DESIGN REPORT: RMAX

Wireless Communications

Author:

Adam Mazzella

Table of Contents

1 Introduction

1.1 Project Overview:

The goal of the RMAX wireless communications team is to establish an in flight communications link between the Yamaha RMAX helicopter and ground station to send and receive recorded flight data. Data including GPS coordinates for the RMAX will be distributed by the ground station, and used by other subsystems to track a specified on ground target. Once the target is located a UGV will be deployed for rendezvous as well as a UAV supply drop in simulation of common search and rescue scenarios. The following document outlines our teams collaborative design effort to establish communications.

1.2 Client:

Our client Northrop Grumman had a goal in mind of providing a diverse group of students with a large, system-level, and multidisciplinary design experience. They had the desire that teams of students and faculty from Cal Poly San Luis Obispo and Cal Poly Pomona will work to develop a system that involves coordination between two unmanned aerial vehicles (UAVs) and one unmanned ground vehicle (UGV). To

1.3 Stakeholders:

1.3.1 Future Generations:

Future generations of the RMAX project will inherit and rely on our success and documentation. So that the project can continue and grow in efforts to improve UAV and UGV search and rescue.

1.3.2 Search and Rescue:

It may be possible that through our project we are able to develop a system that can improve current search and rescue techniques and save lives in the process.

1.4 Goals and Objectives:

1.4.1 Project Goals:

The project was envisioned to locate a simulated target from a UAV and have its location communicated a UGV as well as to a second UAV. The first UAV, a fixed-wing airplane, will determine the target location using computer vision systems and other geo-location techniques while flying autonomously. The relevant information pertaining to the expected location of the target will then be used by the second UAV, such as the Yamaha RMAX helicopter, to autonomously fly to the target location and drop a package. Once it arrives at the target location, the UGV will then use the expected target location to independently identify the location of the simulated target and autonomously navigate to the target for close-up inspection and possibly supply an emergency package.

1.4.2 Wireless-Communications Goals:

The goal of the wireless communications team is to design and implement a wireless system architecture that would allow communication between our unmanned vehicles and ground station. This would be done using directional wireless antennas, XBee RF modules, and a TCP Server-Client software architecture.

1.4.3 Personal Goals:

My personal goals as lead programmer were in collaboration with other members of the Wireless-Communications team to design and code the TCP-Server client software architecture using C++ and Visual Studio. Upon completion of this goal it was my responsibility to test and integrate our directional wireless antennas, and XBee RF modules to allow communication between subsystems.

1.5 Deliverables:

1.5.1 Project Deliverables:

It was the responsibility of the project to deliver a successful collaborative live implementation and demonstration of a search and rescue scenario meeting the specifications requested by our client Northrop Grumman.

1.5.2 Wireless-Communications Deliverables:

It was the responsibility of the wireless communications team to deliver a modular subsystem that would allow quick integration and wireless communication between current and future subsystems for the project.

1.5.3 Personal Deliverables:

It was my responsibility to deliver a modular software architecture written in C++using Visual Studio that could be used successfully to allow the integration and communication between subsystems.

1.6 Duration

1.6.1 Project Duration:

This years iteration of the RMAX project was given 8 months to design and implement the requested project. Starting in Fall quarter September 2012 and continuing until Spring quarter April 2013 students worked in collaboration to complete the project.

1.6.2 Personal Duration:

I joined the project as lead programmer during Winter quarter January 2013 and worked until our deadline of April 2013. During this time I collaborated and designed a modular software architecture for the project.

2 Related Works

2.1 United States. National Coordination Office for Space-Based Positioning, Navigation, and Timing. GPS.gov:. National Coordination Office for Space-Based Positioning, Navigation, and Timing, 17 Feb. 2012. Web. 15 Feb. 2013. <http://www.gps.gov/systems/gps/performance/accuracy/>.

According to the National Coordination Office for Space-Based Positioning, Navigation, and Timing most GPS signals without outside post processing have an accuracy of between three and seven meters. Although originally civilian GPS was limited in accuracy by design this is no longer the case and accuracy now depends on factors such as disruption by the ionosphere, clock differences in the GPS receiver, and signals bouncing off surfaces before they get to the receiver

2.2 "GPS Accuracy and Limitations." *GPS Accuracy and Limitations*. Earth Measurement Consulting, n.d. Web. 15 Feb. 2013. <http://earthmeasurement.com/GPS_accuracy.html>.

Earth measurement gives an in depth analysis of the different forms of GPS and their accuracy. Included in this assessment are normal GPS without any enhancements which has the worst accuracy, wide area differential GPS which uses extra information and calculation to improve accuracy to around three meters, and real time kinematic which uses a base station radio link to geometrically correct for errors. Finally, post processing GPS is discussed but this is less important for us because it is not in realtime. After analyzing this report I can see that with just unmodified GPS we will be able to acquire the necessary to find our target without the addition of extra calculation.

2.3 "Making an IR Object Tracking System." *Let's Make Robots!* N.p., 15 Jan. 2009. Web. 16 Oct. 2012. <http://letsmakerobots.com/node/4428>.

This talks about a persons project to track using IR. It goes into detail about the advantages and disadvantages of using IR, and includes circuit diagrams and pictures

for reference. This could be used to consider what we would need and encounter if we were to use IR to make our GPS system more accurate. The downside is this project seems to be oriented more for robots and indoors so it doesn't really consider the outdoor environment that we would be working in.

2.4 "Comparative Analysis-TCP-UDP." *Laynetworks*. N.p., n.d. Web. 14 Feb. 2013. <http://www.laynetworks.com/Comparative%20analysis_TCP%20Vs%20UDP.htm>.

The User Datagram Protocol (UDP) is a connectionless networking protocol that sends data at any moment without prior notice. UDP is used in large part for error checking in network connections. It is also used for voice over IP (VoIP) because its packet-based and connectionless rules inherit all relevant IP properties, which is acceptable for real-time services such as VoIP. However, UDP is also colloquially called Unreliable DP for a number of reasons. One point is that there is no guarantee that a packet will be delivered and no way of knowing so unless the receiving end communicates in some response. Also, implementation is up to programs that use it and thus there is no flow control. These factors really support that we could only really use UDP for transmission of unimportant data and that we must do TCP to ensure that found packets are received.

2.5 "Satellite coverage in urban areas using Unmanned Airborne Vehicles (UAVs)" *Computer Science Department, University of California Los Angeles* Web. 14 Feb. 2013 <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.74.633&rep=rep1&type=pdf>

This is an implementation of TCP data transfer between a UAV and stationary unit. While this implementation is much more advanced and deals with a different situation than our mission it does provide information about interference and connection methods used between various subsystem parts similar to what will occur during our mission. It also provides methods of calculating data transfer rates while experiencing interference.

3 Requirements:

3.1 Range: System must function within about a mile and a half line of sight between transmitter and receiver for safety. Due to the fact that the human eye can only distinguish objects within three miles, it was determined that this was the maximum range that our UAV would be away from our pilot.

3.2 Frequency: Within a legal band. According to FAA regulations.

3.3 Modular: System can easily allow integration of subsystems. To allow for current and future use in RMAX projects.

3.4 GPS:

1. Receive coordinates indicating UAV location
2. Receive coordinates indicating target location when found

3.5 Data: Package and send secure data between ground stations in according to Stanag 4586 protocol, as requested by Cal Poly Pomona

4 Design and Justification:

4.1 Hardware:

4.1.1 Structure:

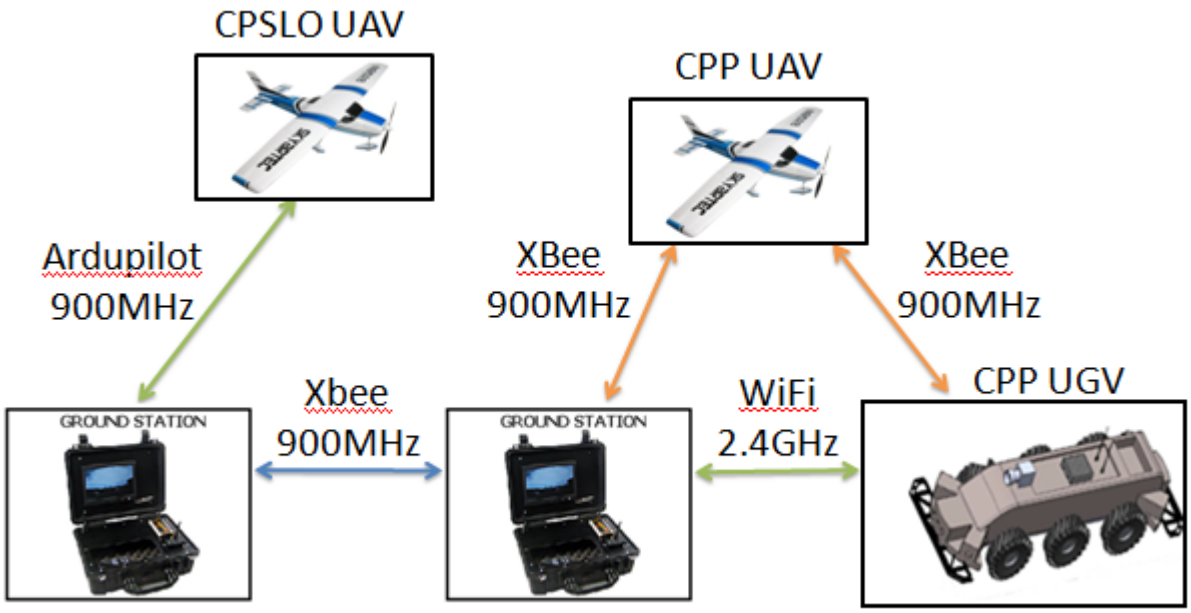

Figure 1: This figure shows the basic hardware structure used for wireless communications between subsystems

4.1.2 Alfa AWUS 036H:

A pair of these directional wireless antennas were used to communicate between the Cal Poly SLO UAV and Ground Station. By creating an Ad-hoc network and using the TCP Server-Client communication architecture we transmitted GPS coordinates, and found packets to the rest of the system. Technical specifications of these antennas are included later in this document.

4.1.3 Digi XBee Pro XSC (S3B)

This newest XBee, allows for legal transmission of RF data at extremely high rates with calculated accuracy. It supports everything we would need for the mission except for an optional video stream because of its limited bandwidth. However, for sending strictly GPS coordinates of the UAV and of the target, XBee was the simplest, cheapest, and most reliable form of close range wireless transmission. Specs of the current model and of its legacy model are shown later in the document.

Specifications	XBee-PRO® XSC (S3)	XBee-PRO® XSC (S3B)
Hardware		
Processor	ADF7025 transceiver, Atmel AT91SAM7S	ADF7023 transceiver, Cortex-M3 EFM32G230 @ 28 MHz
Frequency Band	902 MHz to 928 MHz	
Antenna Options	Wire, U.FL, RPSMA	
Performance		
RF Data Rate	10 Kbps	10 Kbps or 20 Kbps
Indoor/Urban Range	Up to 1200 ft (370 m)	Up to 2000 ft (610 m)
Outdoor/Line-Of-Sight Range	Up to 6 mi (9.6 km)	Up to 9 mi (14 km) w/ dipole antenna Up to 28 mi (45 km) w/ high-gain antenna
Transmit Power	Up to 20 dBm (100 mW)	Up to 24 dBm (250 mW) software selectable
Receiver Sensitivity	-106 dBm	-109 dBm at 9600 baud -107 dBm at 19200 baud
Features		
Spread Spectrum	FHSS	
Operating Temperature	-40° C to +85° C	
Power		
Supply Voltage	3.0 - 3.6 VDC	2.4 to 3.6 VDC
Transmit Current	265 mA	215 mA
Receive Current	65 mA	26 mA
Sleep Current	45 uA	2.5 uA
Regulatory Approvals		
FCC	MCQ-XBEEXSC	MCQ-XBPS3B
IC	1846A-XBEEXSC	1846A-XBPS3B
C-Tick	No	Australia
Pricing		
Development Kit (includes 2 modules)	$149	

Table 1: XBee specifications courtesy of http://www.digi.com/products/wireless-wired-embedded-solutions/zigbee-rf-modules/point-multipoint-rfmodules/xbee-pro-xsc#specs

4.2 Frequency: The frequency ranges in the legal ISM (Industrial Scientific Medical) band are listed below, both our wifi router and XBee conform to the legal limits.

Frequency range		Bandwidth	Center frequency	Availability
6.765 MHz	6.795 MHz	30 KHz	6.780 MHz	Subject to local acceptance
13.553 MHz	13.567 MHz	14 KHz	13.560 MHz	
26.957 MHz	27.283 MHz	326 KHz	27.120 MHz	
40.660 MHz	40.700 MHz	40 KHz	40.680 MHz	
433.050 MHz	434.790 MHz	1.84 MHz	433.920 MHz	Region 1 only and subject to local acceptance
902.000 MHz	928.000 MHz	26 MHz	915.000 MHz	Region 2 only
2.400 GHz	2.500 GHz	100 MHz	2.450 GHz	
5.725 GHz	5.875 GHz	150 MHz	5.800 GHz	
24.000 GHz	24.250 GHz	250 MHz	24.125 GHz	
61.000 GHz	61.500 GHz	500 MHz	61.250 GHz	Subject to local acceptance
122.000 GHz	123.000 GHz	1 GHz	122.500 GHz	Subject to local acceptance
244.000 GHz	246.000 GHz	2 GHz	245.000 GHz	Subject to local acceptance

Table 2: ISM frequency band courtesy of en.wikipedia.org/wiki/ISM_band

4.3 Antenna:

4.3.1 Overview:

Once the TX and RX hardware is established, the correct antenna must be used for stable communication. In order to create an air-link, the radiation patterns of the antennas must encapsulate each other. To figure out the best way of accomplishing this we considered 2 different types of antennas Omni-directional and Directional

4.3.2 Onboard UAV:

4.3.2.1 Considerations:

Being onboard, the transmitter must be able to make a wireless link no matter what state or condition the RMAX finds itself in; this includes traveling upside down, completely sideways, or even completely vertical. We must consider all orientations,

scenarios and possible circumstances that could affect the RMAX before deciding what antenna is going to work best.

4.3.2.2 Comparison

Powered Omni-directional

Although omni-directional antennas provide for the largest range of signal coverage utilizing a spherical radiation pattern, they provide the worst data rates as data is sent away in all directions and only the small portion heading straight towards the receiver is gathered. This will most likely work but make a slow connection.

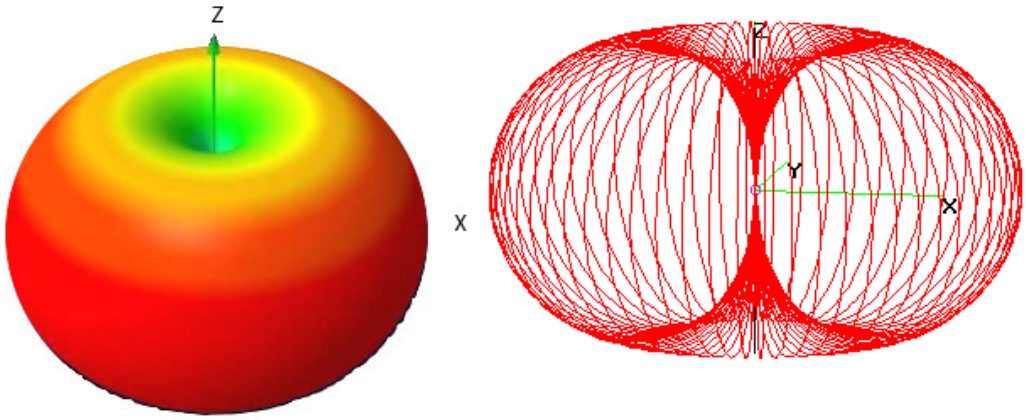

Figure 2: Omni directional Antenna Radiation Pattern courtesy of som.csudh.edu

Directional

The upside of using a directional antenna is that it focuses all its radiation towards a particular direction. What this means is a sort of best case scenario for air-link. What this also means for a moving wireless link is that the direction of optimal antenna placement is not going to be stationary; as the RMAX moves, so must the directional antenna follow the moving perspective of the ground station. To accomplish something of this nature, a Gimbal tracking system set to follow the ground station could be used onboard to move the antenna according to flight patterns.

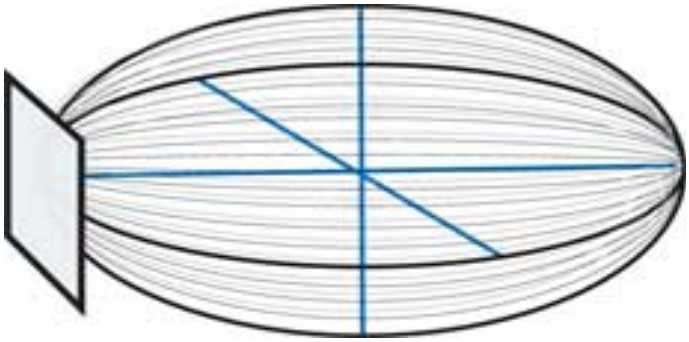

Figure 3: Flat panel antenna radiation pattern courtesy of Digital-Transmissions.co.uk

4.4 Software:

4.4.1 Overview:

After deciding on hardware, a decision had to be made about the software method that would be used to transfer data. As well as how we could make the software modular

4.4.2 UDP:

Acronym for: User Datagram Protocol or Universal Datagram Protocol

Connection: UDP is a connectionless protocol.

Function: UDP is also a protocol used in message transport or transfer. This is not connection based which means that one program can send a load of packets to another and that would be the end of the relationship.

Usage: UDP is used for games or applications that require fast transmission of data. UDP's stateless nature is also useful for servers that answer small queries from huge numbers of clients.

Examples: DNS, DHCP, TFTP, SNMP, RIP, VOIP etc...

Ordering of data packets: UDP has no inherent order as all packets are independent of each other. If ordering is required, it has to be managed by the application layer.

Speed of transfer: UDP is faster because there is no error-checking for packets.

Reliability: There is no guarantee that the messages or packets sent would reach at all. Header Size: UDP Header size is 8 bytes.

Common Header Fields: Source port, Destination port, Check Sum

Streaming of data: Packets are sent individually and are checked for integrity only if they arrive. Packets have definite boundaries which are honored upon receipt, meaning a read operation at the receiver socket will yield an entire message as it was originally sent.

Weight: UDP is lightweight. There is no ordering of messages, no tracking connections, etc. It is a small transport layer designed on top of IP.

Data Flow Control: UDP does not have an option for flow control

Error Checking: UDP does error checking, but no recovery options.

Fields: 1. Length, 2. Source port, 3. Destination port, 4. Check Sum

Final Thoughts: While lightweight UDP does not offer the reliability needed to ensure that we will have accurate data and constant signal during flight. We will consider adding a UDP stream to transfer pictures, and videos of flight due to that they are not mission critical and can function with missing packets.

4.4.3 TCP:

Acronym for: Transmission Control Protocol

Connection: TCP is a connection-oriented protocol.

Function: As a message makes its way across the internet from one computer to another. This is connection based.

Usage: TCP is used in case of non-time critical applications.

Examples: HTTP, HTTPs, FTP, SMTP Telnet etc...

Ordering of data packets: TCP rearranges data packets in the order specified.

Speed of transfer: The speed for TCP is slower than UDP.

Reliability: There is absolute guarantee that the data transferred remains intact and arrives in the same order in which it was sent.

Header Size: TCP header size is 20 bytes

Common Header Fields: Source port, Destination port, Check Sum

Streaming of data: Data is read as a byte stream, no distinguishing indications are transmitted to signal message (segment) boundaries.

Weight: TCP is heavy-weight. TCP requires three packets to set up a socket connection, before any user data can be sent. TCP handles reliability and congestion control.

Data Flow Control: TCP does Flow Control. TCP requires three packets to set up a socket connection, before any user data can be sent. TCP handles reliability and congestion control.

Error Checking: TCP does error checking

Fields: 1. Sequence Number, 2. AcK number, 3. Data offset, 4. Reserved, 5. Control bit, 6. Window, 7. Urgent Pointer 8. Options, 9. Padding, 10. Check Sum, 11. Source port, 12. Destination port

Final Conclusion: It was decided that due to the guaranteed reception of packets that TCP was the method to be used. It would guarantee that we would eventually receive the found target packet as long as a connection existed.

4.4.4 Modular Design:

4.4.4.1 Overview:

Once decided upon structure of communication we needed In order to make the design modular we broke the code into four separate parts with the concept that once the main connection was established between teams we could distribute two sections of code to every group one for on board the plane to capture data and one for on ground to receive the data and use as appropriate for each subsystem. This would allow for independent testing and integration for each subsystem and upon success we could easily integrate each group into the entire network by simply adding a few customized send and receive functions. This modular design can also

serve as a failure mechanism if one group malfunctioned the system could still function.

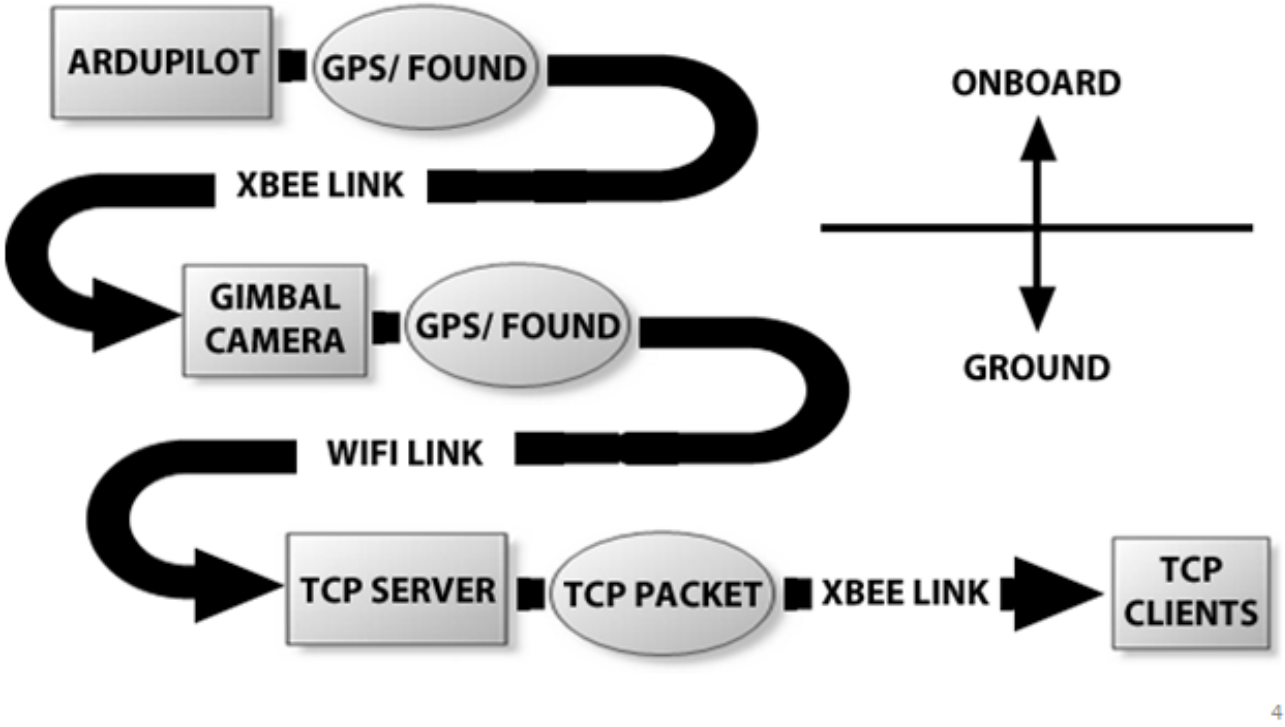

Figure 4: of communications and packet structure used in Final Design Demo

4.4.5 Code – See appendices for code reference

4.4.5.1 server.cpp

This code would be run aboard the RMAX or UAC. It takes recorded data from client.cpp by use of threads and packages it into the correct message header formant and sends it to the tcp_server by use of pipes for transmission to the ground station. This code can be edited by subsystems to receive data.

4.4.5.2 client.cpp

This is the code that should be modified for collection of data aboard the RMAX or UAV. Modify the while loop in main to read data and send an appropriate IPC_Message template. It communicates with server.cpp by use of pipes

4.4.5.3 tcp_server.cpp

It takes no startup arguments and will print out the name of the server which will be used for tcp_client to connect to during startup. It should never be modified as it distributes data to the ground station. It reads packaged data from server.cpp by checking a pipe and outputting it to connected clients by use of sockets.

4.4.5.4 tcp_client.cpp

This is the client that receives data from the air the code should be modified in a specific function to distribute the data to various parts of the mission that are on the ground. It connects through tcp sockets by port and host name and will receive data and transfer it to various subsystems depending on system implementation.

4.4.5.5 messageTypes.h

The universal header file that defines the packet structure sent throughout the network. Modify IPC_Message_Template as needed for added datatypes and use.

4.5 Failure Matrix and Risk Assessment

4.5.1 Possible Errors During System Operation

4.5.1.1 Periods of time with lower data transfer rates

Possible Damage:

1. Bad resolution
2. Slow GPS coordinate refresh rate

How to avoid:

1. Keep directional antennas in correct orientations.
2. Keep UAV within required range
2. Compress packets to allow optimal data transfer

System Response:

System has a running counter of packet ids and will request packet id until it is received meaning that no packet will be lost when data rate is lower.

4.5.1.2 Full but temporary loss of communication between UAV and Ground Station

Possible Damage:

Loss of updating data stream until reconnection

How to avoid:

1. Keep directional antennas in correct orientations
2. Keep UAV within required range

System Response:

1. Attempt to re-establish connection until successful
2. Upon success receive missed packets while receiving current data

4.5.2 Hardware Failure Run

4.5.2.1 Possible Damage:

Failure of current system run

4.5.2.2 How to Avoid:

1. Thorough check of hardware before use
2. Constant maintenance of hardware
3. Proper use of hardware

4.5.2.3 System Response:

Repair or replace hardware with backup unit and retry system run

Frequency Of Occurrence / Damage Caused	Seldom →		Regularly
Minor ↓	Slow Connection	Missed Packets	
Catastrophic	Hardware Failure		

Table 3: Risk Assessment Matrix for Wireless Communications

5 System Integration and Testing:

5.1 Test Plan:

5.1.1 Overview:

Due to the modular nature and multiple parts we were able to do multiple separate tests throughout our process and build upon major events without dependencies. We only had dependencies toward the end of the project where integration and communication with subsystems needed to be tested

5.1.2 Tests:

5.1.2.1 Hardware Range and Accuracy Tests:

Tested by finding limits and accuracy of our wifi antennas and RF transmitters. Done by transmitting data from included tools at various ranges, heights and levels of interference.

Elevation Difference (Feet)	Distance (feet)	Sent	Received	% Loss	Size
0	1430	50	50	0	65000
335	2072	50	50	0	4000
335	2072	50	42	16	40000
507	1678	50	50	0	40000
507	1678	50	49	2	65000
507	1678	50	50	0	6500
10	1460	50	28	44	6500
10	1460	50	12	76	65000
0	1683	50	32	36	4000
343	2100	50	16	68	4000

Table 4: Elevation and Distance Packet Test Results

distance 5dB (mi)	strength (%)	quality (%)	distance 9dB (mi)	strength (%)	quality (%)
0	100	86	0.5	86	85
0.2	82	76	0.8	82	84
0.5	80	80	1	82	76
1	76	71	1.2	74	69
1.5	70	40	1.5	72	65

Table 5: Stream Quality Test Results

5.1.2.2 Test of TCP transfer between software.

Test transfer of packets in a lab situation between networked machines to ensure software functionality.

5.1.2.3 Integration of Wireless antennas

Create a wireless network using long range wireless antennas and test in lab setting packet transfer between networked machines.

5.1.2.4 Field test of System

Repeat range and accuracy tests with current system to ensure they meet standards

5.1.2.5 Integration of collection receive software provided by Daniel Park

Acquire code from Daniel and integrate with existing code

Repeat related above steps with the collection and reception of data.

5.1.2.5.1 Distribute code integrate subsystems.

Distribute code to other groups and work on individual testing of subsystem with system

5.1.2.5.2 Full Scale Test

Simulate entire rmax system that will be demonstrated on demo day

5.1.3 Gantt Chart:

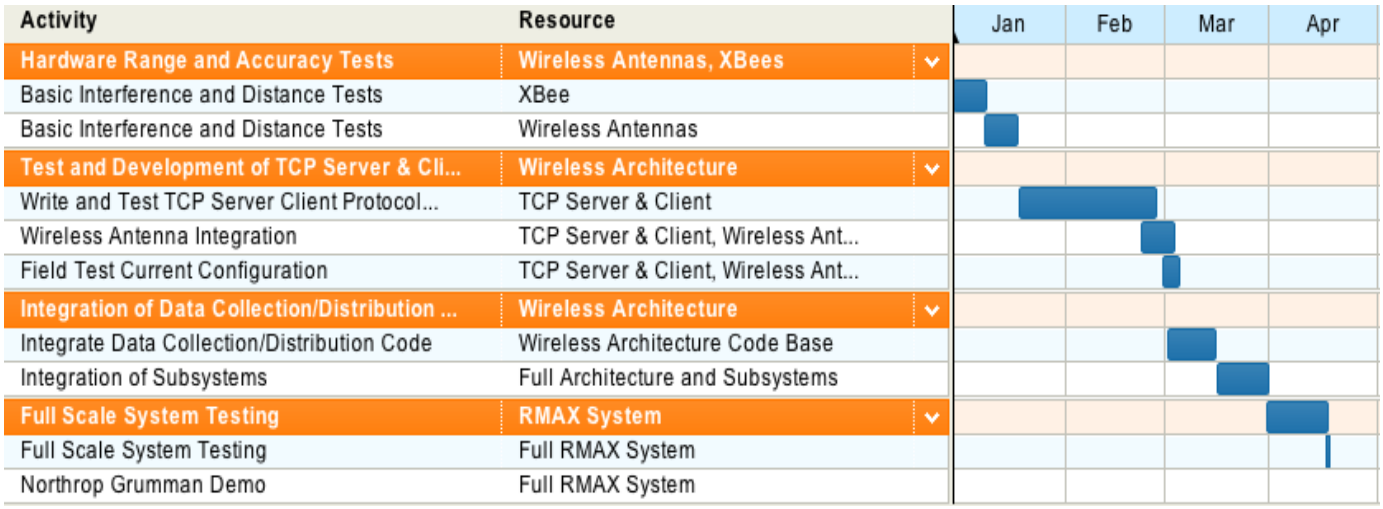

Figure 5: Testing and Development Gantt Chart

6 Conclusion

The Project was a success we were able to do several demonstrations of our fully functioning system during our demonstration for our client Northrop Grumman that met there requests. Our client reported that they were impressed with the results, and has decided to sponsor another year of the RMAX project for the 2013 – 2014 school year.

The goals of the Wireless Communications team were met. We delivered and implemented modular system architecture that can be configured for future versions of the RMAX project to integrate any new subsystem requiring communication within a network. Without the success of our team the success of the project would not have been possible. Following our demonstration a user manual was written for the wireless architecture and transmission hardware that will be passed to the next Wireless-Communications group so that they will be able to quickly continue our work with minimal effort.

While our system was delivered successful there is room for improvement. A UDP stream can be added for the transfer of pictures and video or other non-vital information. There is already groundwork present to easily add this feature if it is ever needed. The wireless antennas and XBee RF transmitters can be upgraded to give faster data transfer rates and better connection quality. This would lower the risk of failed packets during a mission and allow for a quicker recovery in the event of a temporary transmission failure.

7 Works Cited

7.1 "Alfa AWUS036H 1000mW USB Wireless WiFi Adapter Antenna. Long-Range." *Data-alliance.net*. N.p., n.d. Web. 01 Dec. 2013.

7.2 "Beej's Guide to Network Programming." *Beej's Guide to Network Programming*. N.p., n.d. Web. 01 Dec. 2013.

7.3 "CiteSeerX." *CiteSeerX*. N.p., n.d. Web. 01 Dec. 2013.

7.4 "EUROPEAN ANTENNAS Directional Flat Panel Andlow Profile Antennas Technical Information 43221709." *EUROPEAN ANTENNAS Directional Flat Panel Andlow Profile Antennas Technical Information 43221709*. N.p., n.d. Web. 01 Dec. 2013.

7.5 "Five Lessons for a Successful STEM Career." *Lockheed Martin · STANAG 4586*. N.p., n.d. Web. 01 Dec. 2013.

7.6 "GPS Accuracy and Limitations." *GPS Accuracy and Limitations*. N.p., n.d. Web. 01 Dec. 2013.

7.7 "GPS Accuracy." *GPS.gov:*. N.p., n.d. Web. 01 Dec. 2013.

7.8 "Information." - *C*. N.p., n.d. Web. 01 Dec. 2013.

7.9 "ISM Band." *Wikipedia*. Wikimedia Foundation, 17 Nov. 2013. Web. 01 Dec. 2013.

7.10 Kurose, James F., and Keith W. Ross. *Computer Networking: A Top-down Approach*. Boston: Pearson, 2013. Print.

7.11 "Making an IR Object Tracking System." *Let's Make Robots!* N.p., n.d. Web. 01 Dec. 2013.

7.12 "Omni-directional Antenna." *Omni-directional Antenna*. N.p., n.d. Web. 01 Dec. 2013.

7.13 " RF, Wireless RF Receivers, Transceivers, Transmitters Transceivers XBee-PRO® XSC RF Modules." *XBee-PRO® XSC RF Modules*. N.p., n.d. Web. 01 Dec. 2013.

7.14 "TCP - UDP Comparative Analysis - Data Communications and Networks - Free Computer Science Tutorials - Provided by Laynetworks.com." *TCP - UDP Comparative Analysis - Data Communications and Networks - Free Computer Science Tutorials - Provided by Laynetworks.com*. N.p., n.d. Web. 01 Dec. 2013.

7.15 "XBee-PRO® XSC." - *Digi International*. N.p., n.d. Web. 01 Dec. 2013.

8 Appendices

8.1 Materials Used

Visual Studio 2010 C++ edition

2 Alfa AWUS 036H wireless antennas

2 Digi XBee Pro XSC (S3B) RF transmitters

Alfa Networking software

Microsoft Windows XP

Microsoft Windows 7

8.2 Charts and diagrams:

8.2.1 Figures

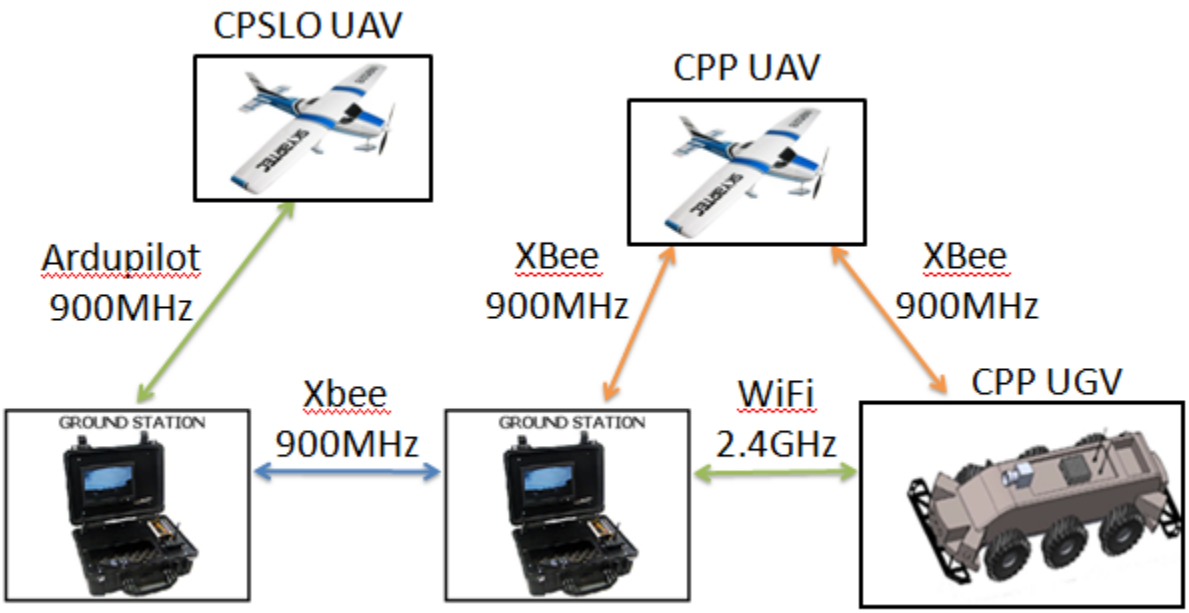

Figure 1: This figure shows the basic hardware structure used for wireless communications between subsystems

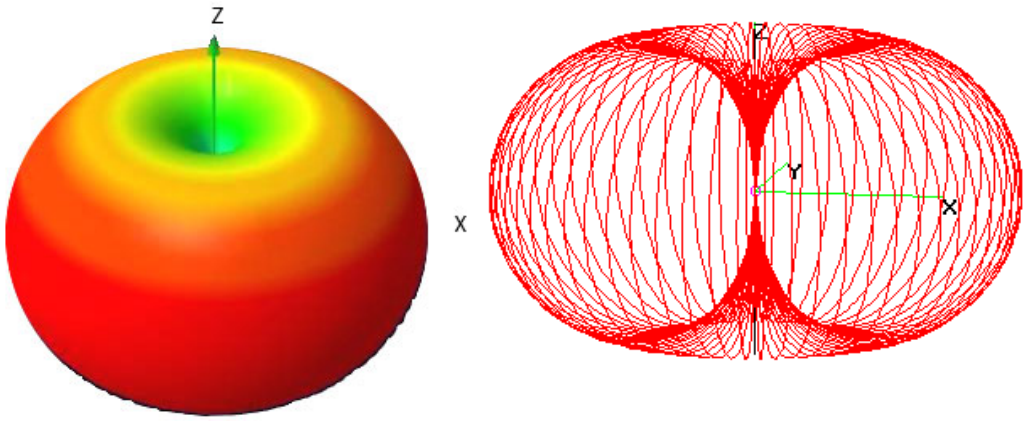

Figure 2: Omni directional Antenna Radiation Pattern courtesy of som.csudh.edu

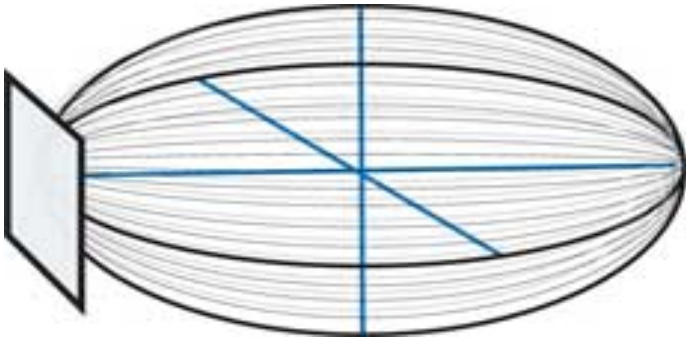

Figure 3: Flat panel antenna radiation pattern courtesy of Digital-Transmissions.co.uk

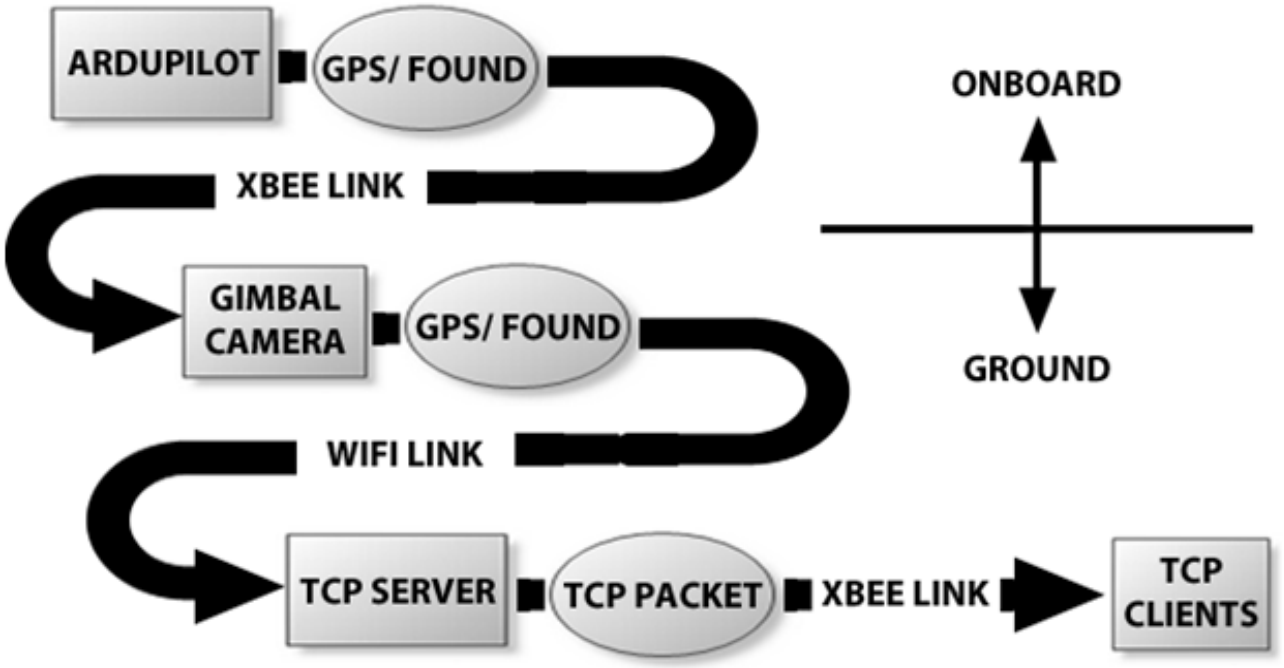

Figure 4: of communications and packet structure used in Final Design Demo

8.2.2 Tables

Specifications	XBee-PRO® XSC (S3)	XBee-PRO® XSC (S3B)
Hardware		
Processor	ADF7025 transceiver, Atmel AT91SAM7S	ADF7023 transceiver, Cortex-M3 EFM32G230 @ 28 MHz
Frequency Band	902 MHz to 928 MHz	
Antenna Options	Wire, U.FL, RPSMA	
Performance		
RF Data Rate	10 Kbps	10 Kbps or 20 Kbps
Indoor/Urban Range	Up to 1200 ft (370 m)	Up to 2000 ft (610 m)
Outdoor/Line-Of-Sight Range	Up to 6 mi (9.6 km)	Up to 9 mi (14 km) w/ dipole antenna Up to 28 mi (45 km) w/ high-gain antenna
Transmit Power	Up to 20 dBm (100 mW)	Up to 24 dBm (250 mW) software selectable
Receiver Sensitivity	-106 dBm	-109 dBm at 9600 baud -107 dBm at 19200 baud
Features		
Spread Spectrum	FHSS	
Operating Temperature	-40° C to +85° C	
Power		
Supply Voltage	3.0 - 3.6 VDC	2.4 to 3.6 VDC
Transmit Current	265 mA	215 mA
Receive Current	65 mA	26 mA
Sleep Current	45 uA	2.5 uA
Regulatory Approvals		
FCC	MCQ-XBEEXSC	MCQ-XBPS3B
IC	1846A-XBEEXSC	1846A-XBPS3B
C-Tick	No	Australia
Pricing		
Development Kit (includes 2 modules)	$149	

Table 1: XBee specifications courtesy of http://www.digi.com/products/wireless-wired-embedded-solutions/zigbee-rf-modules/point-multipoint-rfmodules/xbee-pro-xsc#specs

Frequency range	Bandwidth	Center frequency	Availability
6.765 MHz	6.795 MHz	30 KHz	6.780 MHz	Subject to local acceptance
13.553 MHz	13.567 MHz	14 KHz	13.560 MHz	
26.957 MHz	27.283 MHz	326 KHz	27.120 MHz	
40.660 MHz	40.700 MHz	40 KHz	40.680 MHz	
433.050 MHz	434.790 MHz	1.84 MHz	433.920 MHz	Region 1 only and subject to local acceptance
902.000 MHz	928.000 MHz	26 MHz	915.000 MHz	Region 2 only
2.400 GHz	2.500 GHz	100 MHz	2.450 GHz	
5.725 GHz	5.875 GHz	150 MHz	5.800 GHz	
24.000 GHz	24.250 GHz	250 MHz	24.125 GHz	
61.000 GHz	61.500 GHz	500 MHz	61.250 GHz	Subject to local acceptance
122.000 GHz	123.000 GHz	1 GHz	122.500 GHz	Subject to local acceptance
244.000 GHz	246.000 GHz	2 GHz	245.000 GHz	Subject to local acceptance

Table 2: ISM frequency band courtesy of en.wikipedia.org/wiki/ISM_band

Frequency Of Occurrence / Damage Caused	Seldom →		Regularly
Minor	Slow Connection	Missed Packets	
↓			
Catastrophic	Hardware Failure		

Table 3: Risk Assessment Matrix for Wireless Communications

Elevation Difference (Feet)	Distance (feet)	Sent	Received	% Loss	Size
0	1430	50	50	0	65000
335	2072	50	50	0	4000
335	2072	50	42	16	40000
507	1678	50	50	0	40000
507	1678	50	49	2	65000
507	1678	50	50	0	6500
10	1460	50	28	44	6500
10	1460	50	12	76	65000
0	1683	50	32	36	4000
343	2100	50	16	68	4000

Table 4: Elevation and Distance Packet Test Results

distance 5dB (mi)	strength (%)	quality (%)	distance 9dB (mi)	strength (%)	quality (%)
0	100	86	0.5	86	85
0.2	82	76	0.8	82	84
0.5	80	80	1	82	76
1	76	71	1.2	74	69
1.5	70	40	1.5	72	65

Table 5: Stream Quality Test Results

8.3 Code

server.cpp

Description: This is the first program that should be run. It should be run onboard the RMAX or UAV. It takes recorded data from client.cpp and sends it to the tcp_server for transmission to the ground station. See the commented code for information on editing the code for addition of new data. No command args are needed for the running of this code it simply waits for the start of a client to collect data from.

Code:

```
/* Main RMAX Server
*
* Rewritten to be more "C" friendly
*
* @author: Daniel Park <dpark03@calpoly.edu>
*
*
* Notes:
* dpark - 1. This MUST be the FIRST to START
*         2. This will kick off the TCP & UDP servers?
*/

#include <Windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <vector>
#include "messageTypes.h"
```

```
#define SEVER_MAX_RETRY 10
#define BUFFER_SIZE 1500
#define DEFAULT_PIPE_NAME "\\\\.\\pipe\\RMAXPipe"
#define GOOD_REPLY "OK"

#define MILISEC_TO_WAIT 1

HANDLE updateMutex;
HANDLE TCP_ServerPipe = INVALID_HANDLE_VALUE;
HANDLE spamHandle;
DWORD spamID;
IPC_Message message;

/*
 * We shouldn't need to write anything to the client
 * But here is how you would do it for testing purposes
 */


void print_IPC_Message_Template(IPC_Message_Template out)
{
   switch(out.messageType)
   {
      case GPS:
         printf("Message Type: GPS\n");
         break;
      case VIDEO:
         printf("Message Type: VIDEO\n");
         break;
      case ENGINE_DATA:
         printf("Message Type: ENGINE_DATA\n");
         break;
      case TEMP:
         printf("Message Type: TEMP\n");
         break;
      default:
         printf("Message Type: UNKNOWN\n");
         break;
   }
   printf("Message Length: %d\n", out.messageLen);
   printf("Message:\n");
   for(int i = 0; i < out.messageLen; i++)
   {
      putchar(out.message[i]);
   }
   putchar('\n');
}
void print_IPC_Message(IPC_Message *msg)
{
```

```
    print_IPC_Message_Template(msg -> gps);
    print_IPC_Message_Template(msg -> temp);
}
int sendToClient(HANDLE hPipe, IPC_Message_Template * msg)
{
    BOOL success;
    DWORD bytesWritten;
    success = WriteFile(
        hPipe,
        msg,
        sizeof(IPC_Message_Template),
        &bytesWritten,
        NULL);

    if(!success){
        fprintf(stderr, "InstanceThread[%d]: WriteFile Failed: %d\n", GetCurrentThreadId(),
GetLastError());
        return EXIT_FAILURE;
    }
    fprintf(stdout, "InstanceThread[%d]: Sent Client Response: %s\n", GetCurrentThreadId(), msg-
>message);
    fflush(stdout);
    FlushFileBuffers(hPipe);

    return EXIT_SUCCESS;
}

/*
 * Send to Server
 */
int sendToServer(HANDLE hPipe, IPC_Message * msg)
{
    BOOL success;
    DWORD bytesWritten;
    success = WriteFile(
        hPipe,
        msg,
        sizeof(IPC_Message),
        &bytesWritten,
        NULL);

    if(!success){
        fprintf(stderr, "InstanceThread[%d]: WriteToServer Failed: %d\n", GetCurrentThreadId(),
GetLastError());
        fprintf(stderr, "Server died...\n");
        return EXIT_FAILURE;
    }
    fprintf(stdout, "InstanceThread[%d]: Sent msg to TCP Server\n", GetCurrentThreadId());
    fflush(stdout);
    FlushFileBuffers(hPipe);
```

```
    return EXIT_SUCCESS;
}


/*
 * Thread to spam out to TCP Server to send to base station
 */
DWORD WINAPI SpamThread(LPVOID){
    DWORD waitResult;
    int count = 0;

    //Keep taking the mutex and spam the BaseStation.
    while(1){
        waitResult = WaitForSingleObject(updateMutex, INFINITE);
        switch(waitResult){
            case WAIT_ABANDONED:
                //Okay... who crashed?
                fprintf(stderr, "Spam Mutex: Someone crashed.... \n");
                break;

            case WAIT_OBJECT_0:
                //SPAM BASETATION HERE!!!

                if(TCP_ServerPipe != INVALID_HANDLE_VALUE){
                        //printf("sending\n");
                        //print_IPC_Message(&message);
                    if(sendToServer(TCP_ServerPipe, &message) == EXIT_FAILURE){
                        TCP_ServerPipe = INVALID_HANDLE_VALUE;
                        fprintf(stderr, "TCP SERVER died...\nQuiting Thread....\n");
                        return EXIT_FAILURE;
                    }
                } else {
                    fprintf(stderr, "TCP SERVER IS NOT UP!!!!\nQuiting Thread....\n");
                    return EXIT_FAILURE;
                }

                /*
                if (count == 100000) {
                    fprintf(stdout, "SPAM!: %s\n", message.temp.message);
                    count = 0;
                }
                count++;
                break;
                */

            case WAIT_TIMEOUT:
                //This shound't happen
                break;

            case WAIT_FAILED:
```

```
            //This shound't happen
            return FALSE;
            break;
        }
        if (!ReleaseMutex(updateMutex)){
            fprintf(stderr, "MUTEX RELEASE Error!!!: %d\n", GetLastError());
            return FALSE;
        }
    }
    return true;
}



/*
 * Thread to handle each message coming in and updating the "Master"
 *  message data
 */
DWORD WINAPI ClientThread(LPVOID lpvParam)
{
    HANDLE hPipe;
    BOOL success;
    DWORD bytesRead;
    DWORD waitResult;
    IPC_Message_Template msg;

    if(lpvParam == NULL){
      fprintf(stderr, "ERROR - Pipe Server Failure:\n");
      fprintf(stderr, "   InstanceThread got an unexpected NULL value in lpvParam.\n");
      fprintf(stderr, "   InstanceThread exitting.\n");
      return EXIT_FAILURE;
    }

    hPipe = (HANDLE) lpvParam;

    while(true){
       memset(&msg, 0, sizeof(msg));
       success = ReadFile(
          hPipe,
          &msg,
          sizeof(IPC_Message_Template),
          &bytesRead,
          NULL);

       if(msg.messageType == TCP_SERVER){
          fprintf(stdout,"I HAS TCP SERVER!!!\n");
          //Password for TCP_Server Handle to register with IPC_Server
          if(((std::string)(msg.message)).compare("RMAX PA$$CODE") == 0){
             TCP_ServerPipe = hPipe;
             fprintf(stdout,"TCP SERVER HAS CORRECT PASSWORD!!!!!\n");
```

```
        } else {
            fprintf(stdout,"TCP_SERVER sent wrong password: %s\n", msg.message);
        }
        spamHandle = CreateThread(
                    NULL,
                    0,
                    SpamThread,
                    (LPVOID) hPipe,
                    0,
                    &spamID);
        if (spamHandle == NULL){
            fprintf(stderr, "ActionThread Failed: %d\n", GetLastError());
        }
        fprintf(stdout, "Created TCP_Server Thread!\n");
        return EXIT_SUCCESS;
    }

    if(!success || bytesRead == 0) {
        if(GetLastError() == ERROR_BROKEN_PIPE) {
            fprintf(stderr, "InstanceThread[%d]: Client Disconnected.\n", GetCurrentThreadId(),
GetLastError());
        }
        else
            fprintf(stderr, "InstanceThread[%d]: ReadFile Failed: %d\n", GetCurrentThreadId(),
GetLastError());
        break;
    }

    waitResult = WaitForSingleObject(updateMutex, INFINITE);
    switch(waitResult){
        case WAIT_ABANDONED:
            //Okay... who crashed?
            fprintf(stderr, "Update Mutex: Someone crashed....\n");
            break;

        case WAIT_OBJECT_0:
            //UPDATE HERE!!!
            //ADD IN THE UPDATE FOR EACH DIFFERENT TYPE OF DATA
            switch (msg.messageType){
                case TCP_SERVER:
                    break;
                case GPS:
                    memset(message.gps.message, '\0', BUFFER_SIZE);
                    memcpy(message.gps.message, msg.message, msg.messageLen);
                    message.gps.messageLen = msg.messageLen;
                    break;
                case TEMP:
                    memset(message.temp.message, '\0', BUFFER_SIZE);
                    memcpy(message.temp.message, msg.message, msg.messageLen);
                    message.temp.messageLen = msg.messageLen;
```

```
                break;
            default:
                fprintf(stderr, "Got strange message!!!: %s\n", msg.message);
                break;
        }
        //printf("updated\n");
        //print_IPC_Message(&message);
        break;

    case WAIT_TIMEOUT:
        //This shound't happen
        fprintf(stderr, "Update Mutex: Wait Timeout?!?!....\n");
        break;

    case WAIT_FAILED:
        //This shound't happen
        fprintf(stderr, "Update Mutex: Wait Failed?!?!....\n");
        break;
    }
    if (!ReleaseMutex(updateMutex)){
        fprintf(stderr, "MUTEX RELEASE Error!!!: %d\n", GetLastError());
        return FALSE;
    }

    fprintf(stdout,"UPDATE DONE\n");
    /*
    memcpy(msg.message, GOOD_REPLY, sizeof(GOOD_REPLY));
    msg.messageLen = sizeof(GOOD_REPLY)+1;
    sendToClient(hPipe, &msg); //OPTIONAL!
    */

  } //Will need some type of flag management

  FlushFileBuffers(hPipe);//Force flush data into File
  DisconnectNamedPipe(hPipe);
  CloseHandle(hPipe);

  return EXIT_SUCCESS;
}

int main(int argc, TCHAR *argv[])
{
  int err = 0;
  static HANDLE nPipe, cThread;
  DWORD connected, threadID;
  HANDLE actionHandle = INVALID_HANDLE_VALUE;

  std::vector<HANDLE> vThreadHandles;

  //Init global struct
```

```
    message.gps.messageType = GPS;
    memcpy(message.gps.message, "TEMP MESSAGE FOR FURTURE EXPANSION",
strlen("TEMP MESSAGE FOR FURTURE EXPANSION")+1);
    message.gps.messageLen = strlen("TEMP MESSAGE FOR FURTURE EXPANSION")+1;

    message.temp.messageType = TEMP;
    memcpy(message.temp.message, "TEMP MESSAGE FOR FURTURE EXPANSION",
strlen("TEMP MESSAGE FOR FURTURE EXPANSION")+1);
    message.temp.messageLen = strlen("TEMP MESSAGE FOR FURTURE EXPANSION")+1;

    //Create updating mutex
    updateMutex = CreateMutex(NULL, //Defualt Permissions
                        FALSE, //Not owned
                        TEXT("updateMutex")); //Name
    if (updateMutex == NULL){
        fprintf(stderr, "CreateMutex failed: %d\n", GetLastError());
    }


    while(true){
        nPipe = CreateNamedPipe(
                TEXT(DEFAULT_PIPE_NAME),
                PIPE_ACCESS_DUPLEX,
                PIPE_TYPE_MESSAGE |
                PIPE_READMODE_MESSAGE |
                PIPE_WAIT,
                PIPE_UNLIMITED_INSTANCES,
                sizeof(IPC_Message),
                sizeof(IPC_Message),
                NMPWAIT_USE_DEFAULT_WAIT,
                NULL);

        if(nPipe == INVALID_HANDLE_VALUE){
            err++;
            perror("CreateNamedpiped Failed");
            fprintf(stderr, "CreateNamedpiped failed: %d\n", GetLastError());
            if(err < SEVER_MAX_RETRY){
                fprintf(stderr, "Trying again...\n");
                continue;
            } else {
                fprintf(stderr, "Terminating...\n");
                return EXIT_FAILURE;
            }
        }

        fprintf(stdout, "Waiting for Clients....\n");
        connected = ConnectNamedPipe(nPipe, NULL) ? true : GetLastError();

        if(connected){
            fprintf(stdout, "Client connected... Creating thread....\n");
```

```
            cThread = CreateThread(
                NULL,
                0,
                ClientThread,
                (LPVOID) nPipe,
                0,
                &threadID);

            if(cThread == NULL) {
                perror("Create_Child_Thread failed");
                fprintf(stderr, "CreateThread failed: %d\n", GetLastError());
            } else {
                //Store handle to the thread for future use?
                vThreadHandles.push_back(cThread);
            }
            fprintf(stdout, "InstanceThread[%d]: Created\n", threadID);
        } else {
            // Client couldn't connect
            // Close Handle
            CloseHandle(nPipe);
            fprintf(stderr, "Client tried to connect... But couldn't\n");
        }

        //Check THREADS handles!
        for (std::vector<HANDLE>::iterator itr = vThreadHandles.begin(); itr != vThreadHandles.end();
itr++){
                (HANDLE)*itr;
                //naaaaaaaaaa :P
        }
    }
    return EXIT_SUCCESS;
}
```

client.cpp

Description: This is the code that should be modified for collection of data aboard the RMAX or UAV. Modify the while loop in main to read data and send an appropriate IPC_Message template. This code should be ran last. For multiple data types I suggest making multiple functions to create specific messages and write to server.cpp. This code should be ran last.

Code:

```
 / * Main RMAX Client & TEST program
 *
 * Rewritten to be more "C" friendly
 *
 * @author: Daniel Park <dpark03@calpoly.edu>
 *
 */

#include <Windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "messageTypes.h"

HANDLE pipeMutex;

/*
 * This funciton will create the pipe and return a handle to the pipe.
 *  Or else it will return a NULL.
 */
HANDLE nPipeInit(IPC_Message_Type type)
{
    HANDLE nPipe = CreateFile(
        TEXT(DEFAULT_PIPE_NAME),
        GENERIC_READ |
        GENERIC_WRITE,
        0,
        NULL,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL |
        FILE_FLAG_DELETE_ON_CLOSE,
        NULL);

    if(ERROR_PIPE_BUSY == GetLastError()) {
        fprintf(stderr, "Busy Pipe: %d\n", GetLastError());
        return NULL;
```

```
    }

    if(INVALID_HANDLE_VALUE == nPipe){
        fprintf(stderr, "Error connecting to server: %d\n", GetLastError());
        return NULL;
    }

    //Change Read mode to message mode
    DWORD dwMode = PIPE_READMODE_MESSAGE | PIPE_WAIT;
    bool fSuccess = FALSE;
    fSuccess = SetNamedPipeHandleState(
        nPipe,    // pipe handle
        &dwMode,  // new pipe mode
        NULL,     // don't set maximum bytes since it is on the same machine
        NULL);    // don't set maximum time since it is on the same machine
    if (!fSuccess)
    {
        fprintf(stderr,"SetNamedPipeHandleState failed: %d\n", GetLastError());
        return NULL;
    }

    return nPipe;
}

/*
 * This function will send data down a passed in namedPIPE Handle.
 * Returns success or fail.
 */
int sendToPipe(HANDLE nPipe, IPC_Message_Template * msg)
{
    DWORD written = 0;
    bool success;
    DWORD waitResult;

    waitResult = WaitForSingleObject(pipeMutex, INFINITE);
    switch(waitResult){
        case WAIT_ABANDONED:
            //IPC_Server Crashed
            fprintf(stderr,"IPC_SERVER Crashed\n");
            break;
        case WAIT_TIMEOUT:
            //This shound't happen
            fprintf(stderr, "Update Mutex: Wait Timeout?!?!....\n");
            break;
        case WAIT_FAILED:
            //This shound't happen
            fprintf(stderr, "Update Mutex: Wait Failed?!?!....\n");
            break;
        case WAIT_OBJECT_0:
            fprintf(stdout, "Sending %d byte(s) message: %s\n", msg->messageLen, msg->message);
```

```
            fflush(stdout);
            success = WriteFile(
                nPipe,
                msg,
                sizeof(IPC_Message_Template),
                &written,
                NULL);

            if(!success){
                fprintf(stderr, "Write to Pipe Failed %d\n", GetLastError());
                return EXIT_FAILURE;
            }
            break;
    }
    if (!ReleaseMutex(pipeMutex)){
        fprintf(stderr, "MUTEX RELEASE Error!!!: %d\n", GetLastError());
        return FALSE;
    }
    return EXIT_SUCCESS;
}

/*
 * This function reads from a namedPipe.
 * Returns the string or NULL for fail.
 */
IPC_Message_Template * readFromPipe(HANDLE nPipe)
{
    bool success;
    DWORD read = 0;
    IPC_Message_Template * msg = (IPC_Message_Template
*)malloc(sizeof(IPC_Message_Template));

    if(msg == NULL){
        perror("readFromPipe: MALLOC Failed");
        return NULL;
    }

    success = ReadFile(
        nPipe,
        msg,
        sizeof(IPC_Message_Template),
        &read,
        NULL);

    if(!success){
        perror("Reading from NAMED PIPED Errored");
        return NULL;
    }

    return msg;
```

```
}

DWORD WINAPI listenThread(LPVOID lpvParam)
{
    IPC_Message_Template * ret;
    DWORD waitResult;

    waitResult = WaitForSingleObject(pipeMutex, INFINITE);
    switch(waitResult){
        case WAIT_ABANDONED:
            //IPC_Server Crashed
            fprintf(stderr,"IPC_SERVER Crashed\n");
            break;
        case WAIT_TIMEOUT:
            //This shound't happen
            fprintf(stderr, "Update Mutex: Wait Timeout?!?!....\n");
            break;
        case WAIT_FAILED:
            //This shound't happen
            fprintf(stderr, "Update Mutex: Wait Failed?!?!....\n");
            break;
        case WAIT_OBJECT_0:
            //Read from response.. but who cares!
            fprintf(stdout, "\nListening from Handle\n");
            if((ret = readFromPipe((HANDLE)lpvParam)) == NULL){
                fprintf(stderr, "READ_FROM_PIPE Failed!!\n");
                return EXIT_FAILURE;
            }
            fprintf(stdout, "%s\n", ret->message);
            fflush(stdout);
            break;
    }

    if (!ReleaseMutex(pipeMutex)){
        fprintf(stderr, "MUTEX RELEASE Error!!!: %d\n", GetLastError());
        return FALSE;
    }
}


/**
 * A small test main to test the client code
 */
int main(int argc, char *argv[])
{
    HANDLE nPipe;
    HANDLE listenThreadHandle;
    DWORD listenThreadID;

    fprintf(stdout,"This is a test program for Windows IPC\n");
```

```
fprintf(stdout,"Max READ Buffer is: %d\n", MAX_MSG_SIZE);
int count = 0;
int i = 0;
IPC_Message_Template msg;
msg.messageType = TEMP;

//Create Pipe mutex
pipeMutex = CreateMutex(NULL, //Defualt Permissions
                        FALSE, //Not owned
                        TEXT("updateMutex")); //Name
if (pipeMutex == NULL){
   fprintf(stderr, "CreateMutex failed: %d\n", GetLastError());
}

if((nPipe = nPipeInit(TEMP)) == NULL){
   fprintf(stderr, "NAMED PIPE Init FAILED!\n");
   return EXIT_FAILURE;
}

/*
listenThreadHandle = CreateThread(
        NULL,
        0,
        listenThread,
        (LPVOID) nPipe,
        0,
        &listenThreadID);
if (listenThreadHandle == NULL){
   fprintf(stderr, "ActionThread Failed: %d\n", GetLastError());
}
*/

while(true){
   i = 0;
   if(count % 2 == 1)
      msg.messageType = TEMP;
   else
      msg.messageType = GPS;
   count++;
   memset(msg.message, 0, sizeof(msg.message));
   fprintf(stdout,">");
   fflush(stdin);
   while(i < MAX_MSG_SIZE){
      msg.message[i++] = getchar();
      if(msg.message[i-1] == '\n'){
         break;
      }
   }
   msg.message[i]='\0';
   msg.messageLen = i-1;
```

```
    if(sendToPipe(nPipe, &msg) == EXIT_FAILURE){
        fprintf(stderr, "SEND_TO_PIPE Failed!\n");
        return EXIT_FAILURE;
    }

    fprintf(stdout, "Sent data.\n");
}
return EXIT_SUCCESS;
```

tcp_server.cpp

Description: This code is ran second and should never be modified. It takes no startup arguments and will print out the name of the server which will be used for tcp_client to connect to during startup.

Code:

```
/* RMAX TCP Server
*
*
* @original author: Adam Mazzella <amazzell@calpoly.edu>
*
* Notes:
* 1. This one gets ran second followed by the TCP Client
* 2. No changes should be made to this code without consulting team in charge
*/

// Need to link with Ws2_32.lib
#pragma comment(lib, "ws2_32.lib")

#include <stdio.h>
#include <stdlib.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <iphlpapi.h>
#include "messageTypes.h"

#define TCP_SERVER_SOCKET "80085"
#define UDP_SERVER_SOCKET "80086"
#define TCP_SERVER_BACKLOG SOMAXCONN
#define MAX_CLIENTS 15
#define HOST_LEN 1500

//struct for client info
typedef struct CLIENTS
{
    int num_clients;
    SOCKET client_socket[MAX_CLIENTS];
    HANDLE nPipe;
} CLIENTS;

//prototypes
int handleClients(CLIENTS *clients);
DWORD WINAPI ClientThread(LPVOID lpvParam);
SOCKET tcp_server_setup();
void print_IPC_Message(IPC_Message *msg);
void print_IPC_Message_Template(IPC_Message_Template out);
```

```
int main(int argc, char * argv[])
{
    WSADATA wsaData;
    SOCKET tcp_serverSocket = INVALID_SOCKET;
    int iResult;
    struct addrinfo *addrInfo = NULL, hints;
    HANDLE tcp_cThread;
    DWORD threadID;
    DWORD flag = 1;
    DWORD bytesWritten;
    bool success;

    IPC_Message_Template *verify = (IPC_Message_Template
*)malloc(sizeof(IPC_Message_Template));
    verify->messageType = TCP_SERVER;
    verify->messageLen = strlen("RMAX PA$$CODE");

    memset(verify->message, '\0',MAX_MSG_SIZE);  //Need to null out the memory first

    memcpy(verify->message, "RMAX PA$$CODE", verify->messageLen);
    //opens pipe for communication with IPC Server
    CLIENTS * clients = (CLIENTS *) malloc(sizeof(CLIENTS));
    clients -> num_clients = 0;
    clients -> nPipe = CreateFile(
        TEXT(DEFAULT_PIPE_NAME),
        GENERIC_READ |
        GENERIC_WRITE,
        0,
        NULL,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL |
        FILE_FLAG_DELETE_ON_CLOSE,
        NULL);

    if(INVALID_HANDLE_VALUE == clients -> nPipe){
        fprintf(stderr, "Error connecting to server: %d\n", GetLastError());
        return EXIT_FAILURE;
    }

    if(ERROR_PIPE_BUSY == GetLastError()) {
        fprintf(stderr, "Busy Pipe: %d\n", GetLastError());
        return EXIT_FAILURE;
    }

    fprintf(stdout,"CREATION PIPE: %d\n",clients->nPipe);

    //Send Passcode to IPC_Server
    success = WriteFile(
        clients -> nPipe,
```

```
    verify,
    sizeof(IPC_Message_Template),
    &bytesWritten,
    NULL);

if(!success){
    fprintf(stderr, "Verification Failed: %d\n",GetLastError());
    return EXIT_FAILURE;
}

// Starting TCP Initialization
// Initialize Winsock
if (WSAStartup(MAKEWORD(2,2), &wsaData) != 0) {
    printf("WSAStartup failed with error: %d\n", WSAGetLastError());
    return EXIT_FAILURE;
}

//sets up tcp server socket
tcp_serverSocket = tcp_server_setup();
if(tcp_serverSocket == EXIT_FAILURE)
    return EXIT_FAILURE;


//threads off client handler
tcp_cThread = CreateThread(
    NULL,
    0,
    ClientThread,
    (LPVOID)clients,
    0,
    &threadID);


//listens for any added clients
while(true)
{
    if (listen( tcp_serverSocket, TCP_SERVER_BACKLOG ) == SOCKET_ERROR ) {
        printf( "Listen failed with error: %ld\n", WSAGetLastError() );
        closesocket(tcp_serverSocket);
        WSACleanup();
        return EXIT_FAILURE;
    }

    clients -> client_socket[clients -> num_clients] = accept(tcp_serverSocket, NULL, NULL);
    if (clients -> client_socket[clients -> num_clients] == INVALID_SOCKET) {
        printf("accept failed: %d\n", WSAGetLastError());
        closesocket(tcp_serverSocket);
        WSACleanup();
        return EXIT_FAILURE;
    }
```

```
        clients -> num_clients++;
        printf("Client added with socket number %d number of clients is now %d\n",
            clients -> client_socket[(clients -> num_clients) - 1], clients -> num_clients);
    }
    WSACleanup();
    free(verify);
    return 0;
}

DWORD WINAPI ClientThread(LPVOID lpvParam){
    CLIENTS *clients = (CLIENTS *)lpvParam;
    return handleClients(clients);
}

//sets up tcp socket for server
SOCKET tcp_server_setup()
{
    SOCKET tcp_serverSocket = INVALID_SOCKET;
    int iResult;
    struct addrinfo *addrInfo = NULL, hints;
    char hostname[HOST_LEN];
    ZeroMemory(&hints, sizeof (hints));
    hints.ai_family = AF_INET;       //IPV4 only
    hints.ai_socktype = SOCK_STREAM; //TCP Stream Type
    hints.ai_protocol = IPPROTO_TCP; //TCP Protocal Type
    hints.ai_flags = AI_PASSIVE;     //Socket will be used to bind

    // Resolve the local address and port to be used by the server
    iResult = getaddrinfo(NULL, TCP_SERVER_SOCKET, &hints, &addrInfo);
    if (iResult != 0) {
        printf("getaddrinfo failed: %d\n", iResult);
        WSACleanup();
        return EXIT_FAILURE;
    }

    // Create the TCP Socket
    tcp_serverSocket = socket(addrInfo->ai_family, addrInfo->ai_socktype, addrInfo->ai_protocol);
    if (tcp_serverSocket == INVALID_SOCKET) {
        printf("Error at socket(): %ld\n", WSAGetLastError());
        freeaddrinfo(addrInfo);
        WSACleanup();
        return EXIT_FAILURE;
    }

    // Bind the Socket
    iResult = bind(tcp_serverSocket, addrInfo->ai_addr, (int)addrInfo->ai_addrlen);
    if (iResult == SOCKET_ERROR) {
        printf("bind failed with error: %d\n", WSAGetLastError());
        freeaddrinfo(addrInfo);
```

```
        closesocket(tcp_serverSocket);
        WSACleanup();
        return EXIT_FAILURE;
    }
    if(gethostname(hostname, sizeof(hostname)) < 0)
    {
        perror("gethostname call");
        exit(-1);
    }
    printf("Server Name: %s\n", hostname);
    return tcp_serverSocket;

}

//sends IPC_Message to TCP Client(s)
int handleClients(CLIENTS *clients)
{
    DWORD read = 0;
    int num_clients;
    int send_len;
    bool success;
    WSADATA wsaData;

    IPC_Message * msg = (IPC_Message *)malloc(sizeof(IPC_Message));
    if(msg == NULL){
        perror("readFromPipe: MALLOC Failed");
        return EXIT_FAILURE;
    }


    while(true){

        success = ReadFile(
                clients -> nPipe,
                msg,
                sizeof(IPC_Message),
                &read,
                NULL);
        if(!success){
            printf("Reading from NAMED PIPED Errored %d\n", GetLastError());
            return EXIT_FAILURE;
        }
        //print_IPC_Message(msg);
        num_clients = clients -> num_clients;
        for(int i = 0; i < num_clients; i++)
        {
            //printf("Sending to client %d socket number %d status: ", i, clients -> client_socket[i]);
            send_len = send(clients -> client_socket[i],(char *) msg, sizeof(IPC_Message), 0);
            if(send_len == SOCKET_ERROR)
            {
```

```
                printf("FAILED\n");
                printf("Client with socket number %d removed number of clients is now %d\n", clients ->
client_socket[i], num_clients - 1);
                closesocket(clients -> client_socket[i]);
                for(int j = i; j < num_clients; j++)
                    clients -> client_socket[j] = clients -> client_socket[j+1];
                clients -> num_clients--;
                num_clients--;
                i--;
                //add error fixing
            }
            //else
                //printf("SUCCESSFUL\n");
        }
    }
    free(msg);
}

//printing functions for debugging will need to be modified as IPC structs are modified
void print_IPC_Message(IPC_Message *msg)
{
    print_IPC_Message_Template(msg -> gps);
    print_IPC_Message_Template(msg -> temp);
}

void print_IPC_Message_Template(IPC_Message_Template out)
{
    switch(out.messageType)
    {
        case GPS:
            printf("Message Type: GPS\n");
            break;
        case VIDEO:
            printf("Message Type: VIDEO\n");
            break;
        case ENGINE_DATA:
            printf("Message Type: ENGINE_DATA\n");
            break;
        case TEMP:
            printf("Message Type: TEMP\n");
            break;
        default:
            printf("Message Type: UNKNOWN\n");
            break;
    }
    printf("Message Length: %d\n", out.messageLen);
    printf("Message:\n");
    for(int i = 0; i < out.messageLen; i++)
    {
        putchar(out.message[i]);
```

```
    }
    putchar('\n');
```

tcp_client.cpp

Description: This code should be run third after tcp_server.cpp. This code needs the name of the server printed out by tcp_server to run correctly. This code should be modified to distribute the data to various parts of the mission that are on the ground. See the code for instructions on editing.

Code:

```
/* RMAX TCP Client
*
*
* @original author: Adam Mazzella <amazzell@calpoly.edu>
*
* Notes:
* 1. This is the last program to be run of the three
* 2. Your code to send to your station will go into the commented section of main
* 3. All data station code should be in separate file and header in theory there should just be
*    send to functions written by various groups
*/

#define WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdlib.h>
#include <stdio.h>
#include "messageTypes.h"

// Need to link with Ws2_32.lib, Mswsock.lib, and Advapi32.lib
#pragma comment (lib, "Ws2_32.lib")
#pragma comment (lib, "Mswsock.lib")
#pragma comment (lib, "AdvApi32.lib")


#define DEFAULT_BUFLEN 512
#define DEFAULT_PORT "80085"

//function prototypes
void print_IPC_Message(IPC_Message *msg);
void print_IPC_Message_Template(IPC_Message_Template out);

int __cdecl main(int argc, char **argv)
{
    WSADATA wsaData;
```

```
SOCKET ConnectSocket = INVALID_SOCKET;
struct addrinfo *result = NULL,
            *ptr = NULL,
            hints;
IPC_Message * msg = (IPC_Message *)malloc(sizeof(IPC_Message));
int iResult;

// Validate the parameters
if (argc != 2) {
    printf("usage: %s server-name\n", argv[0]);
    return 1;
}

// Initialize Winsock
iResult = WSAStartup(MAKEWORD(2,2), &wsaData);
if (iResult != 0) {
    printf("WSAStartup failed with error: %d\n", iResult);
    return 1;
}

ZeroMemory( &hints, sizeof(hints) );
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;

// Resolve the server address and port
iResult = getaddrinfo(argv[1], DEFAULT_PORT, &hints, &result);
if ( iResult != 0 ) {
    printf("getaddrinfo failed with error: %d\n", iResult);
    WSACleanup();
    return 1;
}

// Attempt to connect to an address until one succeeds
for(ptr=result; ptr != NULL ;ptr=ptr->ai_next) {

    // Create a SOCKET for connecting to server
    ConnectSocket = socket(ptr->ai_family, ptr->ai_socktype,
        ptr->ai_protocol);
    if (ConnectSocket == INVALID_SOCKET) {
        printf("socket failed with error: %ld\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }

    // Connect to server.
    iResult = connect( ConnectSocket, ptr->ai_addr, (int)ptr->ai_addrlen);
    if (iResult == SOCKET_ERROR) {
        closesocket(ConnectSocket);
        ConnectSocket = INVALID_SOCKET;
```

```
            continue;
        }
        break;
    }

    freeaddrinfo(result);
    if (ConnectSocket == INVALID_SOCKET) {
        printf("Unable to connect to server!\n");
        WSACleanup();
        return 1;
    }
    printf("Connected to server\n");
    // shutdown the connection since no data will be sent only recieved
    iResult = shutdown(ConnectSocket, SD_SEND);
    if (iResult == SOCKET_ERROR) {
        printf("shutdown failed with error: %d\n", WSAGetLastError());
        closesocket(ConnectSocket);
        WSACleanup();
        return 1;
    }

    // Receive until Connection Ends
    do {

        iResult = recv(ConnectSocket, (char *)msg, sizeof(IPC_Message), 0);
        if ( iResult > 0 )
        {
            //THIS IS WHERE YOU ADD YOUR CODE TO DO WHAT YOU WANT WITH THE DATA
            //CURRENTLY IT ONLY PRINTS OUT THE CONTENT of the IPC_Message recieved
            printf("Bytes received: %d\n", iResult);
            print_IPC_Message(msg);

        }
        else
            printf("Connection Ended");

    } while( iResult > 0 );

    // cleanup
    closesocket(ConnectSocket);
    WSACleanup();

    return 0;
}

//printing functions for IPC_Message will need to be modified when IPC_Message is modified
void print_IPC_Message(IPC_Message *msg)
{
    print_IPC_Message_Template(msg -> gps);
    print_IPC_Message_Template(msg -> temp);
```

```
}

void print_IPC_Message_Template(IPC_Message_Template out)
{
    switch(out.messageType)
    {
        case GPS:
            printf("Message Type: GPS\n");
            break;
        case VIDEO:
            printf("Message Type: VIDEO\n");
            break;
        case ENGINE_DATA:
            printf("Message Type: ENGINE_DATA\n");
            break;
        case TEMP:
            printf("Message Type: TEMP\n");
            break;
        default:
            printf("Message Type: UNKNOWN\n");
            break;
    }
    printf("Message Length: %d\n", out.messageLen);
    printf("Message:\n");
    for(int i = 0; i < out.messageLen; i++)
    {
        putchar(out.message[i]);
    }
    putchar('\n');
}
```

messageTypes.h

Description: the header file that defines the packets sent throughout the network. Modify IPC_Message_Template as needed for added datatypes and use. Please note that the same messageTypes must be used for each build of an executable and if different ones are used the network will not function correctly.

Code:

```
/* RMAX TCP Client
*
*
* @original author: Adam Mazzella <amazzell@calpoly.edu>
*
* Notes:
* 1. This is the last program to be run of the three
* 2. Your code to send to your station will go into the commented section of main
* 3. All data station code should be in separate file and header in theory there should just be
*    send to functions written by various groups
*/

#define WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdlib.h>
#include <stdio.h>
#include "messageTypes.h"

// Need to link with Ws2_32.lib, Mswsock.lib, and Advapi32.lib
#pragma comment (lib, "Ws2_32.lib")
#pragma comment (lib, "Mswsock.lib")
#pragma comment (lib, "AdvApi32.lib")


#define DEFAULT_BUFLEN 512
#define DEFAULT_PORT "80085"

//function prototypes
void print_IPC_Message(IPC_Message *msg);
void print_IPC_Message_Template(IPC_Message_Template out);

int __cdecl main(int argc, char **argv)
{
    WSADATA wsaData;
    SOCKET ConnectSocket = INVALID_SOCKET;
    struct addrinfo *result = NULL,
                    *ptr = NULL,
```

```
            hints;
IPC_Message * msg = (IPC_Message *)malloc(sizeof(IPC_Message));
int iResult;

// Validate the parameters
if (argc != 2) {
    printf("usage: %s server-name\n", argv[0]);
    return 1;
}

// Initialize Winsock
iResult = WSAStartup(MAKEWORD(2,2), &wsaData);
if (iResult != 0) {
    printf("WSAStartup failed with error: %d\n", iResult);
    return 1;
}

ZeroMemory( &hints, sizeof(hints) );
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;

// Resolve the server address and port
iResult = getaddrinfo(argv[1], DEFAULT_PORT, &hints, &result);
if ( iResult != 0 ) {
    printf("getaddrinfo failed with error: %d\n", iResult);
    WSACleanup();
    return 1;
}

// Attempt to connect to an address until one succeeds
for(ptr=result; ptr != NULL ;ptr=ptr->ai_next) {

    // Create a SOCKET for connecting to server
    ConnectSocket = socket(ptr->ai_family, ptr->ai_socktype,
        ptr->ai_protocol);
    if (ConnectSocket == INVALID_SOCKET) {
        printf("socket failed with error: %ld\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }

    // Connect to server.
    iResult = connect( ConnectSocket, ptr->ai_addr, (int)ptr->ai_addrlen);
    if (iResult == SOCKET_ERROR) {
        closesocket(ConnectSocket);
        ConnectSocket = INVALID_SOCKET;
        continue;
    }
    break;
```

```
    }

    freeaddrinfo(result);
    if (ConnectSocket == INVALID_SOCKET) {
        printf("Unable to connect to server!\n");
        WSACleanup();
        return 1;
    }
    printf("Connected to server\n");
    // shutdown the connection since no data will be sent only recieved
    iResult = shutdown(ConnectSocket, SD_SEND);
    if (iResult == SOCKET_ERROR) {
        printf("shutdown failed with error: %d\n", WSAGetLastError());
        closesocket(ConnectSocket);
        WSACleanup();
        return 1;
    }

    // Receive until Connection Ends
    do {

        iResult = recv(ConnectSocket, (char *)msg, sizeof(IPC_Message), 0);
        if ( iResult > 0 )
        {
            //THIS IS WHERE YOU ADD YOUR CODE TO DO WHAT YOU WANT WITH THE DATA
            //CURRENTLY IT ONLY PRINTS OUT THE CONTENT of the IPC_Message recieved
            printf("Bytes received: %d\n", iResult);
            print_IPC_Message(msg);

        }
        else
            printf("Connection Ended");

    } while( iResult > 0 );

    // cleanup
    closesocket(ConnectSocket);
    WSACleanup();

    return 0;
}

//printing functions for IPC_Message will need to be modified when IPC_Message is modified
void print_IPC_Message(IPC_Message *msg)
{
    print_IPC_Message_Template(msg -> gps);
    print_IPC_Message_Template(msg -> temp);
}

void print_IPC_Message_Template(IPC_Message_Template out)
```

```
{
    switch(out.messageType)
    {
        case GPS:
            printf("Message Type: GPS\n");
            break;
        case VIDEO:
            printf("Message Type: VIDEO\n");
            break;
        case ENGINE_DATA:
            printf("Message Type: ENGINE_DATA\n");
            break;
        case TEMP:
            printf("Message Type: TEMP\n");
            break;
        default:
            printf("Message Type: UNKNOWN\n");
            break;
    }
    printf("Message Length: %d\n", out.messageLen);
    printf("Message:\n");
    for(int i = 0; i < out.messageLen; i++)
    {
        putchar(out.message[i]);
    }
    putchar('\n');
}
```