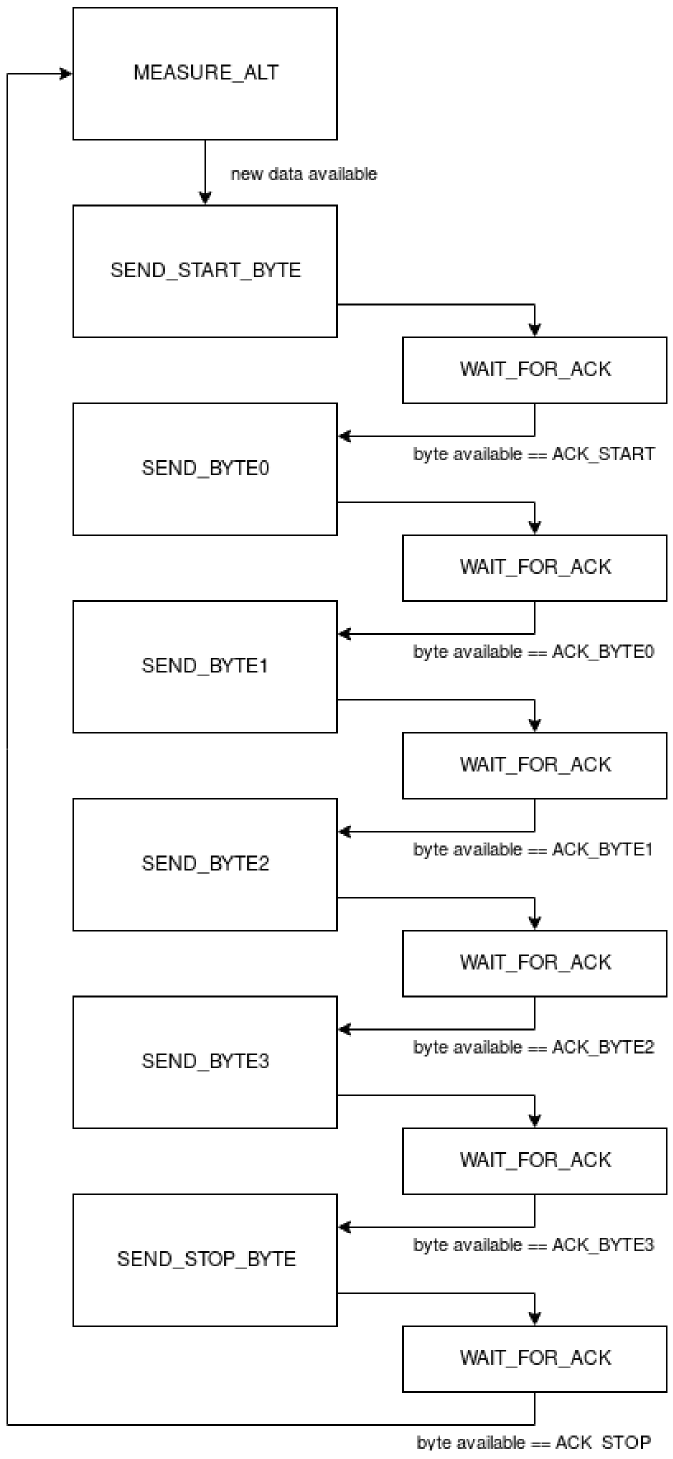


Fig. 23. The Teensy first measures the barometer's altitude, then sends a special 'start' byte. Upon reading this byte, the Pixhawk sends a corresponding acknowledgement byte. The transmission of the 4 data bytes proceed in this same pattern. Finally, a stop byte and stop acknowledgement are exchanged.

Secondly, the team should have spent time making an IR camera work for our specifications before writing a custom communication protocol to send fire detection data. One meme-

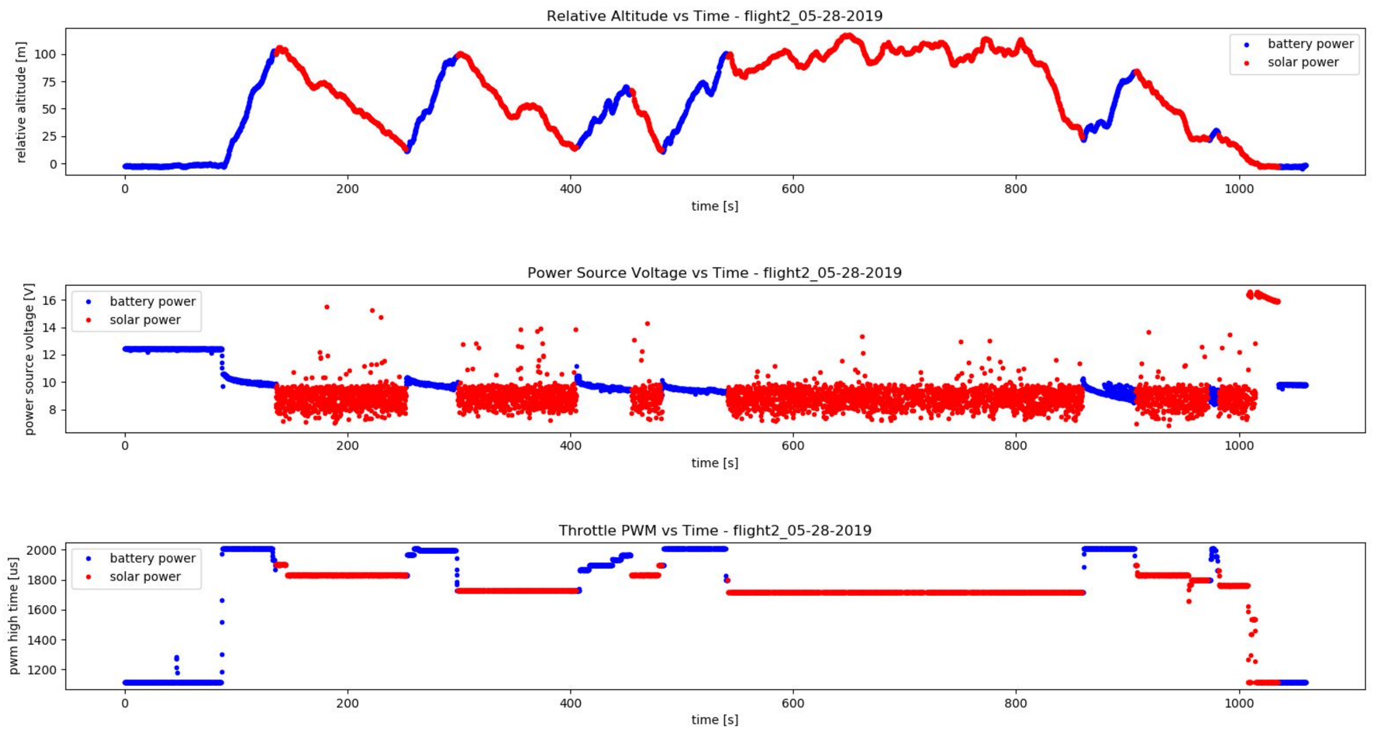


Fig. 24. The top plot shows the relative altitude of the plane over the course of a 20 minute flight. The middle plot shows the voltage of the currently active power source. Notice that the battery voltage is relatively stable, while the solar panel voltage is noisy, and frequency spikes up. The bottom plot shows the manual throttle input given to the plane. A PWM duty cycle of 2000 microseconds is 100% throttle, which was only possible on battery power. The highest possible input with solar power was about 1800 microseconds, or roughly 50% throttle. The sections in blue indicate when the plane was solely powered by the battery. The sections in red indicate when the plane was solely powered by solar panels.

ber spent the first full 3 month design period writing this protocol that ultimately was never used because we had hardware issues with our IR cameras. Next time, we will be sure to reorder the fire detection implementation process.

Next, we would have received better use out of the Pixhawk flight controller if we treated it more as a black box that did nothing but fly the plane autonomously. Many of our bottlenecks arose because we were trying to alter its open source code, ArduPilot. Every single modification made to ArduPilot turned out to be a major chore that always seemed to break something else unrelated to the new addition. And it did not help that the code was poorly documented and lacked organization. In the future, we know to get a second microcontoller such as a Teensy 3.2 much earlier. This will save us countless hours modifying and debugging code that other developers wrote.

Moving on, we learned that unobstructed airflow over a wing is imperative for efficient, safe flying. The first time we attached the solar array to the wing, we worked too quickly and did a poor job. This eventually led to a crash because solar cells detached mid-flight on one side of the wing. The aircraft fell out of the sky like a rock and blew apart upon impact with the ground. Fortunately we were able to purchase a new plane before the project ended, and it never crashed again. However, we could not gain back any of the lost time when we were grounded. The lesson learned is that haste makes waste.

Finally, the team did not spend enough test flying our plane. We assumed things would work well once we got all of the flight controller's parameters sorted out, but of course we were mistaken. Our troubles with the attitude heading reference system may have been resolved if we test flew and debugged in-flight problems more often. Part of the reason why we flew so little was due to our poor laboratory location. It took over an hour to get to the flying field and another hour to get back. On top of this, flights strictly on battery power only lasted ten minutes, and we had two batteries. A lot of time was wasted travelling to spend little time testing. Next time, we will be sure to better streamline our testing process, or design a project that can be more easily tested in our lab room.

7 Future Work

Future work includes designing a new airplane with a lighter airframe, a larger wing area with solar cells integrated inside of the wing, and more efficient solar cells. These three additions will surely put us over a 2 hour solar flight since the plane will require less speed (i.e. less power) to generate enough lift to maintain altitude. On top of this, we will purchase an IR camrea that suits our specifications well, complete more test flights, and put together a robust autonomous airborne fire detection system.

8 Conclusion

At the end of our first 10 week design phase, the team has accomplished a data logging system to capture important flight information such as aircraft attitude, and fire detection parameters. We are ready to integrate solar cells into the new, nice flying plane by utilizing the schematic in Figure 19; along with a full migration of the flight controller from the old plane. After running multiple experiments with our current fire detection method, we have learned that our cheap IR sensor will not provide the range necessary to meet our minimum specification. We are currently exploring other options, and are preparing to implement a new sensor come our final 10 week design phase.

Appendix A Electronics

A.1 Simple RC Configuration

Before motors and servos can be tested, the Taranis X9D RC transmitter must be bound to the FrSky X8R receiver so the two can communicate.

1) With the X9D off, hold down the F/S button on the receiver and power it up. Then let go of the F/S button. The light on the receiver should be solid RED indicating it is in bind mode.
2) Turn the X9D on and select the model you want to bind the receiver to.
3) Short press the menu button, and then press the page button to navigate to page 2 of the settings.
4) Scroll down with the '+' or '-' buttons until you see "Channel Range" and "Receiver No." settings.
5) Under "Receiver No.", highlight [bind] and press "ENT" to confirm. Leave the "Channel Range" as 1-8 with telemetry ON. Press "ENT" again.
6) The receiver's LED will flash GREEN and RED, and the transmitter will beep, indicating that they are binding.
7) After a couple beeps, press "EXIT" on the transmitter and power off the receiver.
8) Power the receiver back on. The LED should turn solid GREEN after a few seconds indicating that it is connected to the transmitter. The two are now paired together under the selected model forever unless the receiver is rebound to a different model.

With that done, servos and motors can be controlled by plugging them into the outputs on the X8R. Power everything off and plug the ESC and four servos into the receiver (see Table 1). Note that the two aileron servos are connected to only one channel using a servo Y-splitter. This causes the servos to behave opposite of one another as they would on a typical aircraft. The motor connects to the ESC via three wires with bullet connectors (see Figure 25).

TABLE 1
FrSky X8R Channels to Aircraft Control Surfaces

X8R Channel	Aircraft Control Surface
1	Ailerons
2	Elevator
3	Motor
4	Rudder

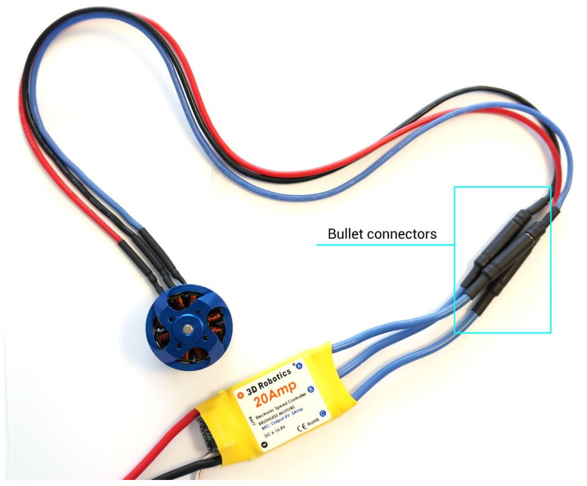


Fig. 25. Brushless motor connected to ESC via bullet connectors found it [9]

Now that everything is connected, follow these steps to center servos and test servo/motor functionality.

1) Ensure the propellers is OFF of the motor shaft, and the transmitter is powered off.
2) Turn on the transmitter.
    a) Ensure all switches are in default position and throttle position is at its minimum (default settings on the X9D will cause it to yell at you if these constraints are not met).
3) Plug the battery into the ESC.
    a) The servos will center right when they receive power for the first time. Make sure that the trim on all channels is centered.
4) Wait for the motor startup sound sequence to play.
5) Move sticks on the transmitter and observe control behavior.
    a) Note that servo direction can be flipped in the transmitter settings, so their directions are not too important at this point.
    b) If the motor spins in the wrong direction when the throttle is applied, switch any two of the three wires that connect it to the ESC.

In this stage, it is also wise to manually calibrate the ESC so it recognizes this specific transmitter's maximum and minimum PWM throttle outputs. Follow the steps below.

1) Ensure the propellers is OFF of the motor shaft, and the transmitter is powered off.
2) Place the throttle stick to its maximum position.
3) Plug the battery into the ESC.
    a) It will play a special sequence to indicate that maximum throttle has been detected.
4) Place the throttle stick to its minimum position within 2 seconds.
    a) The ESC will beep and play its normal startup sequence.
    b) If the normal startup sequence does not play and instead the ESC beeps abnormally, repeat this process and be sure to move the throttle stick to the minimum position within 2 seconds!
5) Calibration is now complete.

A.2 Connect Motors, Servos, and Receiver to Pixhawk

Refer to Figure 26 for the wiring diagram of all the electronics connected to the flight controller. Note that a special cable is required to connect the signal port of the LiPo Cell Voltage Monitor to the TELEM1 port on the Pixhawk. This can be purchased from Craft and Theory, or one can be made using an RS232 converter. We decided to buy the $15 connector. Also observe that although not intuitive, the Pixhawk's IO PWM Out is connected to the PMB's FMU PWM In. This is to map the RC outputs to the FMU header pins rather than to the M1-8 solder pads. This is not described very well in the Pixhawk's quickstart guide!

With all components connected, the first step is to run through Mission Planner's setup wizard. This involves calibrating sensors on the

Pixhawk, flashing it with the latest firmware, choosing the battery sensor module, and configuring the Taranis X9D so Mission Planner recognizes its PWM outputs. If incorrect PWM values are observed, or they do not change with stick movements, navigate to "Initial Setup" → "Servo Output". Then change each channel function from throttle, ailerons, elevator, and rudder to their respective RCINx channels as defined in Table 1.

The next step is to configure the flight modes in mission planner so switches on the X9D can cycle through different modes. If the preconfigured Flight Deck model is used (described in the next section), RC channel 5 will be mapped to switches SD and SC. Play with these to set the preferred flight mode selections. With everything powered on, when one of those switches is flicked, one can see the flight mode change from gray to green in Mission Planner. Now when in MANUAL or STABILIZE mode, and the flight controller armed (1 second press on the GPS safety switch), the X9D should have full control of the motor and servos!

A.3 Flash Taranis X9D with FlightDeck Software

The FlighDeck software from Craft and Theory can serve as a backup ground station directly on the transmitter. It provides everything necessary to fly the aircraft such as heading, altitude, battery voltage, and attitude. Follow the steps below to flash FlighDeck onto the transmitter after purchasing it from Craft and Theory. Note that a 40% discount is given on FlightDeck when it is purchased together with the TELEM cable described in the section above.

1) Download OpenTx Companion at https://www.open-tx.org.
    a) Run the companion executable.
    b) Select radio type 'FrSky Taranis X9D+' (or a different radio that is being used).
    c) Check the 'lua' box and ensure the 'sql5font' box is unchecked.
2) Download latest firmware version.
    a) We used this one: https://www.open-tx.org/2019/01/06/opentx-2.2.3.
3) Put Taranis in bootloader mode by holding both horizontal trims inward and powering on.
4) Connect Taranis to computer with mini-USB cable.
5) Flash firmware to radio using OpenTx Companion's handy user interface.
6) Navigate to SD card drive on the computer that the Taranis is plugged into. It is drive E:\on my laptop. It contains folder such ad LOGS, MODELS, and SOUNDS.
7) Copy all of these files and back them up in a safe location.
8) Download the latest SD card version at https://downloads.open-tx.org/2.2/sdcard/opentx-x9d%2B/.
9) Extract the contents of the .zip folder into the root directory of the SD card (in my case, drive E:\).
10) Download FlightDeck.zip from the email confirmation when it was bought or from the Craft and Theory account created for the purchase.
11) Extract the contents of the "SDcard" folder directly into the root directory of the Taranis SD card (in my case, drive E:\).
    a) Make sure to merge the contents of these folders and replace/overwrite any file already on the SD card.
12) Manually backup all models currently on the X9D to its SD card before proceeding to avoid any potential frustration.
    a) Long press 'Enter' over each model and select 'Backup'.
13) Add the preconfigured FlightDeck model to the transmitter using OpenTx Companion.

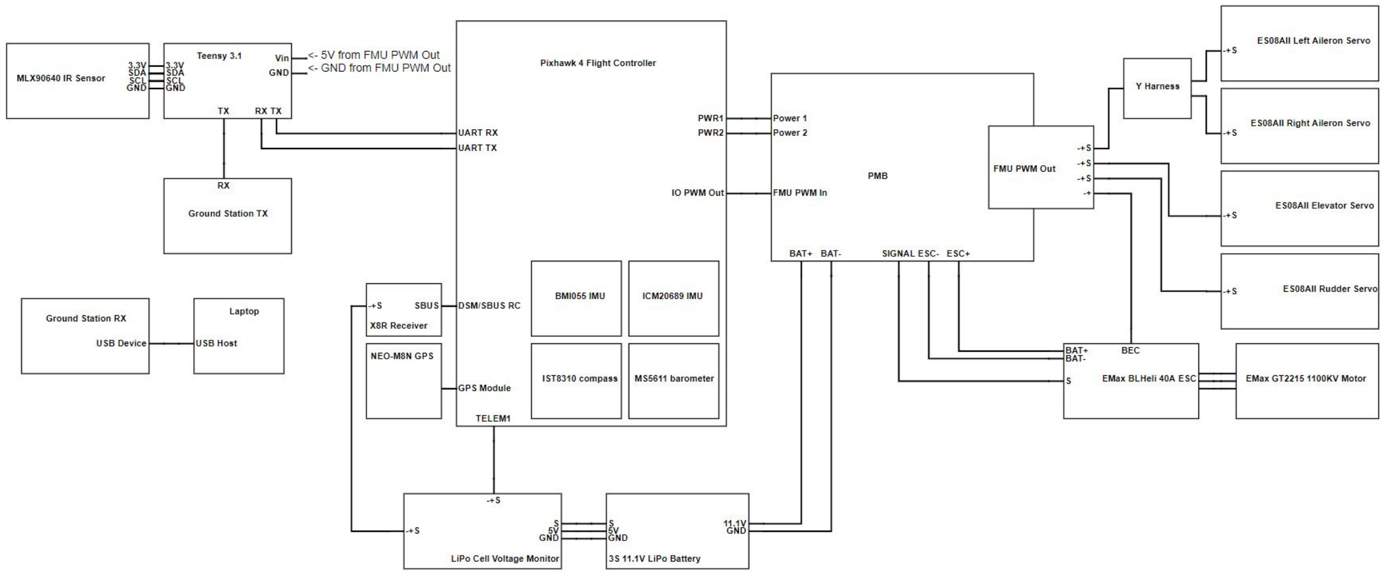


Fig. 26. Diagram of all electronics in the aircraft.

a) Open the X9D+.otx file that came with FlightDeck in OpenTx Companion.
b) Press the “Write Models and Settings to Radio” button.

14) Unplug the USB cable from the Taranis, select ‘Exit’, and confirm. The radio will now boot up normally.
15) Discover new sensors!

a) In Mission Planner, navigate to the full parameter list and set “SERIAL1” = 10 for FrSky SPORT Passthrough, OpenTx.
b) Disconnect from Mission Planner and unplug the micro USB cable.
c) Power the electronics with a battery and wait for the Pixhawk startup sequence to play.
d) Navigate to the Telemetry screen on the X9D and select the “discover new sensors” button.
e) Notice that GPS and CELS (battery lowest cell voltage), amongst others are now available.

APPENDIX B RADIO COMMUNICATION

B.1 SiK Radio LED patterns

The two radio modules on the aircraft, and the corresponding two on the ground station are all equipped with status LEDs. These LEDs blink in different patterns to indicate the state of the radio. When the green LED is blinking, the radio is searching for another radio, and is not yet capable of transmitting or receiving data. When the green LED is solid and not blinking, the radio has established a connection with another radio on its channel, and is capable of transmitting and receiving data. The red LED blinks each time a byte is transmitted or received. The red LED is solid and not blinking when the radio is in firmware update mode.

B.2 Modifying SiK Radio Channels

B.2.1 Introduction

By default, all 2 pairs of telemetry radios used on the aircraft were configured to use the same channel. This means that data sent through one pair of radios was accessible from the other 2 pairs of radios. To create multiple, independent lines of communication, it is necessary to separate the radios into individual channels. Each radio has a parameter stored in its EEPROM called its ’net ID’. This net ID is a single 8 bit integer. Radios with the same net ID can communicate with each other, and radios with different net IDs cannot.

B.2.2 Net ID modification procedure

The firmware of the radios may require a reflash to ensure compatibility. Download SiK HM-TRP. Install Mission Planner, which provides a tool for modifying radio parameters. To change the net ID of a pair of radios, follow this procedure.

1) Connect the local radio to your computer, and apply power to the remote radio.
2) Navigate to Mission Planner>Initial Setup>Optional Hardware>Sik Radio.
3) Click "Load Settings" and wait a while. The settings for both the local and remote radio should appear.
4) Change the net ID setting to a new value. Note that you have not yet changed the remote radio's net ID.
5) Click "Copy required to remote". The remote net ID should change. Note that you have not yet changed the remote radio's net ID.
6) Click "Save Settings" to write the net ID to the local and remote radios, and then wait a while. If you get an error message saying "command failed", don't worry.
7) Once the settings have been saved, close the configuration interface and unplug the radios.
8) Plug the radios back in and reopen the configuration interface.
9) Click "Load Settings", and wait a while.
10) The new net IDs should show up when the settings load.

ACKNOWLEDGEMENTS

Kodiak would like to thank the authors of Small Unmanned Aircraft, and Unmanned Aircraft Design for providing such detailed descriptions on UAV design and control. He would also like to thank all researchers that are apart of Glider soaring via reinforcement learning in the field [8] for conducting such excellent work that will contribute to his capstone project.

Max would like to thank Precision Hawk for giving us around two thousand dollars of electronic equipment for free. We would also like to thank the people on forums online that provided help describing ArduPilot functionality.

REFERENCES

[1] "ArduPlane Home," ArduPilot. [Online]. Available: http://ardupilot.org/plane/index.html. [Accessed: 06-Mar-2019].
[2] "MAVLink," Wikipedia, 08-Nov-2018. [Online]. Available: https://en.wikipedia.org/wiki/MAVLink#Packet_Structure. [Accessed: 05-Mar-2019].
[3] "Pixhawk 4," Basic Concepts · PX4 User Guide. [Online]. Available: https://docs.px4.io/en/flight_controller/pixhawk4.html. [Accessed: 06-Mar-2019].
[4] "Teensy v3.1 - 32 bit arduino-compatible microcontroller board," Velleman Spotlight. [Online]. Available: https://www.velleman.eu/products/view?id=420178&country=be&lang=en. [Accessed: 06-Mar-2019].
[5] "Banggood.com, "3DR Radio Telemetry Kit With Case 433MHZ 915MHZ For MWC APM PX4 Pixhawk for FPV RC Airplane RC Toys & Hobbies from Toys Hobbies and Robot on banggood.com," www.banggood.com. [Online]. Available: https://bit.ly/2JisfKa [Accessed: 18-Mar-2019].
[6] Beard, R. and McLain, T. (2012). Small unmanned aircraft. Princeton, N.J: Princeton University Press.
[7] C. Dillon, "Physicists Train Robotic Gliders to Soar like Birds", Ucsdnews.ucsd.edu, 2018. [Online]. Available: https://ucsdnews.ucsd.edu/pressrelease/physicists-train-robotic-gliders-to-soar-like-birds. [Accessed: 20-Feb-2019].
[8] G. Reddy, J. Wong-Ng, A. Celani, T. Sejnowski and M. Vergassola, "Glider soaring via reinforcement learning in the field", Nature, vol. 562, no. 7726, 2018. Available: 10.1038/s41586-018-0533-0.
[9] "Motor to ESC," Instructables. [Online]. Available: https://www.instructables.com/id/Beginners-Guide-to-Connecting-Your-RC-Plane-Electr/ [Accessed: 07-March-2018].
[10] Sadraey, M. (n.d.). Unmanned aircraft design.
[11] Electromagnetic Spectrum, Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Electromagnetic_spectrum
[12] Henry Cruz, Martina Eckert, Juan Meneses, Jose-Fernan Martinez, "Efficient Forest Fire Detection Index for Application in Unmanned Aerial Systems", 16 June 2016
[13] Ahmad Alkhatib, "A Review of Fire Detection Techniques", The University of South Wales, UK, 5 March 2014
[14] Chiachung Chen, "Determining the Leaf Emissivity of Three Crops by Infrared Thermometry", MDPI, 15 May 2015
[15] Allison, Johnston, Craig, Jennings, "Airborne Optical and Thermal Remote Sensing for Wildfire Detection and Monitoring", MDPI, 18 August 2016
[16] Albini, F. (1976). Estimating wildfire behavior and effects. USDA Forest Service, Intermountain Forest and Range Experiment Station, General Technical Report INT-30, 92 pp.

[17] David M. Doolin and Nicholas Sitar ”Wireless sensors for wildfire monitoring”, Proc. SPIE 5765, Smart Structures and Materials 2005: Sensors and Smart Structures Technologies for Civil, Mechanical, and Aerospace Systems, (17 May 2005)

[18] A. L. Westerling, A. Gershunov, T. J. Brown, D. R. Cayan, M. D. Dettinger, “Climate and Wildfire in the Western United States”, American Meteorological Society, May 2003

[19] I. Bosch, A. Serrano, and L. Vergara, “Multisensor Network System for Wildfire Detection Using Infrared Image Processing,” The Scientific World Journal, vol. 2013, Article ID 402196, 10 pages, 2013.

BIOGRAPHIES

Kodiak North is currently a senior at the University of California - Santa Cruz. He is majoring in Robotics Engineering, and looking for a career in the new drone industry once he graduates. On his free time, Kodiak enjoys surfing, repairing his truck, and flying his RC race quadcopter.

Maxwell Bradley is a senior at UC Santa Cruz. He is a computer engineering major specializing in computer system, which dabbles in both hardware and software. He will be continuing into the computer engineering MS program here at UCSC after he completes his bachelors, and hopes to work for himself after he graduates. When he isn’t laboring over some piece of code, he enjoys distance running, playing the guitar, and checking out breweries.

Zane Bradley is a 4th year Computer Engineering student at the University of California, Santa Cruz. He loves hiking, traveling, listening to music, and programming in his free time.