Data transfer, Control Systems, and Sensors CMPE 185 Final Project

Maxwell Bradley, Kodiak North, and Zane Bradley

Abstract—The purpose of this document is to teach someone the basics of EDIT THIS BRO. Readers will be able to insert tables, figures, mathematical formulas, bibliographies, and references by the end of the tutorial.

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

CONTENTS

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

An important part of this project is the data transfer from the plane to the ground station. 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. This section of the paper will talk about the communication of data between the in-flight aircraft and the ground control station read by our product user.

2.2 Materials and MAterial Background

A multitude of different components make up the data transmission path between the plane itself and the ground station. Each of those parts will now be discussed in detail.

2.2.1 Pixhawk Flight Controller

The Pixhawk flight controller is the main brain of the entire plane. It receives IR information from the IR sensor mounted downwards pointing at the ground through a serial communication port (using the I2C protocol). The Pixhawk is where the majority of the data transfer work occurs in the second version of the data transmission design. It uses an overall system scheduler to decide when to send MAVLink packets to the ground station. Within the function that is scheduled in the overall scheduler, a smaller MAVLink scheduler exists. Within the MAVLink scheduler, 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, respectively. Its packet structure is described in the following picture.

Field name	Index (Bytes)	Purpose
Start-of-frame	0	Denotes the start of frame transmission (v1.0: 0xFE)
Payload-length	1	length of payload (n)
Packet sequence	2	Each component counts up their send sequence. Allows for detection of packet loss.
System ID	3	Identification of the SENDING system. Allows to differentiate different systems on the same network.
Component ID	4	Identification of the SENDING component. Allows to differentiate different components of the same system, e.g. the IMU and the autopilot.
Message ID	5	Identification of the message - the id defines what the payload "means" and how it should be correctly decoded.
Payload	6 to (n+6)	The data into the message, depends on the message id.
CRC	(n+7) to (n+8)	Check-sum of the entire packet, excluding the packet start sign (LSB to MSB)

Fig. 1. Max B1

The start frame transmission is the hex character set 0xFE, which tells the ground station that the packet is incoming. 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.

The most important part of this packet is the message id, the 5th byte in the index. This byte signals which MAVLink message the incoming data corresponds to. MAVLink messages are defined in a file called common.xml. 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. As a result, the team has become very familiar with its function.

2.2.3 ArduPilot

After looking at all the free and open source autopilots out there on the market, Ardupilot, an open source autopilot that runs on the Pixhawk4, was selected. 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. It's 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 meat for the ArduPlane module is contained in the libraries. The programmers for ArduPilot wisely chose to combine the largesse of the code in libraries which all different vehicle versions that run ArduPilot (ArduPlane, ArduSub, ArduTractor, etc.)

use. These libraries contain code for sending MAVLink 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 916MHz. The flight controller uses this link to send flight data. The ground station sends GPS coordinates for the plane to track to on over this link as well. A picture of this module is below.

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.

3.0.1 System Set Up

Our data transmission system starts in the Pixhawk scheduler code in 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 of the hundreds of processes competing for the processor to run. This is what the scheduler looks like in code:

```
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. Max B3

The highlighted line of code 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 is where changes started:

```
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. Max B4

As a backup to the above function, 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 overall project structuring, but made finding the actual implementation of MAVLink sending very challenging.

Below is the fallback function at the end of the previous function that links the plane implementation to the common MAVLink packets that all ArduPilot vehicles use. This function is contained in the library files that all ArduPilot vehicles contain.

```

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

Fig. 4. Max B5

By editing code in either the library function or the ArduPlane specific function, it was possible to make changes to the MAVLink messages that were being sent to the ground station. Changes were made to 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). To edit MAVLink message definitions, changes needed to be made in the common.xml or ardupilotmega.xml files.

3.0.2 Design Process

The design process was begun by trying to prevent the flight controller from having to handle any of the data transmission at all. Instead, the 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 was done 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 looked like this:

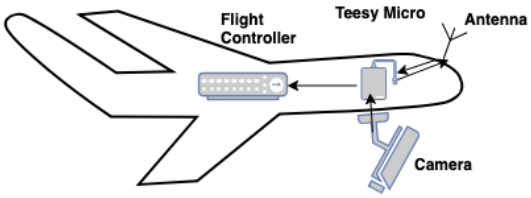

Fig. 5. Max B6

However, this set up became less than optimal when the incredible wealth of the already existing ground control software became apparent. It was discovered that the ground station software, Mission Planner, could be used to send GPS waypoints to the plane which it would then track to. The interface between the flight controller and the ground control station would only work if it could receive bytes from the ground station, and only allowing the Teensy access to the ground station would prevent this huge benefit from being realized. It didn't make sense to spend a lot of time designing a system where the Teensy read in fire detection data forwarded them to the flight controller. 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, replaced with a more overloaded flight controller. A diagram of the current set up is below.

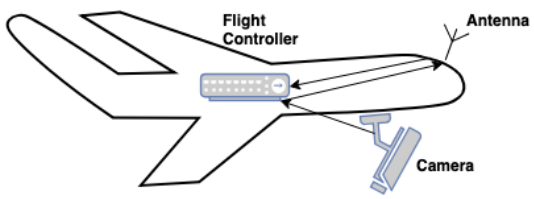

Fig. 6. Max B7

After the flight controller became the center of data processing and transmission, 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 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 will be covered in-depth in the problems part of this discussion, it was necessary to overwrite one of the existing MAVLink definitions. The MAV_CMD_NAV_LAND with ID 21 message was selected because the plane should never be allowed to land automatically This message was overwritten with a custom packet that contained fire detection data. This packet would only be sent when the sensing apparatus actually detected a fire, even though the message would be scheduled for sending at least once a second. The simple control flow of this setup is as follows.

For the ground station that would receive messages, a Python script that used the Python package PyMavlink's submodule mavuti was used. This allowed a reading of the serial port and a parsing/translation of the incoming MAVLink message into human readable data. The associated data was then used to make plots, such as the following that plots latitude and longitude from the incoming GPS data packets.

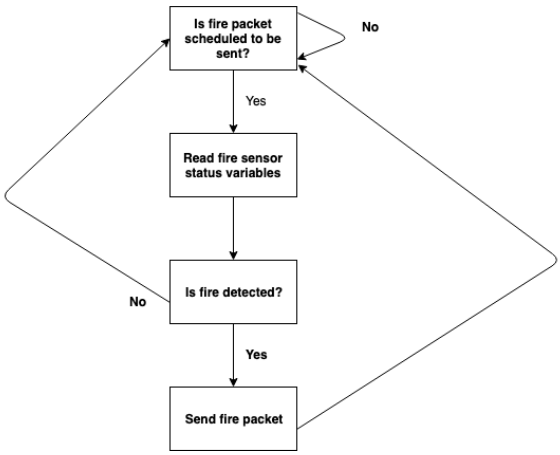

Fig. 7. Max B8

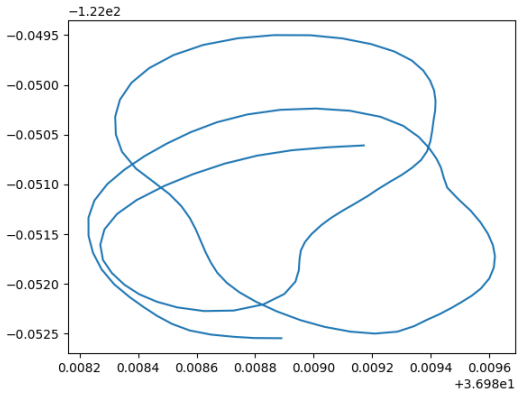

Fig. 8. Plot of the aircraft's maiden flight using GPS coordinates that were logged in real time via the telemetry link.

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.

3.0.3 Design Problems

The biggest and most enveloping design problem that was 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 this 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 would not be able to be sent by the current MAVLink protocol. Originally, a custom packet with an ID value of 11065 was created and the sending function for that specific packet was called. 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 being used rather than the updated and more flexible MAVLink version 2. Overwriting the message was a necessary evil to send custom packets unfortunately.

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

4 Data Transfer Conclusion

Data transmission software is essential to pulling real time data that closely reflects and explains the events that occured 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.

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.

5 Control System

5.1 Introduction

The control systems section will explore the mysterious art of controlling planes without a

pilot on-board. The overall plan is to control the aircraft in an optimal way to detect wildfires, but for now, we are just focused on control in general.

6 Project Conclusion

LaTeX is a very useful tool for generating professional PDF documents. Its equation editor makes it simple to add lengthy mathematical equations in a short amount of time, and allows for images to be inserted and BRU NEEDS AN EDIT. There is a learning curve to the language, but hopefully by now readers are familiar with the basic commands and terminology. Thanks to the World Wide Web, tutorials and forums are available in seconds to help learn any advanced LaTeX commands. Thanks for reading!

Appendix

A large WHAT EVEN IS AN APPENDIX list of LaTeXmath symbols can be found in [**?**].

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 ([7]) for conducting such excellent work that will contribute to his capstone project.

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] Beard, R. and McLain, T. (2012). Small unmanned aircraft. Princeton, N.J: Princeton University Press.

[6] 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].

[7] 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.

[8] Sadraey, M. (n.d.). Unmanned aircraft design.

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.