Vanderbilt University School of Engineering

Department of Electrical Engineering and Computer Science

CmpE204

Independent Study: Embedded Flight Design

End of Semester Report

Submitted: May 7, 2010

Thomas Griffith

**Table of Contents**

1. Introduction………………………………………………………………………..2
2. Involvement………………………………………………………………………..3
3. Communication…………………………………………………………………….3
4. LL to Base Station……………………………………………………………..3
5. HL to Base Station……………………………………………………………..5
6. HL to LL……………………………………………………………………….6
7. Loopback Control Example…………………………………………………….7
8. Thrust Measurement………………………………………………………………..8

E. FRODO……………………………………………………………………………..10

1. Challenges………………………………………………………………………….11
2. Conclusion and Lessons Learned ………………………………………………….11
3. Appendix…………………………………………………………………………...12

**I. Introduction:**

My semester project has been concerned with several aspects of the embedded quadrotor from Ascending Technologies, called the Hummingbird with Autopilot. The purpose of this project is to demonstrate a control system that controls the flight of the quadrotor in such a way that it is autonomous in nature. This is done by having one or more base station controllers that communicate wirelessly with the quadrotor via the IEEE 802.15.4 standard, using ZigBee radio modules. The communication messages that are involved between the quadrotor and the base stations can include packets that contain data such as GPS coordinates, acceleration, velocity, height, attitude data, thrust, battery voltage, compass readings, raw sensor outputs, and remote control information. The focus of this project is on the attitude and thrust data, since they are the most important for the basic control of an aircraft. Thus, one of the main goals of this project was to calculate the amount of thrust and determine how various conditions of the vehicle affected the thrust so that a control system can effectively operate and guide the quadrotor.

This project has dealt with several aspects of the communication of the quadrotor because of the fact that the quadrotor is versatile enough to not be limited to only one form of communication. The most basic form of communication was between the low-level board of the quadrotor and the base station controller because the protocol was already developed and only had to be implemented by the user. Next, the communication between the high-level board and the low-level board was of much greater difficulty because access is very limited to the low-level board in such a way that the code on it cannot be modified or viewed. However, the communication between the high-level board and the base station is less difficult in that only a protocol has to be developed to handle the packets that are sent between the two. Finally, the communication among the base-station controllers calls for highly embedded programming using the communication API’s of various operating systems so that they pass messages amongst each other. Thus, the other main objective of this project was to implement the communications of the quadrotor in such a way that a control example could be realized.

**II. Involvement:**

1. *Communication and Control:*
2. *LL board to Base Station:*

The communication between the low-level board and the base station is perhaps the simplest and most straight forward of the communication forms as it utilizes an already developed protocol from Ascending Technologies. The protocol utilizes the serial bus in order to send packets between the ZigBee module on the low-level board and the ZigBee module on the base station. The ZigBee module on the low-level board is connected to the port on the quadrotor’s board called “LL serial 0” via what appears to be a custom connector, whereas the ZigBee module on the base station utilizes the standard mini-USB connector through the virtual “COM X” port on Windows and the “/dev/ttyUSB0” port on Linux with both ports operating at a baud rate of 57600.

The protocol for input to the low-level board differs from the protocol for output from the low-level board in that the messages contain different information. The input to the board consists of a start string containing the string of chars “>\*>”. Then a description char (or chars) is used to determine what kind of data is being input. This can be ‘m’ for turn motors on, ‘p’ for data struct request, or “di” for data input. When ‘m’ is used, it should be followed by a short of either 0 or 1 to switch the motors off or on. When ‘p’ is used, it should be followed by a short that signifies the specific structure of the requested data structure, which can be seen in the appendix of this report. After this is sent through the serial interface, a packet containing the requested structure can be expected to arrive. When “di” is used, the pitch, roll, yaw, thrust, control byte (where bit 0 enables pitch, the bit 1 enables roll, bit 2 enables yaw, and bit 3 enables thrust), and checksum (calculated by adding the other five data with 0xAAAA) must be sent as shorts in that order directly following “>\*>di”.

The protocol that is used as output from the LL board is different in that six different pieces of information are used. First, the same start string as the other protocol, “>\*>”, is utilized to denote the beginning of the packet. Then, an unsigned short that denotes the length in bytes of the respective structure is sent over the serial interface. Followed by that is the unsigned char, packet descriptor, which corresponds to the respective data struct that is being sent and may be viewed in the appendix of this report as well. The descriptor’s number can then be used to determine the actual types of each of the bytes that are to follow. Next, the actual data structure is sent, and the type will vary depending on the specific structure. Many of these data structures are also included in the appendix of this report. After the data structure, the crc16 is sent as an unsigned short and is calculated using the standard crc16 algorithm. Finally, a stop string is appended to the end, which includes the string of characters “<\*<”.

The sending and receiving of data from and to the base station was utilized with a script written in Python 2.6. The script requires the serial package to utilize the serial interface, the struct package to pack and unpack data represented as C types, and the ctypes package to calculate the checksum value in terms of a C short. The script works by first opening the respective communication port as a serial object (“COMX” for Windows or “/dev/ttyUSB0” for Linux). Then, data can either be written to or read from the serial object by invoking read() or write(). When data is being read, it should be unpacked using the struct module, and, likewise, when data is being written, it should be packed using the struct module. The order in which this script called these functions was by first writing a motor or attitude command, then writing a request for a packet (such as status), and then reading that packet. This was all done in a loop such that packets are continuously being written and read. The reading of the packet was done in such a way that states are used to keep track of which byte is currently being read in the packet.

Both Windows and Linux (using the same host machine) were tested to see which had the shortest round-trip time by using the NumPy and SciPy modules of Python to calculate the mean and standard deviations of their round-trip times based on 1,000 round-trips, which included the sending of a motor start/stop packet, sending of an LL\_Status structure request, and receiving of the LL\_Status data structure. Windows was found to have a mean round-trip time of 74.0 ms with a standard deviation of 13.8 ms, while Linux was found to have a mean round-trip time of 31.9 ms with a standard deviation of 0.929 ms; thus, Linux is the clear winner. It is interesting to note that the deviated times tend to come in pairs with one being about half the other, which suggests that a packet is dropped.

The best possible time for the round-trip of this example that includes the sent motor control packet, the sent request packet, and the received status packet can be calculated as follows. Where LMS is the length of the sent messages, LMR is the length of the received message, LH is the length of the MAC layer protocol, and br is the baudrate:

∆Tbest = (2LMS + LMR + 2LH) / br

∆Tbest = [((2(10 \* 8)) bits) + ((28 \* 8) bits) + ((2(14 \* 8)) bits)] / (57600 baud)

∆Tbest = 10.6 ms

Thus, the difference between the best possible round-trip time and the observed mean round-trip time is 21.3 ms. This difference can most likely be accounted for the wireless transmission delay of the ZigBee radio module.

The LL to base station communication is thus effective, since a control system doing the calculations on the base station could probably handle 32 Hz fine. However, it is still a bit inefficient because of the fact that the data that feeds the control must be transferred wirelessly twice. This leads us to explore alternative methods in communication.

1. *HL board to Base Station*

In addition to the LL board being able to communicate with the base station, the HL board can communicate with the base station as well. In order to do this, a protocol must be created by the user that handles both the input and output of user created and defined packets. In addition, all communication messages must be sent over the “HL 0” serial port which is located on the top board of the quadrotor. This port uses the same custom connection developed by Ascension Technologies. However, there is no ZigBee module already connected to this port that comes with the quadrotor; thus, it is up to the user to use his own ZigBee module with a custom connection adapter to fit it to the serial port. In my example, I used the custom female mini-USB to the custom connection dongle to do this (Note: this dongle is the female side of the mini-USB, and most ZigBees are female as well, hence the ZigBee is not being used). Thus, this example bypasses the wireless communication, but it still shows that a custom protocol is possible.

In order for one to implement a protocol to handle messages between the HL board and the base station, one must make modifications to the code in the SDK, which is provided by Ascension Technologies. All of the code is placed inside the “Autopilot\_HL\_SDK” folder and should be opened with Ascension Technologies’ packaged eclipse environment. Install instructions and the setup executable are located in the parent folder, and it should be noted that this has only been tested on a Windows machine. However, any machine with the ARM gcc compiler, OpenOCD and the JTAG drivers installed should be able to compile and build these files within the SDK. The file that is used by the HL board itself is main.hex, which is always generated after a successful build. To download this to the board, the mini-USB to AscTec’s custom connection dongle should be connected to the “HL serial 0” port on the board. Then, the quadrotor should be turned on while jumping the HL bootloader jumper pads, which are small gold contacts. A loud buzz should now indicate that it is ready to be flashed. One can then flash main.hex to the HL board by following the directions in the appendix of this report. When the flash is done, the quadrotor can be turned off.

For the quadrotor to receive data to the HL board from the serial port, the ISR labeled “void uart0ISR(void) \_\_irq” in uart.c is the handler for all received data from the serial bus, and it interrupts whenever new data comes into the “HL serial 0” port. To develop a protocol, this ISR must be modified in such a way that new packet descriptors are added. To do this, case 2 should be modified to add these new descriptors because the data are pulled in this case from the receiver buffer register of the UART circuit. These descriptors should represent new data structures that are to be created by the developer. Once this is done, the data structures can be updated in the SDK loop, which is called at 1 kHz and is located in sdk.c. It is then the programmer’s choice to do whatever he wants with the new received data. The data structures and protocols can be anything, but the sender on the base station should take account for the structure of the packet including that of the data structure as should be done with the packets in for the LL protocol.

In order for the quadrotor to send data from the HL board out to the serial port, the UART\_SendPacket(void \*data, unsigned short count, unsigned char packetdescriptor) function in uart.c should be called. This function sends data through the UART chip out onto the serial bus. A data structure may be passed as \*data, where count is the size of the structure, and packetdescriptor is the 1 byte description that distinguishes this packet from other packets, so that the receiver knows what to do with it. The HL board to base station communication is thus used in the loopback control example that I developed.

1. *HL board to LL board*

The HL board may also communicate with the LL board to a certain extent. The reason why it is limited is because the code on the LL board cannot be modified, thus only certain data can be transferred between them. Two data structs exist on the LL board that can be read and updated by the HL board. These two structs are the LL\_1khz\_control\_input and LL\_1khz\_attitude\_data and are located within the file, LL\_HL\_comm.c.

The LL\_1khz\_control\_input struct is just like it sounds in that it allows one to input data to the LL board immediately. Pretty much all of the data available on the quadrotor can be stored in this structure, such as attitude commands, thrust, speed, heading, and GPS coordinates, and, in addition to being used to input data, it can be read from to determine the values of other data on the quadrotor, such as battery voltage, accuracy, and status information. This data struct is written to the LL processor by invoking the HL2LL\_write\_cycle(void) method within the LL\_HL\_comm.c, however this method is called every iteration within the main loop and is thus called at 1 kHz already.

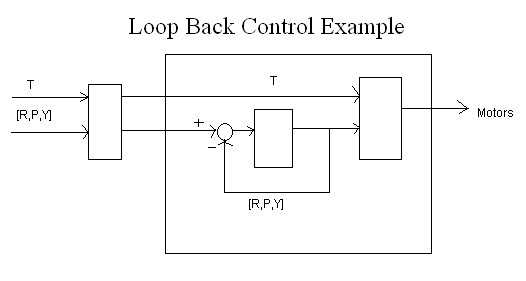
The LL\_1khz\_attitude \_data struct is mainly used for updating information on the HL board as this struct is never actually written to while using the HL board. The data stored in it consists of attitude data, acceleration, orientation data, height, and remote control data. Thus, it provides the HL board with the sensor information that only the LL board has direct access to. Since this struct is providing data to the HL board, it is called within the SSP\_rx\_handler\_HL(unsigned char SPI\_rxdata) method in LL\_HL\_comm.c, which handles all of the receives of LL data on the HL board. Thus, the data on the LL board is actually being sent to the HL board by means of a serial bus.

1. *Loopback control example*

In order to test and verify the quadrotor, a simple loopback control example was developed. In this loopback control system, the current data that is about to be fed into the motors is looped back to a controller. In this case, the controller is really just the base station, and the values that are being pushed into the motor are printed out on the screen as they are actually being pushed. HL board to LL board communication is utilized in my loopback control example as data needs to be passed from the HL board down to the LL board so that the motors can spin according to the right pitch, roll, yaw, and thrust. Likewise, in this loopback control system, data needs to be retrieved from the LL board in order to update what is currently happening.

In my example, I developed my own protocol to handle data being transferred from the base station to the high-level controller and then back again from the high-level controller to the base-station. I added two new packets, with attitude denoted by the descriptor ‘a’, and thrust denoted by the descriptor ‘t’, to the uart0ISR(void) function. Two new data structs were also created, with Attitude holding shorts called pitch, roll, and yaw and with Thrust holding a short called value. These struct members are all set equal to the receiver buffer register when their respective value enters the receiver buffer register. In addition to this, in the SDK loop, which is called at 1 kHz, the buffer for the LL motor controller is set equal to the values in the Attitude and Thrust structures. Then, the main loop writes that data for immediate transfer to the LL processor, where it is sent to the motors. In addition, the data is sent back to the base station from the HL board to complete the loop.

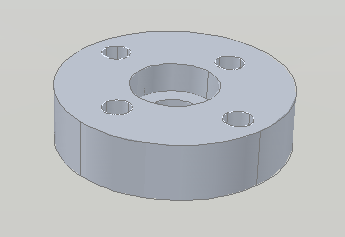
The results of my control example proved to be mostly correct as the thrust did change on both the motors and on the base station as they should have with the changes that I kept making to the thrust values. The values of the pitch, roll, and yaw could be seen on the base station after they were changed and sent to the HL board, but I could not tell if they were actually being represented on the motors themselves because the quadrotor was tethered down to a table. In fact, the quadrotor had to stay connected to a USB cable the entire time because there is not a ZigBee module connected to the “HL Serial 0” port as was previously mentioned. However, I believe that these values were actually being represented on the motors because the code is practically the same as that for the thrust, which did prove to work. Thus, I feel that the control example was a successful representation of communication and control of the quadrotor using the HL board.



1. *Thrust measurement:*

To control the quadrotor in a deterministic manner, the behavior of the thrust of the quadrotor should be known so that various control techniques can be utilized to account for any unexpected events. It is known that certain factors can affect the performance of motors, specifically the motors used on a quadrotor craft. It is said that various factors such as age of the propeller and motor, battery voltage, etc. can cause the motor to perform differently. Specifically, this can be the case with thrust because it involves how much force the motor can exert on something, and if it is the case that the motor is not in prime condition, then the amount of force exerted could be different than from the amount of force exerted from that motor in new condition. Thus, it is important to measure the thrust of the quadrotor in certain conditions.

This experiment utilized a 10 lb compression/tension load cell, called the LC703 from Omegadyne, in order to correctly measure the amount of thrust exerted by the motors. In order to determine the readings from the load cell, an Omega DP25-S digital meter was utilized. But first, a part had to be designed and machined to accommodate a motor on the load cell. Thus, thus part consisted of a 0.25” thick aluminum disc with a hole of the height 0.125” in the center for the screw head to sit on and for the motor’s spindle to clear. In addition to that, a 0.125” deep counter sink with a 10-32 thru hole is put in place to accommodate the 10-32 screw size that is in the center of the part. There are also four other 3-48 thru holes around the center hole. One pair of opposite holes is spaced about 0.75” from center to center while the other pair of holes is spaced about 0.625” from center to center. All of these holes can be drilled using a press. The rest of the assembly is assembled by attaching the part to the motor with the screw inside of it and then securing it with the four plastic screws that the motor came with. Next, put a spacer on the screw, and screw it into the load cell. The other side of the load cell can then be secured to a table with a strip of aluminum and a G-clamp.



The wiring from the load cell was then connected to the digital meter, according to the directions in the manual, which I high recommend reading. The type of sensor inputs used was the meter-powered bridge input. The calibration should then commence by first selecting the decimal point position. In this case, it is wise to choose FFF.F, since the expected value of the thrust is in the hundreds. Next, calibration is further done by scaling it without known loads. This is the tricky part, because values need to be taken from a table and multiplied together. The values used are in the appendix of this report, and should be utilized to configure the scaling of the digital meter.

For the load cell measurement, I utilized the LL board to base station communication with the standard protocol that is utilized by Ascension Technologies to send commands serially to the serial port. In doing so, the range of thrust command values from 0 to 4095 represent 0% to 100% throttle on the motor. The thrust commands for two different battery levels, one at a full 12.6 V and the other nearing empty at 11.1 V, were then sent over the ZigBee radio module one at a time, and the corresponding meter readings were recorded. The following graphs represent the thrust data. There is some slight variation between the data with the standard deviation between pairs of data being 1.29 g, where the lower voltage test had a bit lower numbers but not much. In addition, the 0 command value for 12.6 V was 12 g and for 11.1 V, it was 11 g. The highest value of 4095 gave 295 g for 12.6 V and 290 g for 11.1 V. Thus, the differences are minor.

1. *FRODO:*

In addition to working with just the hardware of the quadrotor, I also had the opportunity to do some development on FRODO, which is the real-time message passing system used for communicating between nodes. The goal of FRODO is to have a control system that can pass messages quickly amongst nodes. My main task in the development of FRODO was to develop high resolution timers on Linux based on the timers already developed for Windows and OSX. To incorporate the high resolution timers on Linux machines, I used the POSIX library, which has microsecond precision capabilities.

Furthermore, I developed test cases for the use of several nodes in one message passing system. These test cases utilized the FRODO API in order to synchronously schedule messages to be sent from one node to another node. In a test environment using a cluster of computers, messages from one node would be able to schedule arrivals on other nodes with an error of only a few microseconds, which is pretty phenomenal. By testing out several cases, I found a race condition within the core FRODO code, which was soon fixed and was able to further improve FRODO. This exercise introduced me to highly-embedded programming in a UNIX environment, which was a great way for me to expand my horizons in embedded systems.

**III. Challenges:**

By far the biggest challenge of this project was figuring out how the quadrotor worked with the very little documentation and support given by Ascending Technologies. The user manual did not mention much of anything about the programming of the actual HL SDK, and furthermore, the provided SDK had very little commenting. The only way to figure out what was going on was to read literally all the code several times, which is no trivial task considering there are about fifteen files with many lines of code each. I must say I didn’t fully understand the code until I completed the loopback control example; although I understand it pretty well now, there are still a few things that are not so obvious to me. The design of the quadrotor system itself also wasn’t that easy to follow in terms of the HL to LL communication.

In addition to the SDK not being commented, the serial communication that was provided for the LL board had very little information given about it. The only commands given in the manual were a bit cryptic to understand for someone who hasn’t worked with serial communication much, and the manual gave no obvious solutions of how to approach it. The manual just gave a few packet formats and data structs and expected the user to figure the rest out. And the absolute worst part about the serial communication documentation was that the serial command to start the motors was nowhere in the manual at all, nor available on the website. The command to send attitude data to the motors was there, but the command to start the motors was not, which makes the attitude command pointless if the motors can’t start. The company didn’t answer my several e-mails about it, thus I resorted to calling them to get the answer. In addition, to my knowledge, there is no way to start the motors by the HL board. In order for me to run the loopback example, I had to first start the motors with the LL command and the ZigBee, then disconnect the ZigBee, attach their custom adapter, and then start running code such as attitude commands to be able to use the HL board. I feel that much more could have been accomplished with this project if the company had done a better job with their documentation.

**IV. Conclusion and Lessons Learned**

In conclusion, I felt that this project was an overall success. Despite the hiccups with the documentation, I was still able to implement a control system with the quadrotor. I also developed methods of utilizing the various communication means of the quadrotor, including LL to base station, HL to base station, and HL to LL; and I even had to integrate all three of these together to complete the loopback example. Although the quadrotor may not be able to fly fully autonomously with tricks and the like, it can now implement the basic building blocks that can be utilized to make it behave in a more advanced manner. One huge lesson I learned, though, from the experience with the quadrotor company is to always ask questions if something doesn’t make sense.

I learned a great deal about engineering in general while working on this project. While working with the other team members, I learned how real engineers develop and test designs. Before now, I haven’t had much experience at all working with a team of real engineers, and now that the semester is over, I feel like I am taking with me a vast amount of much needed experience. I really feel that working with these guys has helped me a tremendous amount.

In addition to the team experience, I was able to put together a very large amount of my past education into one experience as this project has called for things that I have learned in many different classes such as Microcontrollers, Embedded Systems, Computer Networks, Real-time Systems, Statistics, and Operating Systems. I have never participated in such a project before that has required the knowledge of so many different areas, and this has helped me put it all together. Furthermore, this experience that I have gained from this project will also help in the very near future when I will be working on another embedded UAV control system in Germany this summer. Thus, I truly feel that this project was a success on several levels including education and achievement.

**V. Appendix:**

***All code is available on the SVN.***

*LL Communication and Control:*

Packet Descriptors for requesting data structures:

0x0001 LL\_Status

0x0002 IMU\_RawData

0x0004 IMU\_CalcData

0x0008 RC\_Data

0x0010 CTRL\_Out

0x0080 GPS\_Data

0x0100 current\_way

0x0200 GPS\_Data Advanced

0x0800 CAM\_Data

Packet Descriptors for sending data structures:

0x01 PD\_IMURAWDATA

0x02 PD\_LLSTATUS

0x03 PD\_IMUCALCDATA

0x04 PD\_HLSTATUS

0x05 PD\_DEBUGDATA

0x11 PD\_CTRLOUT

0x12 PD\_FLIGHTPARAMS

0x13 PD\_CTRLCOMMANDS

0x14 PD\_CTRLINTERNAL

0x15 PD\_RCDATA

0x16 PD\_CTRLSTATUS

0x17 PD\_CTRLINPUT

0x18 PD\_CTRLFALCON

0x20 PD\_WAYPOINT

0x21 PD\_CURRENTWAY

0x22 PD\_NMEADATA

0x23 PD\_GPSDATA

0x24 PD\_SINGLEWAYPOINT

0x25 PD\_GOTOCOMMAND

0x26 PD\_LAUNCHCOMMAND

0x27 PD\_LANDCOMMAND

0x28 PD\_HOMECOMMAND

0x29 PD\_GPSDATAADVANCED

*HL firmware flash information*

Using Flash Magic, ensure the following settings are correct:

Device: LPC2146

Oscillator: 14.7456 MHz

Interface: None(ISP)

Baudrate: 115200

COM: should correspond to Device Manager

Check “Erase all Flash and Code Rd Prot”

Select main.hex

Uncheck all of the options in Step 4

Click on Start Programming

*Scaling of digital meter for load cell:*

Maximum load: 10 lbs

Output: 2 mV/V

Sensor Excitation: 10 Vdc

Output: 20 mV

IN1: 0

IN2: 2000

Rd1: 0

Rd2: 10