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

The design process requires large data bandwidth out of the plane for analysis on electrical and sensor readings. Because of this, creating a reliable data transfer pathway out of the plane is a huge priority. We went through a huge amount of different approaches, each with portability, reliability, and efficiency trade-offs.

2.2 System Components

2.2.1 Pixhawk Flight Controller

The flight controller is the main brain of the entire plane. 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 Teensy Microcontroller

The Teensy is a very small and light microcontroller with an easy interface with standardized protocols like I2C and SPI. For this part of the project, we used it to communicate over unreliable telemetry radio to get data out of the plane fast with minimal set up overhead.

2.2.3 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

TABLE 1
MAVLink structure [2].

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.

2.2.4 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.

2.2.5 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.

2.3 Results and Discussion

2.3.1 MAVLink Code Implementation

Our MAVLink data transmission system starts in the Pixhawk scheduler code within the main file called ArduPlane.cpp. The flight controller

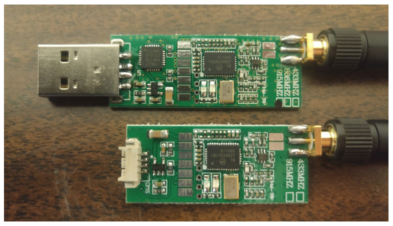
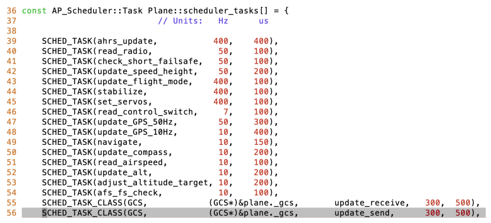


Fig. 1. 100mW 915MHz telemetry modules [5].

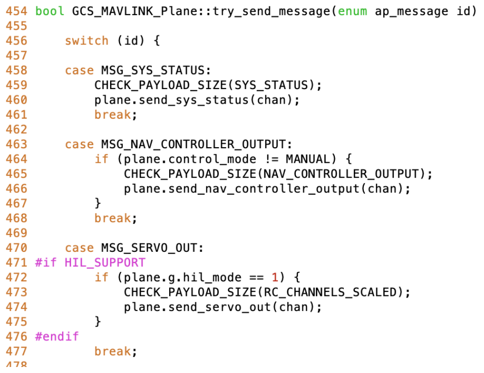
implements a scheduling system that figures out which process out of hundreds for the processor to run. The scheduler code can be found in figure 2.



```
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.

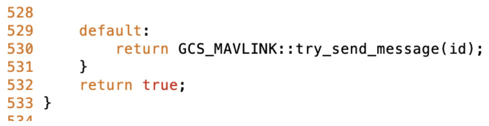


```
bool GCS_MAVLINK_Plane::try_send_message(enum ap_message id)

    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

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

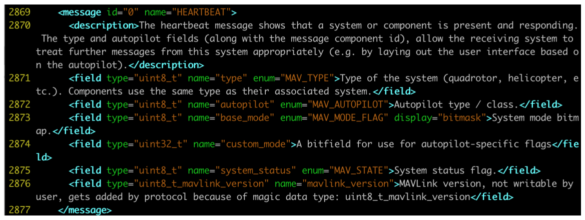
Figure 4 is the fallback function at the end of the try_send_message function that links the plane MAVLink implementation to the common MAVLink packets that all ArduPilot vehicles use. This function is contained in the library files that all ArduPilot vehicles have access to.



```

    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, we were able to make changes to the MAVLink messages that were being sent to the ground station. We 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). We 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" name="system_status" enum="MAV_STATE">System status flag.</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*

We 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.

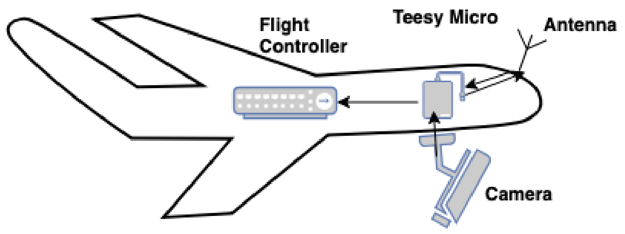


Fig. 6. Diagram of dual microcontroller setup

However, this original set up became less than optimal when the incredible wealth of the already existing ground control software became apparent. We discovered that the ground control software, Mission Planner, and command line tool, MavProxy, 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 at the time to focus on 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.

While this new set up removed complexity from the system, it placed a lot more stress on the flight controller. To reduce the stress on the flight controller, we decided to place the

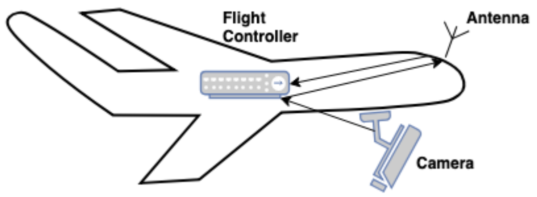


Fig. 7. Updated design without Teensy microcontroller

Teensy micro back on board. It would have been in charge of performing all fire detection if we had gotten the IR sensor to work. 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. We figured that this was far less stressful on the system than trying to squeeze image processing into an already tightly scheduled system. The third design layout can be found in figure 8.

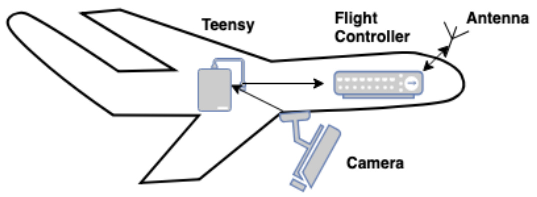


Fig. 8. Third design

Unfortunately, the huge amount of work to set up MAVLink packets to send even a byte of data became far too much for the team to deal with as we got closer to the deadline. Instead of using MAVLink for logging all information off the plane, we switched to using a bare UART protocol off of the Teensy microcontroller, sending data on an independent telemetry band. This made getting data logging far easier and with minimal set up time. Instead of running Python scripts with MAVLink li-

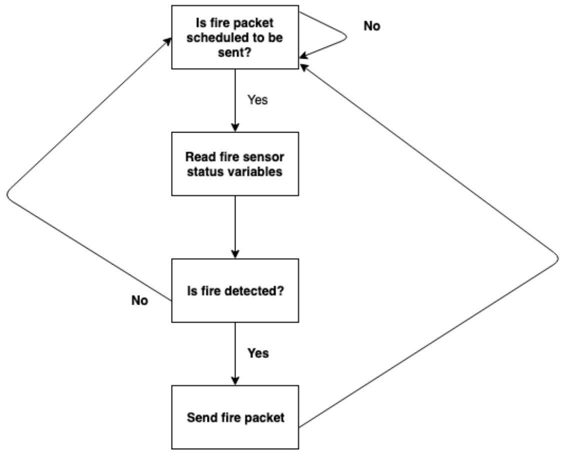

Is fire packet scheduled to be sent?
No
Yes
Read fire sensor status variables
Is fire detected?
No
Yes
Send fire packet

Fig. 9. Control flow of original fire packet transmission algorithm

braries, we parsed quickly through the data we received over the serial UART link. In addition, the hardware on board the flight controller we were using was not very accurate, so most of the data coming out of the Pixhawk was useless anyways. This is unfortunate, but taught us a huge amount about how MAVLink works.

This new lightweight design for real time flight data yielded information far faster than the old MAVLink set up. We were able to quickly make GPS plots after each flight if we desired, like the one in 10.

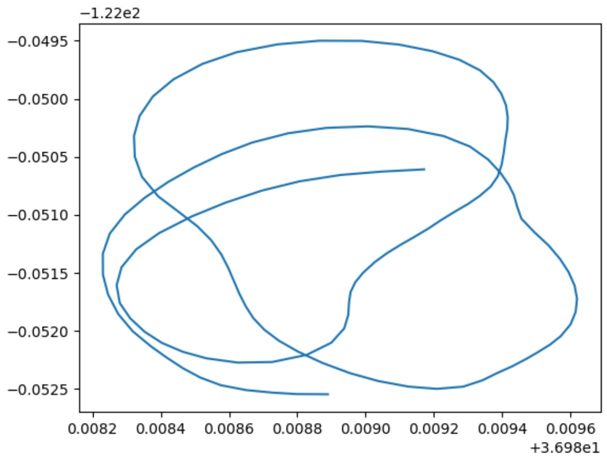


Fig. 10. Plot of the aircraft's maiden flight using GPS coordinates that were logged in real time via the lightweight UART telemetry link.

The final set up for the data transmission pathway can be seen in figure 11.

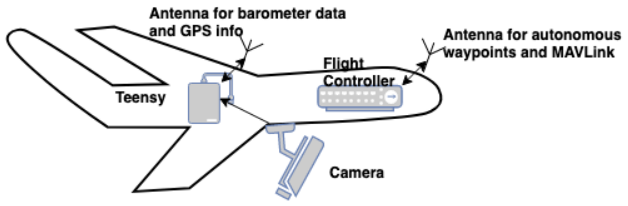

Antenna for barometer data and GPS info
Antenna for autonomous waypoints and MAVLink
Flight Controller
Teensy
Camera

Fig. 11. Final data transfer set up

2.3.3 *Design Problems*

One of the biggest problems we 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.

We realized this when we 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, we 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, we 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. This flight controller software was not well documented and almost impossible to debug. If we had to redo this project, we would have not touched the internals of the flight controller and really MAVLink at all.

Another huge issue with MAVLink was that the data that it was pulling was from the bad sensors on board the flight controller. The Pixhawk4 we had had a completely inaccurate barometer, which made our altitude control algorithm impossible to implement. The amount of time spent setting up MAVLink packets was really wasted trying to get garbage data out of a flight controller that should have been a black box.

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. However, we should have made a very reliable data pathway less of a priority and instead focused on the sensor network onboard the plane. We discovered that the MAVLink data transmission protocol is very hard to work with. We recommend that all future pursuers of aerial projects put use unreliable data transmission and focus on minimum specification rather than a TCP/IP protocol that will deliver data reliably.

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 12). 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.

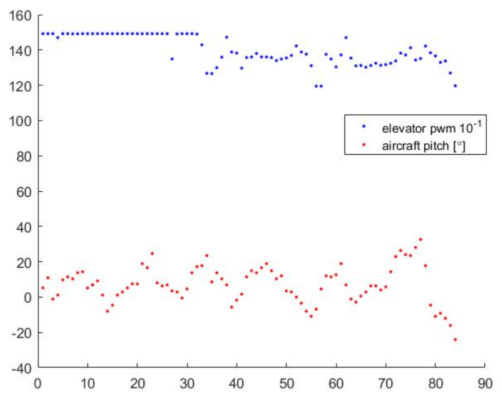


Fig. 12. This plot if of aircraft pitch (red) and elevator deflection (blue) in PWM over the flight duration. A decrease in elevator PWM equates to the control surface deflecting upward, i.e. it causes the aircraft to pitch up. Notice that only pitch up elevator was applied throughout the entire flight, 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 we proposed to have onboard the aircraft scans a stream of images for signs of fire. These images were to be captured by an Infrared camera. We finished up Winter quarter with a less sensitive IR module, and then came into Spring quarter with a more IR sensitive camera. The plan was to mount this new camera on the underbody of the aircraft fuselage, have it scan the ground below, and transmit over the data link if an unusual IR signature was found.

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

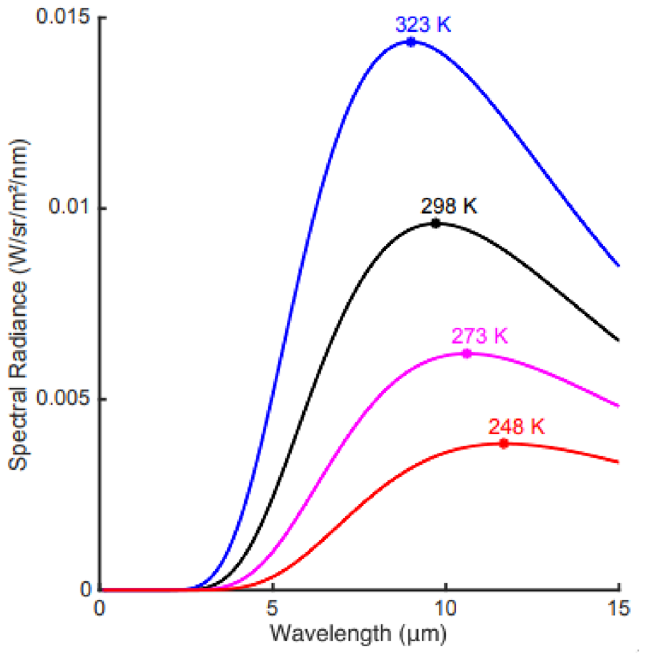


Fig. 13. Planck curves for temperatures 323K, 298K, 273K, and 248K. Notice that as temperature decreases, the peak wavelength increases and the total area under the curve decreases. This means that the temperature of an object can be detected, if we know the peak wavelength of its electromagnetic radiation.

4.3 Fire Detection System Implementation

Our first fire detection system made 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 the camera, and therefore cannot be detected by the system.

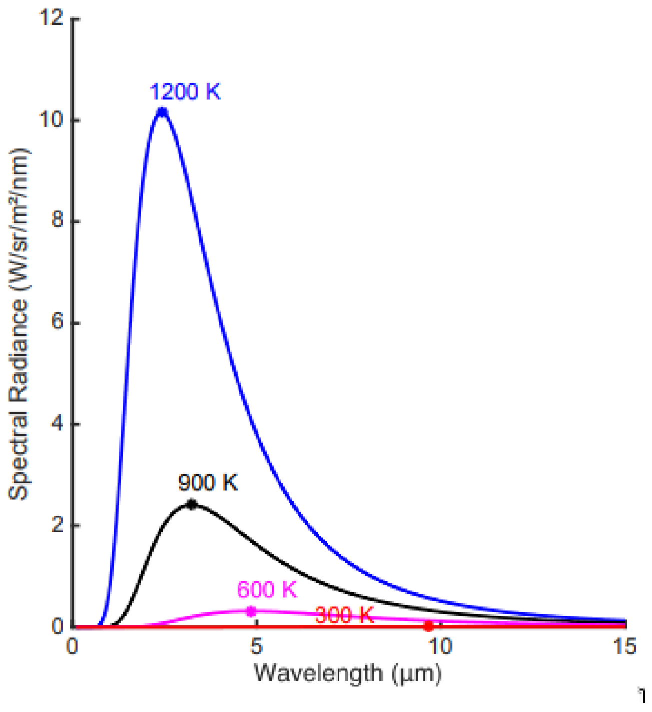


Fig. 14. Planck curves for temperatures 1200K, 900K, 600K, and 300K

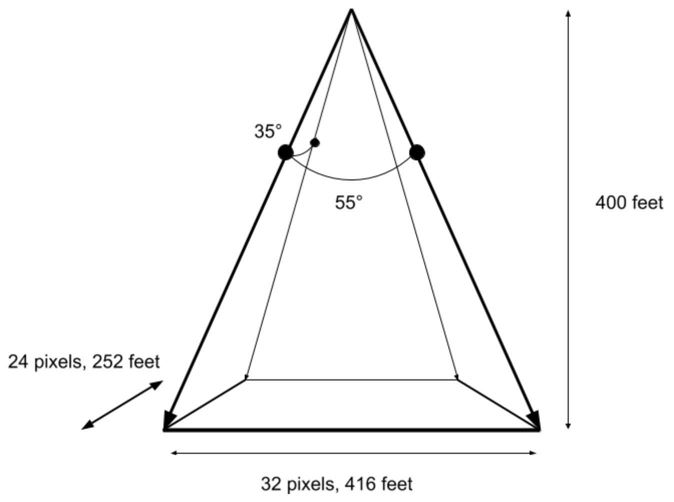


Fig. 15. From a height of 400 feet, images captured by the MLX90640 have a physical width of 416 feet and height of 252 feet. The FLiR Lepton however, in operation, features a sensitivity that is more than ten times higher than the MLX90640.

Because the sensitivty of this camera was not up to par with our minimum specification, we decided to upgrade to the FLiR Lepton IR camera. This camera featured a sensitivity of 0.03 meters squared per pixel, which was much higher than our minimum specification. The Lepton has a non-standard 16-bit I2C command communication line and a SPI line for sending image data to the master microcontroller.

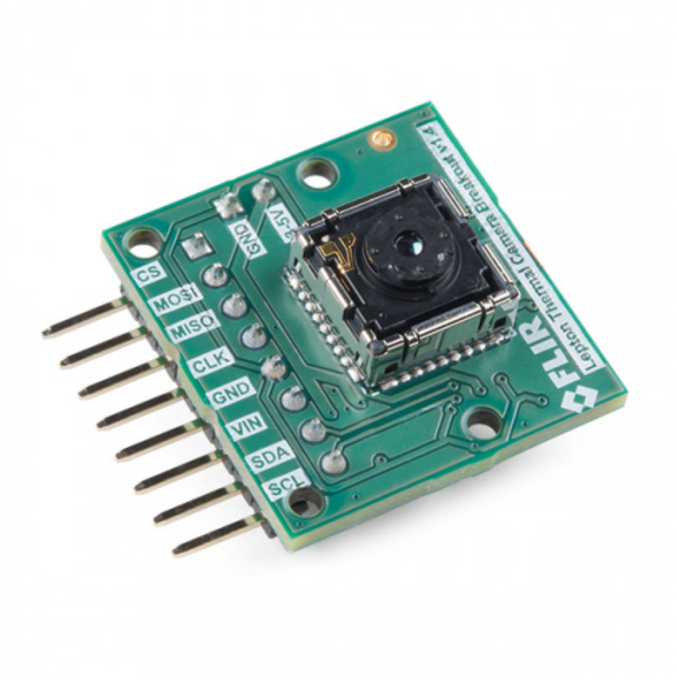


Fig. 16. The FLiR Lepton IR camera. This nearly $300 camera features a resolution of 0.03 meters squared per pixel from 400 feet in the air. It is the best intersection of cost and functionality that we could afford.

We purchased this camera and plugged it into the breakout board that it came with. We then spent the next month attempting to interface with it. We tried all the software libraries that FLiR recommended, writing our own library, and getting data out just using SPI. Unfortunately, we never received anything back from the camera besides a NACK.

4.4 Limitations and Problems

After spending a large amount of time attempting to interface with the camera, we decided that it had been shorted out by electrostatic discharge the second that we plugged it into the breakout board. This is unfortunate, but highly common. Other senior design teams lamented about the sensitivity of these modules to ESD and how fragile yet expensive they are.

However, the shorting of the camera was bad luck and wasn't a difficult part of our minimum specification. If we had a working camera, getting fire detection going would have been as easy as writing a simple algorithm to

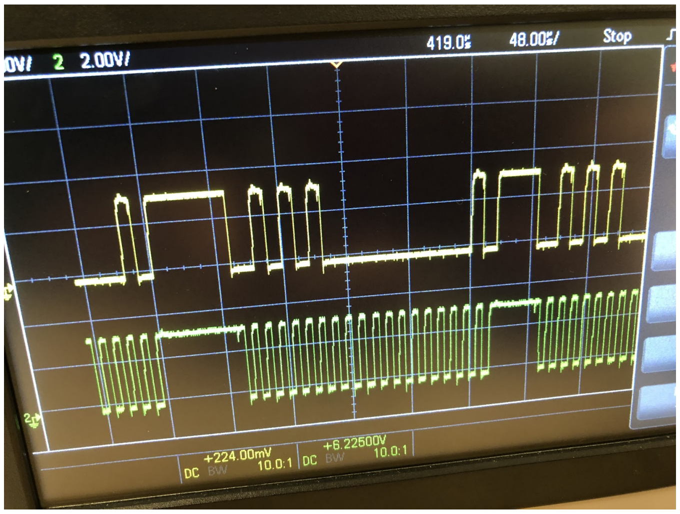


Fig. 17. This oscilloscope output shows the serial data line on top and the clock line on the bottom. As per standard I2C device functionality, the master transmits the slave address (0x2A for the Lepton) on the line and waits for the device to respond by pulling the line low on the clock cycle highlighted in red. Unfortunately, in this case, the device never responded and left the line high, which constitutes an I2C NACK.

count the number of pixels reading a high heat signature and mapping that to an area based on our altitude. This camera should have been plug and play, but, due to unforseeable bad luck, ended up being a waste of time.

4.5 Future Fire Detection Work

Given a functional IR camera, fire detection would have been relatively easy. We could have borrowed external libraries to run image recognition techniques that could give more information to first responders. 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, we made the design decision to use the sun as an additional power source to power our motor. The suns photons are converted into electrical energy by solar cells mounted on the wings of the plane. The solar array could not generate enough power alone to maintain our planes altitude, so it acted as a way to limit the amount of power the battery used.

5.1 Choosing Solar Cells

When choosing which solar cells to use in our design, we had to consider several constraints. Firstly, the cells had to be lightweight. All additional weight that the plane carries increases the power output needed to maintain altitude. A heavier plane requires more lift to counteract gravity, which means a higher airspeed is needed to generate that lift, which means the power draw of the motor must increase. Secondly, the solar cells must be high efficiency. The efficiency of a solar cell is the fraction of photon energy hitting its surface that is successfully converted into electrical energy in the form of a voltage and current. Because an airplane wing has curvature, the cells need to be able to flex slightly to match the shape of the wing. Our last constraint was cost. We had a finite budget, and sought to meet our other constraints as best as possible, while remaining within our budget. The cells we decided on were SunPower C60s. They are lightweight, at 10 grams each, have an efficiency of 22.5 percent, have some flexibility, and cheap enough to fit our price range.

5.2 Building the Solar Array

Before constructing the solar array and attaching it to the wing, we had to decide on how many to use, and how to wire them. To maximize the power output of the solar array, we placed as many cells as would fit onto the wing, without any hanging off, or being bent to the point they would crack. Complying with these constraints, we fit a total of 26 cells on the wing. We decided to wire the solar cells in series, both because it was simple, and gave an output voltage above the motors voltage requirement. The mounting of the cells involved soldering the entire array of cells on a table, then carefully applying glue to the cells and placing them on the wing. Below is a diagram of our solar array layed out on the plane wings.

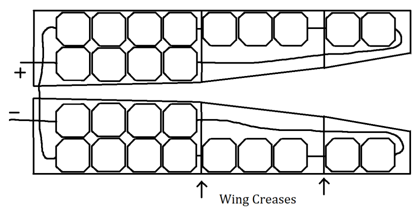


Fig. 18. Solar Cell Layout

5.3 Aircraft Aerodynamics

We learned an important lesson in aerodynamics after a solar cell came partially detached during a test flight, and caused the plane to crash. Fast moving air had caused an upward force to be exerted on the cell, and ripped it off of the wing. Back in the lab, we took some time to carefully reconsider the way we attached our solar cells to the wing. To keep the flight characteristics of the plane from being altered, and to prevent the solar cells from detaching, we needed to ensure the cells flexed to match the curvature of the wing, and that no air would be able to flow underneath the cells. To ensure that the cells matched the curvature of the wing, we weighed down the entire area of the cell with sandbags while the glue dried. We had previously used spare batteries and other lab equipment to weigh down the drying cells. We took an additional precaution by laying a line of transparent packing tape around the edges. This was to keep fast moving air from reaching underneath the solar cells and potentially breaking them off.

5.4 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 19. This

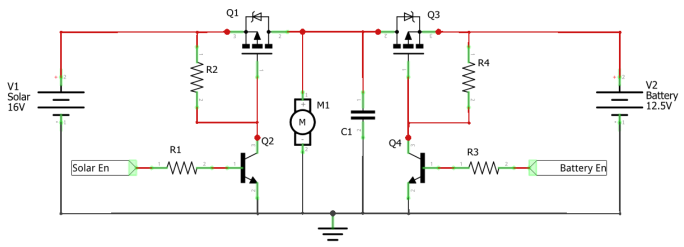


Fig. 19. Power source switch first iteration. Q1 and Q3 are SUP70101EL power MOSFETS, Q2 and Q4 are 2N3904 transistors. R1 is 180 $k\Omega$, R2 is 18 $k\Omega$, R3 is 180 $k\Omega$, and R4 is 12 $k\Omega$. C1 is a 300 μF capacitor that filters out noise when switching sources. Solar En and Battery En 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 potentially damage it. We mitigated this issue by adding transistors Q5 and Q6 in Figure 20. These were added to use body diodes to our advantage. By tying the sources of Q1

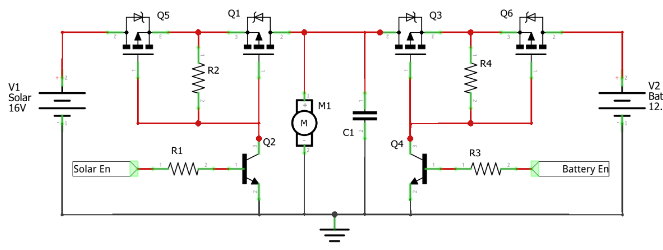


Fig. 20. Power source switch final iteration. Q1, Q3, Q5, and Q6 are SUP70101EL power MOSFETS; Q2 and Q4 are 2N3904 transistors. Note that the sources of Q1 and Q5; and Q3 and Q6 are tied together to prevent current backflow. R1 is 10 $k\Omega$, R2 is 18 $k\Omega$, R3 is 10 $k\Omega$, and R4 is 12 $k\Omega$. 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.

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 20's caption for the updates. This circuit was created using dead bug 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.

5.5 Flight Time Extension

5.5.1 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.

5.5.2 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.

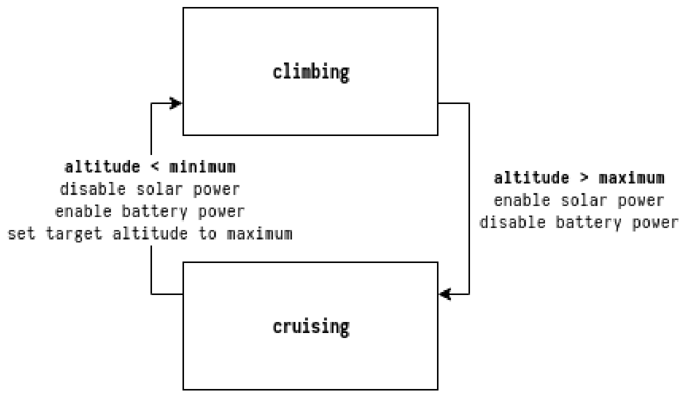


Fig. 21. While climbing, the plane exclusively uses battery power. The flight control system is commanded to fly to a target altitude. While cruising, the plane exclusively uses solar power. It will attempt to maintain the same target altitude, but will fall if insufficient solar power is available.

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.

5.5.3 *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.

5.5.4 *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.

5.5.5 *Results*

The 3 plots in figure 23 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 beginning to save a few headaches.

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 member spent the first full 3 month design period

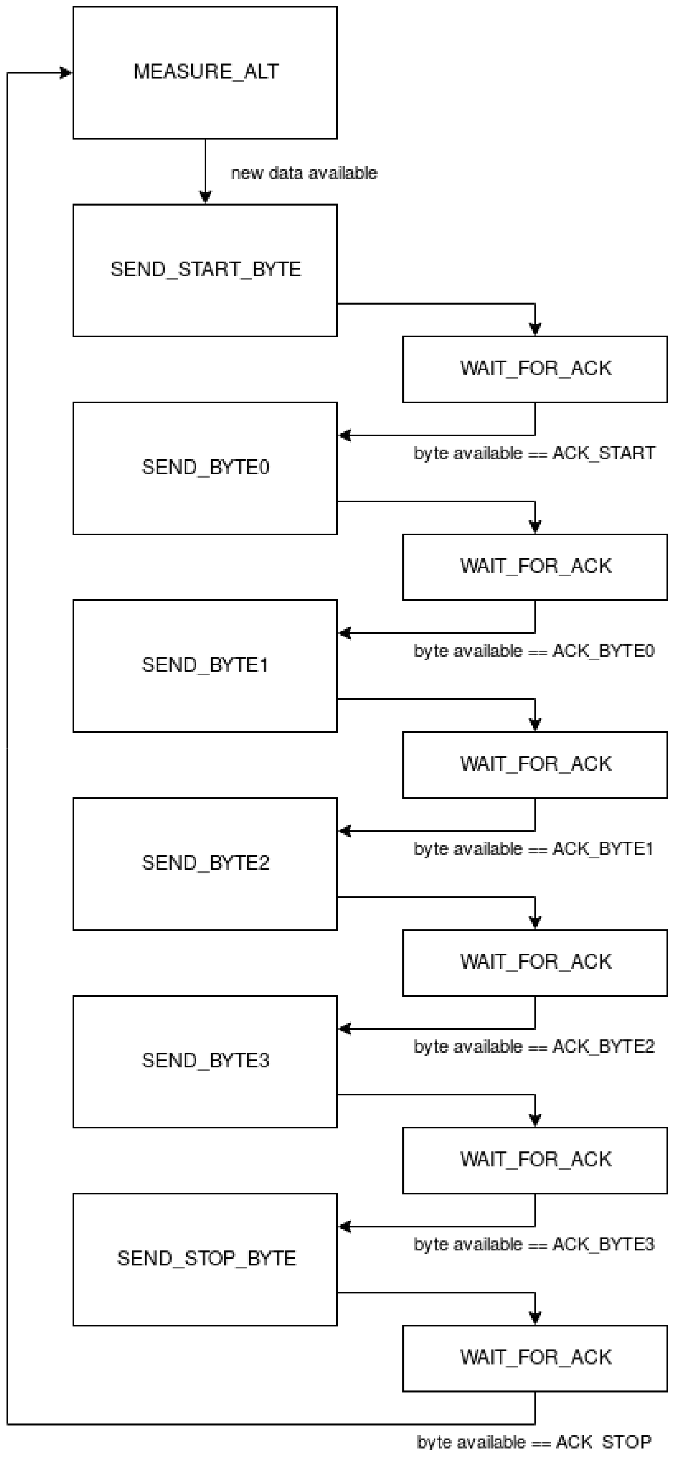


Fig. 22. The Teensy first measures the barometer's altitude, then sends a special 'start' byte. Upon reading this byte, the Pixhawk sends a corresponding acknowledgement byte. The transmission of the 4 data bytes proceed in this same pattern. Finally, a stop byte and stop acknowledgement are exchanged.

writing this protocol that ultimately was never used because we had hardware issues with our IR cameras. Next time, we will be sure to re-order 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 microcontroller 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.

alt_vs_time_flight2_05-28-2019.jpg

Fig. 23. 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.

7 Conclusion

With all of our work put it the last six months, we have produced an autonomously flying solar powered aircraft. The team successfully near tripled the flight time using a style of flying where battery power is consumed only when the plane is climbing. Unfortunately, our design lacks any fire detection system because we ran into issues with every IR camera we purchased, but the platform is totally ready for the addition of a small IR camera. 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 camera that suits our specifications well, complete more test flights, and put together a robust autonomous airborne fire detection system.

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 2). 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 24).

TABLE 2
FrSky X8R Channels to Aircraft Control Surfaces

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

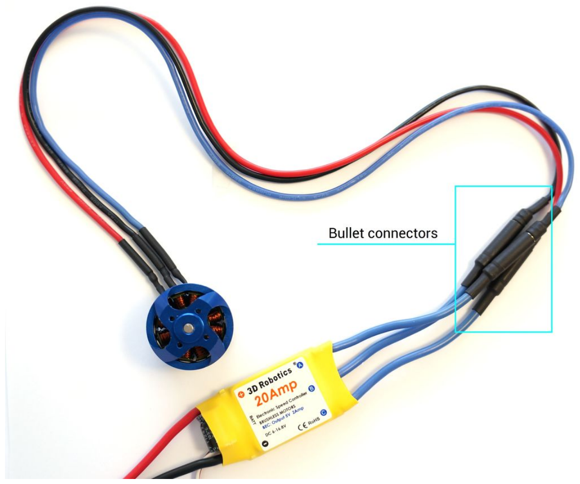


Fig. 24. Brushless motor connected to ESC via bullet connectors found it [9]

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 25 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" $\rightarrow$ "Servo Output". Then change each channel function from throttle, ailerons, elevator, and rudder to their respective RCINx channels as defined in Table 2.

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!

hardwareBlockDiagram.jpg

Fig. 25. Diagram of all electronics in the aircraft.

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.

    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 radios net ID.
5) Click Copy required to remote. The remote net ID should change. Note that you have not yet changed the remote radios 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, dont 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 freestyle 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.