AUTOMATED REFUELING FOR HOVERING ROBOTS

A Major Qualifying Project Report

Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements
for the Degree of Bachelor of Science

by

Nigel Cochran, RBE/ME

Janine Pizzimenti, RBE/ECE

Raymond Short, RBE/CS

Date: March 13, 2012

Approved:

Professor Ken Stafford, ME/RBE Advisor

Professor William Michalson, ECE/RBE/CS Advisor

Disclaimer: This work is sponsored by the Assistant Secretary of Defense for Research & Engineering under Air Force Contract #FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.
Abstract

Small-scale, battery-powered unmanned aerial vehicles (UAVs) suffer from short mission times before they must land for manual refueling, making the UAVs not truly autonomous for extended periods of time. This solution aims to be a significant improvement on previously proposed refueling solutions from the research literature, while adding the novel functionality of being universal for many UAVs that are battery powered and can perform vertical takeoff and landing. The proposed design is a base station that positions the landed UAV to a known orientation, then exchanges and charges the UAV’s battery. This solution allows for persistent flight of the UAV by maximizing its in-air duty cycle.
Abstract: Pizzimenti

1. Introduction: Cochran, Pizzimenti
2. Background: Cochran, Pizzimenti, Short
   2.1 Introduction: Cochran
   2.2 Lithium Polymer Batteries: Pizzimenti
   2.3 Existing Designs of Battery Exchanging Bases: Cochran, Short
3. Methodology: Cochran, Pizzimenti, Short
   3.1 Introduction: Pizzimenti
   3.2 Justification: Pizzimenti
   3.3 Design Overview: Pizzimenti
   3.4 UAV Alignment: Cochran
   3.5 Custom Skids: Cochran
   3.6 Custom Battery Holder: Cochran
   3.7 Battery Exchange: Cochran
   3.8 Lithium Polymer Charging: Pizzimenti
   3.9 System Level Flow: Short
   3.10 Wireless Communications: Short
   3.11 Program Design: Short
4. Results: Cochran, Pizzimenti
5. Discussion: Cochran, Pizzimenti, Short
   5.1 UAV Alignment Device: Cochran
   5.2 Battery Exchange Mechanism: Cochran
   5.3 Battery Stations: Cochran
   5.4 Future Recommendations: Cochran, Pizzimenti, Short
6. Conclusion: Cochran
# Table of Contents

Abstract ........................................................................................................................................... ii

Authorship ..................................................................................................................................... iii

Table of Contents ........................................................................................................................... iv

Table of Figures ............................................................................................................................. vii

Chapter 1: Introduction .................................................................................................................... 1

Chapter 2: Background .................................................................................................................... 3

  2.1 Introduction ............................................................................................................................ 3

    2.1.1 Justification ...................................................................................................................... 3

    2.1.2 Ascending Technologies Pelican UAV ............................................................................. 3

  2.2 Lithium Polymer Batteries ....................................................................................................... 4

    2.2.1 General ............................................................................................................................ 4

    2.2.2 Balanced Charging ......................................................................................................... 5

    2.2.3 Battery Charging vs. Exchanging .................................................................................. 5

      2.2.3.1 Existing Charge Only System ............................................................................... 5

      2.2.3.2 Charge vs. Exchange Economic Comparison .......................................................... 6

  2.3 Existing Designs of Battery Exchanging Bases ...................................................................... 6

    2.3.1 Landing Alignment ......................................................................................................... 7

    2.3.2 Battery Mating Systems ............................................................................................... 8

    2.3.3 Battery Transfer Mechanisms ...................................................................................... 9

    2.3.4 Battery Storage ............................................................................................................. 10

  2.4 Summary ............................................................................................................................... 12

Chapter 3: Methodology .................................................................................................................. 14

  3.1 Introduction ............................................................................................................................ 14

  3.2 Justification ............................................................................................................................ 14
# Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Complete Base</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>AscTec Pelican UAV</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>Charge Only UAV Landing Contacts</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>Passive Alignment Apparatus (Toksoz, 2011)</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>Active Alignment System (Suzuki, 2011)</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>UAV Rail Attachment and Battery Pack (Toksoz, 2011)</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>Leaf Spring Battery Attachment (Suzuki, 2011)</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>Servo Actuated Magnet Connection (Swieranga, 2010)</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>Rack and Pinion Transfer Mechanism (Toksoz, 2011)</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>Offset Battery Carousel</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>Centered Battery Carousel</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>Vertical Drum Battery Holder</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>X-Y Plane Battery Storage</td>
<td>12</td>
</tr>
<tr>
<td>14</td>
<td>In-Flight Duty Cycle Saturation</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>Diagram of Complete Base Station</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>Alignment Arm</td>
<td>18</td>
</tr>
<tr>
<td>17</td>
<td>Complete Landing Zone</td>
<td>18</td>
</tr>
<tr>
<td>18</td>
<td>Worst Case Torque on Alignment Arms</td>
<td>19</td>
</tr>
<tr>
<td>19</td>
<td>Touch Latches</td>
<td>21</td>
</tr>
<tr>
<td>20</td>
<td>Custom Touch Latches with Electrical Connectors</td>
<td>21</td>
</tr>
<tr>
<td>21</td>
<td>Symmetrical Battery Connection Layout</td>
<td>22</td>
</tr>
<tr>
<td>22</td>
<td>Top View of Cart</td>
<td>24</td>
</tr>
<tr>
<td>23</td>
<td>Entire Battery Cart</td>
<td>25</td>
</tr>
<tr>
<td>24</td>
<td>Venom Easy Balance LiPo Charger</td>
<td>26</td>
</tr>
<tr>
<td>25</td>
<td>Button Circuit Diagram</td>
<td>27</td>
</tr>
<tr>
<td>26</td>
<td>LED Circuit Diagram</td>
<td>27</td>
</tr>
<tr>
<td>27</td>
<td>System Level Flow Diagram</td>
<td>28</td>
</tr>
<tr>
<td>28</td>
<td>Completed Overall System</td>
<td>31</td>
</tr>
<tr>
<td>29</td>
<td>Landing Mechanism</td>
<td>31</td>
</tr>
<tr>
<td>30</td>
<td>Battery Cart</td>
<td>32</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>31</td>
<td>Battery Transfer Mechanism</td>
<td>32</td>
</tr>
<tr>
<td>32</td>
<td>Battery Charging Station</td>
<td>33</td>
</tr>
<tr>
<td>33</td>
<td>UAV Electrical Contacts</td>
<td>33</td>
</tr>
<tr>
<td>34</td>
<td>Battery Pack Electrical Contacts</td>
<td>33</td>
</tr>
<tr>
<td>35</td>
<td>Battery Pack</td>
<td>33</td>
</tr>
<tr>
<td>36</td>
<td>Modified Battery Charger</td>
<td>33</td>
</tr>
<tr>
<td>37</td>
<td>Pololu Controller</td>
<td>33</td>
</tr>
</tbody>
</table>
Chapter 1: Introduction

Unmanned aerial vehicles (UAVs) are an increasingly important focus for academic and military robotics research. These UAVs are typically small, commercially available quad-rotors or helicopters, which have the ability to perform fully autonomous missions for a variety of applications such as aerial surveillance. For this project, the refueling system was designed for the small, battery-powered Ascending Technologies Pelican UAV.

On-board lithium polymer (LiPo) batteries fuel many of these autonomous aerial vehicles because they have a high energy density and the ability to sustain high current loads. However, the battery’s increased weight translates directly to increased energy required from the UAV’s motors, limiting its flight time and mission abilities. Therefore, these UAVs suffer from short mission times before they must land to refuel. This refueling process typically requires human intervention, making the UAVs not truly autonomous for extended periods of time.

A fully autonomous system for refueling these small-scale UAVs can increase mission time and minimize human involvement in these systems. The proposed system is composed of a base station that reliably positions the UAV after landing, then exchanges and charges the UAV’s batteries. This allows for persistent flight of the UAV by maximizing its in-air duty cycle. This base station is also designed to be easily made universal for many UAVs that are battery powered and can perform a vertical takeoff and landing through the use of custom skids.

Several institutions have already developed systems that aim to fill this need through methods of exchanging batteries, charging batteries, or performing both operations. This solution aims to be a significant improvement on previous designs present in the research literature, while adding the novel functionality of being usable with any small battery-powered UAV. The proposed solution contains the following features:

- Custom UAV skids to allow multiple types of hovering robots to use the same charging base
- A large landing area that can handle a wide range of landing errors through an active alignment system
- A new approach to battery terminal connectors that allows for easy attachment and release
- The capability to store and charge up to eight lithium polymer batteries
The base described in this paper and shown below in Figure 1, was a successful first prototype and proof of concept, which was able to perform a complete battery exchange. However, it was unable to reliably perform battery exchanges due to a problem with the manner in which electrical connections were made.

![Figure 1: Complete Base](image)

However, through some modifications, this base can be transformed into a fast and reliable device capable of exchanging the batteries of a UAV. Currently, it has already proven reliable at centering the UAV while accounting for error in both position and orientation. Furthermore, through a combination of hardware and software, it is capable of coping with misalignments in a reliable manner. Finally, using consumer off the shelf (COTS) chargers and stationary charging stations, this solution is able to successfully charge a lithium polymer battery autonomously. Therefore, despite the unreliability of the final product, this project should be considered a success as the research performed, lessons learned, and base developed pave the way for a fast and reliable solution.
Chapter 2: Background

2.1 Introduction

2.1.1 Justification

The need for an autonomous system to replace the battery of an Unmanned Aerial Vehicle (UAV) is directly related to the need for UAVs. In 2009, the Department of Defense (DOD) published its FY2009-2034 Unmanned Systems Integrated Roadmap dictating its plan for the use of unmanned systems in the United States Military. The report notes that unmanned systems have been well received by commanders in the field for both their versatility and persistence. It highlights that systems have transitioned from being remote controlled to fully autonomous, and that this autonomy will only become more useful in the future. Part of the increase in autonomy will be in the area of ground support, where the DOD hopes to have automated aircraft refueling in its inventory as early as 2016. This goal is directly aligned with the creation of a system capable of servicing the battery of a UAV. (United States, 2009)

The necessity for UAVs is also apparent in the goals set out by the roadmap. The number one priority continues to be reconnaissance and surveillance while the second priority is target identification and designation (United States, 2009). Both of these tasks are perfectly suited to small scale UAVs. However, the batteries that power current systems lack the energy to remain powered for extended periods of time. The implementation of an autonomous system capable of replacing and recharging the battery of a UAV allows for extended mission lengths and the benefit would be even greater if this apparatus was integrated with an Unmanned Ground Vehicle (UGV).

In addition to the military needs, many research institutions have a need for an autonomous battery exchanging system, as it assists in experimentation. This is the field where previous systems have already been developed. Three different systems capable of autonomously refueling UAVs have been identified and are discussed later in this section. Additionally, other already existing systems focus on some aspects of this project such as the balancing of lithium polymer batteries and predicting the lifetime of these batteries.

2.1.2 Ascending Technologies Pelican UAV

The UAV this autonomous battery exchange system is designed for is the Pelican made by Ascending Technologies. It is a robust and modular robotic platform capable of withstanding 36 km/h winds, and carrying a large payload of 500 grams. This particular model is equipped with an Intel Atom
processor running a GNU/Linux based operating system and has 1 GB of RAM onboard. For navigation, the Pelican comes equipped with both a GPS and compass module, which enable the robot to operate via GPS/height waypoint commands. It has a maximum flight time of 25 minutes with a maximum speed of 50 km/h and a cruising speed of 36 km/h. Its standard weight is 750 grams, including the battery, but the particular UAV used for this system will have a weight closer to 875 grams due to various accessories. (Ascending Technologies, 2011) While a variety of different lithium polymer batteries are compatible, the Massachusetts Institute of Technology Lincoln Laboratory (MITLL) Pelican is equipped with a 5000 mAh, three-cell battery giving it a typical flight time of 18 minutes. An image of a stock pelican can be seen below in Figure 2.

![AscTec Pelican UAV](image.png)

**Figure 2: AscTec Pelican UAV**

2.2 Lithium Polymer Batteries

2.2.1 General

Due to a good shelf life and high energy density, lithium batteries are popular rechargeable batteries for many portable electronics. However, they have some dangerous characteristics that need to be considered when using them in electronics design. Because metallic lithium is very reactive and has a high energy density, the batteries can easily be over-charged/discharged or overheated, which can cause them to explode or catch on fire. When charging lithium batteries, protection circuits are used to ensure that the batteries are not charged to a dangerous state. (Luque & Hegedus, 2003)
2.2.2 Balanced Charging

Because lithium polymer batteries can be highly dangerous at certain voltage levels, they need to be monitored carefully to avoid under/overcharging and damaging the battery. This is an especially important issue for batteries with multiple cells, because if the charges of each cell are unbalanced, one cell could be at a dangerous level while the voltage level of the whole battery pack remains nominal. To avoid this issue, various balancing circuits are used to monitor and control the charge levels of individual battery cells, as outlined below. (Cao et al., 2008)

Shunting is one method of balancing that removes additional charge from cells that are at higher voltage levels than other cells in the pack, while waiting for cells at lower levels to continue charging. These circuits can use various components to control the dissipation of charge. (Cao et al., 2008)

Shuttling methods use external charge storage elements to transfer energy between cells. This enables the cells of higher levels to store charge into capacitors. Then cells with a lower charge can drain from these capacitors to even the battery levels. Unlike shunting, these methods can be used while the batteries are discharging without draining the battery. (Cao et al., 2008)

In addition to shunting and shuttling methods, there are also energy conversion methods, which use isolated energy sources and convertors to bring cells with lower voltages up to similar levels as the higher cells. These circuits using transformers, step convertors, and additional power sources are too large and expensive for UAV use. (Cao et al., 2008)

2.2.3 Battery Charging vs. Exchanging

2.2.3.1 Existing Charge Only System

At MIT in 2007, the charging of lithium polymer batteries on UAVs was investigated and a charge-only apparatus was designed. This system used an off-the-shelf balance charger that was modified to be controlled by a microcontroller instead of the manual buttons. The UAV would land, guided by a square funnel, and would make contact with the base using metal contacts on its feet and its center as shown in Figure 3. (Dale, 2007)
This system could be placed on the back of a moving ground vehicle to provide mobile recharging. Even though a fast recharge rate was used to increase the in-flight duty cycle of the UAV, it still spent twice as much time charging as it did flying. It was theorized that this could be greatly improved by exchanging the batteries instead of charging them. (Dale, 2007)

2.2.3.2 Charge vs. Exchange Economic Comparison

At the Korea Advanced Institute of Science and Technology (KAIST), an economic analysis of charging stations and exchanging stations was used to propose which method is the best for improving the UAV’s in-flight duty cycle. The analysis factored in the following criteria:

- The times for docking, charging, and flying
- The number of batteries and UAVs using the base
- The costs of the batteries, chargers, UAVs
- The of a charging station, or an exchanging station

Assuming the total number of UAVs in flight at any given time as the independent variable, for lower duty cycles it was more economical to recharge, but to maximize the duty cycle exchanging was the better option. (F., Suzuki, et al., 2010)

2.3 Existing Designs of Battery Exchanging Bases

A system capable of replacing the battery of a UAV can be broken down into the following subsystems. First is a device that can capture the UAV and account for error in landing location and orientation. Second, there must also be a quick, yet reliable, way for the battery to be attached to and removed from the UAV. Next, a mechanism or series of mechanisms is required to transfer the battery from the UAV to its charger. Finally, a system for storing the batteries while charging is needed.
2.3.1 Landing Alignment

A device capable of reliably aligning the UAV is important to ensure that mating between the base and UAV can be completed reliably. There are two approaches to aligning a UAV upon landing: passive and active. Passive systems guide the UAV into the desired position during the landing procedure without the use of actuators (Swieringa, 2010; Toksoz, 2011; Suzuki, 2011). Existing passive systems rely upon sloped sides to guide the UAV into the desired location. They have also been shown to be reliable, with one system capable of achieving an x-y position accuracy of 1 millimeter (Toksoz, 2011). An example of a passive alignment system can be seen below in Figure 4.

![Passive Alignment Apparatus](image)

Figure 4: Passive Alignment Apparatus (Toksoz, 2011)

Instead of a passive system, an active device can be used to align the UAV. This system works by allowing the UAV to land anywhere on a designated landing pad and then physically moving the UAV to the desired position and orientation. Two different approaches for an active alignment system designed by Suzuki et al. can be seen below in Figure 5. One of these systems uses bars to push the UAV to the center while the other uses a cable and pulley system to squeeze the UAV to the desired location (Suzuki, 2011).
Passive systems are mechanically simple but rely on gravity and could be less reliable if the base is not on a level surface. Active systems require actuators that consume power and add complexity and weight, but provide more accuracy and reliability.

2.3.2 Battery Mating Systems

While there are many different solutions for how to attach a battery pack to the UAV, a robust solution should be able to account for slight misalignments. Figure 6 below shows a mating system that uses two sliding rails, which worked well for a system that does not need to account for much position or orientation error (Toksoz, 2011).

In addition to rails, there are many different spring-oriented mechanical solutions to mating the battery pack to the UAV. One proposes a leaf spring device on the UAV and corresponding notches on the battery as seen in Figure 7 (Suzuki, 2011).
Another solution is to connect the battery pack to the UAV using magnets (Suzuki, 2011, Swieringa, 2010). Magnets are advantageous because they can compensate for small errors, but there is no mechanical locking mechanism between the battery pack and the UAV allowing them to become dislodged in some situations. Another disadvantage of neodymium magnets in particular is that they are very brittle and can shatter (Suzuki, 2011). The magnets could be also used as the battery terminals (Swieringa, 2010); however, this leads to potential issues with short-circuiting due to metal debris (Suzuki, 2011).

The major disadvantage with magnets is that they can never be “turned off.” Therefore, a device must be used to disengage the battery pack from the UAV. One way this can be done is to use a more powerful electromagnet mounted on the base to pull the battery pack away from the UAV (Suzuki, 2011). Another way this can be done is by mounting the UAV magnets on a servo that can be rotated, shearing the magnets apart, causing the battery pack to drop (Swieringa, 2010). While effective, this solution adds moving parts and significant weight to the UAV. An example of this system can be seen in Figure 8.

2.3.3 Battery Transfer Mechanisms

The battery transfer mechanism is needed to move the battery from the UAV to its charger and vice versa. Its requirements are largely dictated by the attachment method chosen. For example, the mating solutions mentioned above require purely linear motion for successful mating to occur, however the magnetic solutions could use other forms of motion to move the battery pack. Some devices capable of creating these kinds of motion include rack and pinion, four-bar linkages, electric linear actuators, lead/ball screws, and
scissor lifts. The systems used in previous work were scissor lifts and a rack and pinion (Suzuki 2011, Swierianga, 2010, Toksoz, 2011).

In conjunction with the rail style mating system discussed above, Toksoz’s solution used a rack and pinion transfer mechanism system. These two systems worked well together because horizontal linear motion was required. The gear rack was manufactured as part of the battery pack, meaning only the pinion had to be present on the base. One pinion was located directly below the UAV’s docked position and another in each of the battery charging locations (see Figure 9) (Toksoz, 2011). While this solution worked well in its particular system, it restricts the battery to be attached horizontally from the side only. Other solutions that loaded the batteries vertically used a scissor lift to transfer the batteries (Suzuki, 2011, Swierianga, 2010).

![Figure 9: Rack and Pinion Transfer Mechanism (Toksoz, 2011)](image)

2.3.4 Battery Storage

The battery storage and charging component plays a significant role in the system. Many existing designs rely on a battery transfer mechanism with only one degree of freedom and therefore can only move batteries from one predetermined location to the UAV (Toksoz, 2011, Suzuki, 2011, Swierianga, 2010). Therefore, the battery storage system must also move so that multiple batteries can be brought to that predetermined location. A rotating battery carousel was found to be preferred, but carousels that rotated both horizontally and vertically were used (Toksoz, 2011, Suzuki, 2011, Swierianga, 2010). Swierianga’s approach was to have a battery carousel offset from the landing pad such that one battery could be located directly under the center of the UAV (Figure 10). Suzuki’s similar approach is a centered circular battery holder and a device to push one of the batteries into the center so that it can be moved to the UAV as Figure 11 shows (Suzuki, 2011).
The final approach demonstrated by the existing systems used two vertical drums instead of one horizontal carousel. Having two drums instead of one had the advantage of being able to move the new battery into position while the old battery was still being removed (Toksoz, 2011). Their complete solution can be seen below with the two battery drums located to either side of the UAV (Figure 12).
Finally, other methods of storing the batteries were considered, but not chosen, for the existing systems. One of these is to locate the batteries linearly in either the x-y or the x-z plane (Suzuki, 2011). This design is more compact, at the cost of mechanical complexity as can be seen by the gantry system in Figure 13.

2.4 Summary

In summary, several complete systems, in addition to various procedural methods, address the problem of autonomously replacing the battery of a UAV. It can be positioned on the landing pad using
either active or passive devices. The battery can be loaded into the UAV from either the bottom or side, with many individual solutions for each. Additionally, the batteries were stored in some form of circular configuration, but the number of containers and manner in which they were transferred between the UAV and the station varied greatly.
Chapter 3: Methodology

3.1 Introduction

The main deliverable of this project is a well-documented design of an autonomous UAV refueling base station for MIT Lincoln Laboratory to build and expand upon to meet future needs. Since the design must be ready to be manufactured by Lincoln Laboratory at the end of our project, the focus is on a simple, yet robust design that is easily expandable. Using the resources available at Lincoln Laboratory, prototypes of the individual systems of the base station were manufactured, tested and redesigned through an iterative process. This chapter outlines the choices made in the design of the refueling system.

3.2 Justification

To determine the advantage of an exchange and charge system over a charge only system, the in-flight duty cycle of the UAV was calculated for a variety of situations. First, the quantity being compared, in-flight duty cycle must be determined. In-flight duty cycle is defined as the ratio of time hovering to the total amount of time in a charge cycle, as shown below.

\[ C = \frac{T_f}{T_f + T_s} \]

For \( C = \text{in-flight duty cycle} \), \( T_f = \text{flight time} \), \( T_s = \text{service time} \) (1)

Because \( T_f \) is determined by the capacity of the batteries and capability of the UAV, \( T_f \) must be found next. For a charge only system, \( T_s \) is defined very simply as

\[ T_s = T_l + T_c \]

For \( T_l = \text{time it take the robot to land and orient itself on the base} \), \( T_c = \text{charge time} \) (2)

For an exchanging system, \( T_s \) is more complex and depends on the number of batteries and the increased service time for the more complicated mechanical operations of the base. In this model, it is also chosen that the number of chargers is equivalent to the number of batteries, so that batteries do not have to wait for an available charger. First, the time spent on the base’s mechanical operation can be defined as:

\[ T_b = T_l + T_e \]

For \( T_b = \text{time to complete all UAV base operations} \), \( T_e = \text{time to exchange batteries} \) (3)

Next, the amount of time the UAV will spend waiting for a battery to be fully charged must be calculated. This value has a minimum of zero, is related to the number of batteries charging and the cycle time (flight time and base time) of the UAV, and can be seen below.
\[ T_w = \begin{cases} T_c - (N_b - 1) \cdot (T_f + T_b), & \text{for } T_c > (N_b - 1) \cdot (T_f + T_b) \\ 0, & \text{for } T_c \leq (N_b - 1) \cdot (T_f + T_b) \end{cases} \]

For \( T_w \) = the amount of time the UAV has to wait for a charged battery, \( N_b = \text{number of batteries} \) (4)

It follows that for an exchange and charge system, \( T_s \) can be defined as

\[ T_s = T_b + T_w \] (5)

In summary, this shows that if there is just one battery in our particular system, the charge only system has a small time advantage over the exchange and charge system, but that an exchanging system will have a higher duty cycle for all other cases. If enough batteries are used, the duty cycle is saturated at a maximum value for that input. This can be show in Figure 14 below where this algorithm is run for conservative figures from this base design.

![In Flight Duty Cycle](image)

**Figure 14: In-Flight Duty Cycle Saturation**

The point where the duty cycle begins to saturate shows the minimum number of batteries required for continuous UAV missions with no wait time. In this case, the saturation level is five batteries, however eight batteries were chosen for the system for a safety factor of around 1.5. See Appendix A for the MATLAB code for this model.
3.3 Design Overview

An overview of the system with the important features labeled can be found in Figure 15 below. Each aspect of the system will be outlined in detail in the following sections. The sponsor’s requirements for the system can be found in Appendix B.

Figure 15: Diagram of Complete Base Station
3.4 UAV Alignment

3.4.1 Previous Designs

The landing mechanism is the first part of the system that the UAV meets. Its purpose is to take the UAV and place it in the desired location to perform the battery transfer. Several different designs were considered for this mechanism including both passive and active systems. The passive systems all shared the same concept of having sloped sides that guided the UAV to the desired central location. The differences arose in the geometry of the sides and ranged from simply having four sloped sides to having many sloped sides, which accounted for a larger variety of landing situations. One problem with all of these passive systems is that the UAV lands on a sloped surface, which could be an issue for many UAVs that require flat landing surfaces. In the interest of making a universal system, these passive systems were not chosen.

Active alignment systems give the base more design control over how to position the UAV. One idea was to have wedges move from each of the four sides to squeeze the UAV and the other idea was to use some form of moving arms to center the UAV. While both active methods were viable, the bottom loading battery exchange mechanism that was chosen would interfere with moving wedges. While there were several different variations of moving arms, the final design had two four-bar linkages as they provided a good balance of simplicity and reliability.

3.4.2 Current Design

The landing zone has 20 inches by 20 inches of flat, open space for the UAV. The specifications from Lincoln Lab, as shown in Appendix B, call for a device capable of handling UAV position errors of ±6 inches, so this landing area is more than sufficient. While the landing pad is capable of handling large errors, the battery transfer mechanism needs accurate positioning of the UAV. Therefore, it is necessary to have the two arms that move the UAV to the center of the landing pad. Another requirement from Lincoln Lab was that the UAV’s orientation could vary by as much as ±15 degrees. These arms, however, are capable of handling any orientation except for when the feet of the UAV skid are perpendicular to the arms. This problem was of little concern as it arose well outside the customer’s specification. The arms have trouble with 45° angles because they do not induce any rotation in the UAV and the system becomes jammed. However, by moving the arms one after the other instead of simultaneously, most of these situations can be overcome. The arms themselves are a combination of two laser-cut pieces of acrylic stacked to make an L shape (see Figure 16).
Each of the arms forms the coupler of a parallelogram four-bar linkage. Each arm is powered by its own servomotor, so that the arms can be actuated one at a time to aid in reliably rotating the UAV when it is perpendicular to the arms. The ground link was printed of ABS plastic using a fused deposition modeling (FDM) machine so that the servomotor could be mounted to it and located at the correct height. This ground link also has an overhang that covers the other arm when in the fully closed position (see Figure 17) to restrain the arm from being pushed up by the UAV. Lastly, the two ground links were located offset from the center to help reduce the amount of torque required from the servos to hold the UAV in place.
To determine the torque requirements for the servos, a static analysis was performed. The worst-case scenario was determined to be when the four-bar is in the completely retracted position and the UAV is at the farthest point on the L from the coupler as shown by the orange and red arrows in Figure 18 below.

![Figure 18: Worst Case Torque on Alignment Arms](image)

The mass of the UAV, the mass of the arm, and the coefficient of friction between the UAV and the landing pad were all considered. From these calculations (see Appendix C), a torque of 67 oz-in was found. A safety factor of three was then applied and therefore each arm needed a servo capable of producing at least 200 oz-in. After researching various types of servos, one capable of producing 582 oz-in was chosen because of its reasonable price and high torque output.

One of the important considerations for the four-bar mechanisms was to determine if each would have its own servo or if the two of them would be driven by a common motor. While the required torque of 400 oz-in could be achieved by one motor with the use of a transmission, this would greatly complicate the design of the system. For a one motor system, the motor would ideally be located in the center to minimize the distance for a chain or belt to travel from the motor to each of the two actuators, but this is exactly where the rest of the system must operate. In addition, when a cost comparison was performed between two servos and one motor, it was found that the additional materials required by the motor would make it a much more expensive and complicated option. Therefore, two servos were chosen due to their simplicity and cost efficiency.
Skid coverings were added to the catching arms to provide vertical restraint to the UAV. This addition became necessary due to the vertical force required by the battery mating design. The skid coverings also had the added benefit of being able to integrate the ability to hot-swap the UAV in the future. This would provide continuous power to the UAV so that it never powers down while the battery is being replaced. This feature can be integrated into the foot covers by having electrical contacts on those covers and the tops of the skids, limiting the possibility of short-circuiting due to contacts on the bottom of the UAV as with most current designs.

3.5 Custom Skids

The custom skids were designed to be easily adapted to many UAVs and the design was based upon the Pelican’s skids. Since the custom design was based on the existing skids, it was unnecessary to create an entirely new set of skids. Instead, skid extensions were created to widen the stance of the UAV. The footprint had to be made large to allow for the battery and its case to fit completely inside the skids of the UAV. If this were not done, then when the four-bar alignment mechanisms were engaged, they would interfere with the vertical movement of the battery pack. Additionally, to improve upon the original skid design, the new skids have rounded tips to help minimize the problems when the skids are perpendicular to the alignment arms.

3.6 Custom Battery Holder

3.6.1 Previous Designs

One early design for mating the battery pack was neodymium magnets, which have large attractive forces and are small and lightweight. These magnets would be placed on both the battery pack and mounting plate, and provide some assistance in alignment. One way to remove the battery pack is to shear the magnets from each other with a rotational movement of one magnet, causing the poles to misalign. The addition of a servo or motor either on the battery pack or on the UAV would add far too much weight, exceeding one of our primary specifications. As an alternative to shearing the magnets, an electromagnet on the battery exchange system could be used to attract a magnet on the bottom of the battery pack. This electromagnet would then produce a force greater than the primary magnets when removing the battery pack. In testing, the electromagnet was far too weak when mated with another magnet; however, a large enough piece of steel as an alternative was unacceptable due to weight issues.
3.6.2 Current Design

The battery mating approach used in the final product was a non-magnetic touch latch. These latches are intended for use with cabinets and have a push-on-push-off actuation, and can be seen in Figure 19 below.

![Figure 19: Touch Latches](image)

Instead of using the stock male end of the touch latch, the male ends are integrated into the battery holder, so that additional alignment features could be added. These features, as shown in Figure 20 below, made the battery mating system capable of handling ±0.2 inches of misalignment, which is large enough that the touch latches can to be engaged when the battery pack is aligned correctly within the UAV skids. The touch latches had the advantage of a low weight (0.7 grams) and a mechanical attachment system, but had the disadvantage of being relatively tall at around 3/4”. Additionally, the touch latches required that the UAV be restrained vertically to counteract the pushing force required to actuate the latches.

![Figure 20: Custom Touch Latches with Electrical Connectors](image)

To minimize the pushing force required to make electrical connections the design shown above was chosen. This design has vertical copper plates on the battery holder and pins with springs facing horizontally towards the plates. This configuration was chosen over having vertically mounted pogo pins because it significantly decreased the travel required by the pogo pins. A gap had to be left between the charger/UAV and the battery pack in the resting position so that the battery pack could be released. If horizontal pins were used, then their travel would have to be double this gap, however, by mounting them horizontally, the spacing could be chosen instead of specified by the touch latches. More detailed diagrams of the battery contact design can be found in Appendix D.
When deciding the placement of the electrical contacts on the battery holder, careful consideration was needed so that the battery’s terminals do not short and damage the battery. A rotationally symmetrical design allows the signals always to make the correct connection whenever they are attached. Show below in Figure 21 is the contacts color coded by signal.

![Symmetrical Battery Connection Layout](image)

**Figure 21: Symmetrical Battery Connection Layout**

Because the main power and ground signals require 20 amps, they are connected through both sides of the holder to take advantage of the extra pins. On the other hand, the balancing signals use very little current and only one of the two contacts is used at a time.

### 3.7 Battery Exchange

#### 3.7.1 Previous Designs

The battery transfer system is the most complex part of the entire device. It takes the battery from the UAV, places it in a charger, and vice versa. The majority of these ideas focused on loading the battery pack from the bottom of the UAV, but horizontal loading designs had several different modifications but generally consisted of having a centered ring of batteries. This ring could be stationary and rely upon the UAV to be rotated with each battery pack having its own actuator to move it into the UAV. Similarly, the UAV could be stationary, with a rotating ring of batteries and one device to move the battery from its charger to the UAV. The problem with these horizontal loading systems is that all of the UAV alignment systems blocked horizontal loading.

The rest of the possible battery transfer systems loaded the battery from beneath the UAV. Again, this concept had several different configurations. In all cases, the batteries were held in a circular ring, but in some cases, this ring was concentric with the UAV, while in others it was offset. An offset ring would have to
rotate, so that one battery pack is directly beneath the UAV. Some sort of device would then push or pull the battery pack to move it from the charger to the UAV. The main problem with this system is the space it requires, as the footprint of the base must be increased by however much the ring is offset.

The most space efficient way to store the batteries was found to be in a ring concentric with the UAV. This ring could be either stationary or rotating. When examining the problem of storing multiple devices and transferring them to a single location, the example of a CNC machine was found. More specifically, the telescoping and rotating tool transfer device was considered to be used in conjunction with a rotating concentric ring. While this system has been proven extremely reliable in real-world applications, it was deemed too complex for this system. Most of the difficulty with this design stemmed from the use of pneumatics to achieve the telescoping movement. Having only one pneumatic device made the necessary peripheral equipment too costly and bulky to consider pneumatics while linear actuators were too expensive and large for the strokes required.

3.7.2 Current Design

3.7.2.1 Turntable Design

The final design focused on having a stationary concentric ring, for easiest battery transfer. It has the significant advantage of having the battery chargers remain stationary. Being stationary eliminated the need for complex wire management for multiple battery chargers and their accompanying wiring rotating around the base. These stationary battery packs then necessitated having a rotating turntable with a cart capable of both horizontal and vertical motion. While there are many different ways to generate these motions, it was decided to use a rack and pinion, with the pinion on the cart, to generate the horizontal motion, and a scissor lift to generate the vertical motion. This final design was ultimately chosen thanks to its potential for both speed and simplicity when compared to other approaches. In order to actuate the turntable, a stepper motor was chosen because it can perform exact, repeatable rotational movement in discrete steps.

A weakness of stepper motors is that they are susceptible to resistive torques. A high enough resistive torque will cause the motor to “skip” a step, resulting in an absolute position error. In order to account for this possibility, photo interrupters are used to localize the turntable at each of the battery stations. Therefore, the stepper motor simply moved until it reached these positions instead of solely relying upon counting steps. To minimize the force needed to rotate this device, a circular bearing was used. This aids the stepper motor, decreasing the chances of it skipping steps while moving between stations. The bearing system also provides adequate weight distribution over the turntable.
In order to minimize the time taken for the entire battery exchange process, a simple algorithm was designed to account for the abilities and limitations of the turntable assembly. The turntable was limited to approximately 360° of rotation, as a slip ring was not used and only a finite amount of extra wire can be allocated. A partial solution to this problem is the ability of the cart to move to either end of the turntable along its track. These two considerations create a scenario in which the turntable must rotate a maximum of three stations, with the worst case requiring the turntable to move three stations and the cart to move to the other end of its track.

3.7.2.2 Battery Cart Design

The cart has to be able to vertically raise and lower the battery pack while allowing for misalignment, and move horizontally from a charger to underneath the UAV. As the touch latches being used to hold the battery pack to the UAV only require a pushing motion to be actuated, the cart does not need a solid physical attachment. Instead, the cart just has to be able to hold onto the battery pack enough so that it does not move around while being transferred from the UAV to the charger. This is done by having four holes in the cart (as shown in Figure 22) and four smaller pins on the battery pack. This allows for misalignments of up to 0.15 inches, without the battery pack extending past the cart.

![Figure 22: Top View of Cart](image)

Additionally, the shape of the top of the cart required significant attention. Along with the holes to allow for misalignment, the platform was created in a cross shape. A cross-shaped hole in the landing zone was created because the UAV is aligned in 90° increments resulting in four possible orientations. The cart then had to be able to fill this hole while remaining flush with the landing pad. This lack of gaps was very important for the alignment device so that the UAV could be properly slid into the desired location without interference. Lastly, the edges of this cross were given a chamfer to help correct any slight misalignment between the cart and hole in the landing pad. This piece can be seen above in Figure 22.

As previously mentioned, a scissor lift was chosen to move the battery pack vertically a distance of 3.75 inches. This eliminated the possibility of using linear actuators, as they must be very long (approximately
12 inches) to have even close to that stroke. While not as severe, lead screws and slider cranks also require a large amount of space to operate. To keep the cart as compact as possible, a four-bar linkage and a scissor lift was found to be the best option. Because the cart must fit through the hole in the landing pad and the touch latches need linear motion, it requires a significant period of purely linear motion, essentially eliminating four-bar linkages. Therefore, a scissor lift was chosen as the actuator for vertical motion in the cart. When designing the scissor lift, it was important to ensure that the servo powered lead screw could provide enough torque to both move the battery and push the touch latches across the range of its motion.

The maximum force the scissor lift had to produce was found to be approximately 7 lbs. The torque required by the lead screw to produce this force was then calculated over the entire range of the scissor lift and the maximum was found to be around 100 oz*in. As the servo was capable of producing 582 oz*in this value was deemed acceptable. These calculations can be seen in Appendix C.

Along with vertical motion, the cart is also responsible for horizontal motion to travel from the charger to the UAV. A rack and pinion supported by a linear bearing was used instead of a lead screw, which would have been a much larger and more expensive alternative. Additionally, instead of using traditionally linear bearings, a less expensive slide for a drawer was used. This was possible because the linear bearing had to withstand a relatively small load and did not require high precision. Instead of finding a motor, the servo being used in other applications within the system was modified to be continuous. While this servo has much more torque than required (Appendix C), this decision was made to help simplify the ordering of parts and controls of the overall system. An image of the cart can be seen in Figure 23.

Figure 23: Entire Battery Cart
3.8 Lithium Polymer Charging

The most necessary subsystem of our base is the charging and balancing of the lithium polymer batteries that power the UAV. While designing a custom charging circuit would provide the most control over finding the ideal charge parameters for the batteries, many commercially available LiPo charging solutions exist that could satisfy the requirements of this project. When purchasing LiPo chargers, the following features were most important:

- Charge current of 5A (recommended current for 5AHr battery)
- Built-in balancing (to prevent dangerous charge levels in battery cells)
- Simple controls (easy to reverse engineer and autonomously control)

While all of these characteristics are common in LiPo chargers, they rarely all appear in the same system. Because of the dangers when overcharging individual cells in LiPo batteries, nearly all high current chargers had built-in balancing, but most of these chargers that unfortunately had complicated controls using a microprocessor and a LCD screen with many menus and options. Similarly many chargers with only a simple ‘start-charge’ button only charged at a rate of 1-1.5A. Fortunately, the Venom Easy Balance LiPo Charger, as shown in Figure 24 below, satisfied nearly all of the requirements.

![Venom Easy Balance LiPo Charger](image)

Figure 24: Venom Easy Balance LiPo Charger

The Venom charger charges and balances 2-4 cell batteries at 0.5-4.5 Amps and has only a button, a dial, and LED lights to control the charge. To implement this charger in an autonomous application, the operation of each of its inputs and outputs had to be analyzed with a multimeter, so that they could be
properly controlled. Because the maximum charge rate of 4.5 A will always be used, the dial to control the maximum current was glued in place at 4.5 A. The button was determined to be a low current, active low signal, which the Pololu can control as shown below in Figure 25. With only control over the dial and button, the base station can start the charge, but there is no feedback of the chargers’ state.

![Figure 25: Button Circuit Diagram](image)

The multicolor LEDs on the Venom charger indicate the state of the charger to the user. Because the color of the LED is determined by the voltage across it, the base controller can “see” the color by monitoring this voltage, as shown below in Figure 26.

![Figure 26: LED Circuit Diagram](image)

Each state contains different colors and patterns, therefore a static view of the colors cannot definitely determine the state. To determine what the pattern is, the controller will monitor the LED voltage readings three times during a one-second period. A table of the states and their associated LEDs is as follows:

<table>
<thead>
<tr>
<th>LED Pattern</th>
<th>LED Voltage(s)</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blinking Orange (2 Hz)</td>
<td>Orange (2.7V), No light (0V)</td>
<td>Battery connected; Waiting</td>
</tr>
<tr>
<td>Solid Orange</td>
<td>Orange (2.7V)</td>
<td>Battery Charging</td>
</tr>
<tr>
<td>Solid Green</td>
<td>Green (4.7V)</td>
<td>Charge Completed</td>
</tr>
<tr>
<td>Blinking Green and Red (2 Hz)</td>
<td>Green (4.7V), Red (0.6V)</td>
<td>No battery connector or battery incorrectly connected; Waiting</td>
</tr>
<tr>
<td>Blinking Red (2 Hz)</td>
<td>Red (0.6V), No light (0V)</td>
<td>Error; Charge started with no battery or battery incorrectly connected, or battery disconnected during charge</td>
</tr>
<tr>
<td>No light</td>
<td>No light (0V)</td>
<td>Error; No power to charger</td>
</tr>
</tbody>
</table>

With the inputs and outputs of the charger reverse engineered, the base station is able to maintain autonomous closed loop control of the chargers.
3.9 System Level Flow

Figure 27 below shows the general progression of system-level states while one or more UAVs are in use with the base station. The majority of the time, the base station is checking the voltage level information it receives from each of the UAVs to determine when a UAV should return for refueling. When multiple UAVs are in the system, whether a UAV is currently using the base must be monitored. Assuming the landing pad is free of obstructions, the UAV with low battery is told it is free to land. Upon landing, the base performs the refueling operation on the UAV. Finally, the UAV is sent a signal clearing it for takeoff, and the base returns to monitoring the voltage levels of all UAVs. For more detailed diagrams of the system’s flow, see Appendix F.

![System Level Flow Diagram](image-url)
3.10 Wireless Communications

The Robot Operating System (ROS) was chosen as a development environment for this system because of the messaging framework and the process management provided. The messaging framework allows users of ROS to create messages that contain any information that needs to be sent between processes. In the case of this system, the UAV, which is already running ROS onboard, publishes messages containing data on its position, orientation, and battery voltage. These messages are received by the base station, which uses this information to determine whether the UAV should return to have its battery exchanged. With multiple UAVs that may run low on battery at the same time, a UAV may send a land request message to the base station. This message is processed by the base station and a land confirmation message is sent to that UAV when the base station is able to receive it for a battery exchange.

The process management aspect of ROS allows for very easy simulation and testing of different aspects of a program. The primary use of this was during development of the communications handling. During this testing, it was necessary to be able to quickly add and remove simulated UAVs from the environment. This tested not only the ability of the base to communicate and handle multiple UAVs at once, but also correctly find and track new UAVs that have been added to the system.

The flow of the system’s states is a linear progression. Each individual state only leads to one other state, so this entire cycle can be optimized at each state individually to contribute to the optimization of the entire process. Moreover, no single state is explicitly more important to optimize than another, providing a level of flexibility during development and testing.

3.11 Program Design

The software for the system is designed to be modular, separating different subsystems to allow for easy abstraction. This abstraction creates different levels within the code, the first of which is low-level control. This controls the basic communications to and from the USB controllers for setting and obtaining values of sensors, motors, etc. The next level contains a number of files, each for one specific device such as a servo, battery charger, or stepper motor. These files contain all the functions needed to control the device, relying on the underlying low-level functions to perform necessary communications. A further level is a file containing the generalized actions of these devices. These functions include activities such as bringing the cart to one end of the track, rotating the turntable to a certain station, and centering the UAV on the landing base.
All of these action-specific functions and their underlying workings are contained in the MaestroController class, named for the Mini Maestro USB controller used for communications with the hardware.

To handle the high-level progressions of the system, the BaseStation class was created. This class contains the state machine for the system and handles the communications between the base and the UAVs. The communications between the base and UAVs is run on a separate thread, as information must be constantly gathered from all active UAVs. Requests from other UAVs in the air while one is on the base would not be noticed during the primary flow of the program if communications were not threaded.

The remainder of the base station’s functions maintains the state of the system. The state machine is the only thing that can change the state of the system, but it relies on occasional input for when to change states, such as when a UAV has low battery. This state machine calls functions from the MaestroController class to carry out necessary actions. For example, when the base is in the “centering UAV” state, the base must first call a function to acknowledge that the UAV has landed. Assuming the UAV has landed, the base must then call the CenterUAV function, which then chains calls down the levels to send control pulses to the servos.

For the complete code, see Appendix G.
Chapter 4: Results

The implementation of our systems successfully completed an exchange in slightly under five minutes, with the majority of this time taken by the scissor lift. The scissor lift took approximately 30 seconds to move from fully retracted to fully extended, thereby accounting for nearly three minutes of the exchange.

This section shows some images of the completed prototype. For more images, see Appendix E. Figure 28 below shows the complete base from multiple angles.

![Completed Overall System](image1)

Figure 28: Completed Overall System

Figure 29 below shows the landing mechanism. The landing mechanism was capable of misalignments of up to 12 inches and any amount of rotational error. Additionally, the base has dimensions of 24”x24”x12” which is smaller than the design specification. However, it had a mass of over 30 pounds, which can be attributed to the steel Unistrut frame. It also does not have the capability to hot-swap, but was designed so that it can be easily integrated at a later date.

![Landing Mechanism](image2)

Figure 29: Landing Mechanism
Figure 30: Battery Cart

Figure 30 above shows the battery cart, and Figure 31 below shows the cart with the turntable. The cart could move from the center to the edge of the turntable and the turntable could rotate to any of the eight charging stations.

Figure 31: Battery Transfer Mechanism
The charging station, as shown in Figure 32 above, proved problematic and the source of the unreliability of the system. The cause of this problem was the gold pogo pins shown below in Figure 33. Figure 34 shows the copper plates with which these pogo pins interact.
Figure 35: Battery Pack

Figure 35 shows the complete battery holder. Part of the design specification was that all additions to the UAV must sum to less than 100 grams. This included the battery pack, UAV battery mate, and foot extensions. These had a total mass of 108 grams, which was deemed acceptable by the project sponsor during flight tests.

Figure 36: Modified Battery Charger

Figure 36 above shows the COTS battery chargers after being modified to allow for feedback and control. Figure 37 below is the corresponding Pololu Controller used to control and monitor the charger. The charger system was very successful and reliable with no functional issues.

Figure 37: Pololu Controller
Chapter 5: Discussion

This project should be considered a successful development of a proof of concept for a device capable of exchanging the batteries of a UAV. It was able to successfully perform a complete battery exchange cycle but suffered from unreliability that can be directly attributed to a single component. With some additional refinement, this project can be made to meet all of the design specifications and exceed most of them. As a whole, the base physically fit within the required and goal dimensions but exceeded the maximum weight. The excessive weight of the base was due to the use of Unistrut for the frame. This frame was chosen for its prototyping capabilities and was expected to exceed the weight requirements. It is not expected that the final design would use this heavy steel frame and should therefore easily fit under the 30 pound requirement. The requirements of each subsystem will be discussed in the following chapter.

5.1 UAV Alignment Device

After construction of the device and preliminary testing, some improvements were made. The catching mechanism worked well. However, it was found that the servos did not generate as much torque as expected. It is possible that the coefficient of friction was higher than anticipated; however, the servos can run at up to 16 amps while the power supply that was used was limited to 5 amps. Therefore, the arms sometimes had problems pushing the UAV all the way to the center. To alleviate this, the software was modified so that it would close each arm twice independently. By doing this, the UAV was mostly centered and then pushed completely to the center with the second attempt. This modification also had the benefit of eliminating the device’s problem with 45 degree angles. Therefore, the alignment device achieved the ideal specifications for both displacement and rotation.

5.2 Battery Exchange Mechanism

5.2.1 Scissor Lift Mechanism

The scissor lift also benefited from several slight modifications. The scissor lift was redesigned to have a longer travel because the arms of the catcher deflected more than anticipated, increasing the needed vertical displacement. Additionally, during testing it was found that the nut of the lead screw was the weakest component of the scissor lift. This part was difficult to manufacture and install, causing delays. To counteract this for the future, one of the links was designed to be a dependable point of failure. Therefore, this easily cut
and replaced component became the weakest part of the scissor lift to protect other parts. To measure the travel of one of the arms, the potentiometer was extracted from the servo as a convenient way to use its feedback functionality. However, it was found that the potentiometer was very unreliable due to noise. To help alleviate this problem, a limit switch was mounted on the scissor lift so that it was triggered when the scissor lift was completely lowered.

An added problem of the scissor lift was its speed. In hindsight, a motor should have been used instead of a servo. The scissor lift alone accounted for nearly three of the total five minutes required to perform a battery exchange. If the speed of the scissor lift were improved, the base would be fully capable of reaching the required exchange time of three minutes.

Additionally, the horizontal movement of the scissor lift was quite slow and limited. This was due to an unresolved intermittent problem when sending large position changes from the Maestro Controller to the servomotors. To fix this problem, speed control was implemented in the software, but this resulted in a decreased overall speed. Despite purchasing roller limit switches, the approach angles were such that the scissor lift could only approach from one direction. While this lack of bi-directionality does not negatively affect the current configuration, it could pose a problem in the future.

5.2.2 Turntable Mechanism

The rotation of the turntable proved to be successful. It was feared that because the photo interrupters were so far from the battery pack being aligned, the accuracy could be off quite significantly. During testing, however, it was found that when coupled with the code to induce agitation, there were no problems with aligning the battery pack with the charger/UAV.

5.3 Battery Stations

5.3.1 Chargers

The decision to use stationary battery chargers was successful. Its best attribute is that the wire management for the battery chargers is greatly simplified, as they do not move. Furthermore, coupling these stations with modified COTS chargers reduced the cost, complexity, and danger of the entire charging system. To improve the battery charging system, code to predict the lifetime of batteries could be implemented. This extends to both the remaining time the UAV can be in flight as well as the number of times the battery has been charged.
5.3.1 Electrical Connectors

The attachment of the battery pack to the UAV/charger received a significant amount of attention, but in the end was never entirely successful. The solution was unreliable, required tight tolerances, and was very labor intensive. While the mechanical attachment system was reliable, the pogo pins used for the electrical connection added much more force than anticipated. It was found that when inserting the battery pack, one side of the battery pack would always actuate before the other. When this happened, the path of the un-actuated side would transform from linear motion to an arc-like path. This arc then intersected with both the plastic edges of the charger along with the electrical contacts. Therefore, instead of pushing in the contacts, the battery pack was trying to bend the contacts. To alleviate this problem, the software was modified such that the platform would oscillate slightly about the desired photo interrupter. Additionally, the cart for the scissor lift moved back and forth. Doing this made it so that the battery pack would naturally align with the UAV/charger. Additionally, it added dynamics to the system, which helped compress the pogo pins. It was found that the linear motion of the cart was more effective at actuating the pogo pins while the rotation of the turntable was better at fixing misalignments. Unfortunately, while these changes improved the performance, the mating was still very unreliable and still required a large amount of force to actuate the electrical contacts.

5.4 Future Recommendations

In its current condition, this project is a good first prototype and proof of concept. It has many good traits but also many this which can be improved. With these improvements, the base should be able to exceed all requirements.

For the base as a whole, the device should be transitioned toward production grade construction. First, the heavy Unistrut frame should be replaced by a lighter and more integrated frame. By doing this, the goal weight of 20 pounds without batteries can be achieved and potentially even the ideal weight of 10 pounds as well. Additionally, if a shorter stepper motor were used, then the ideal height of 12 inches can be met as the height is already very close to this value.

5.4.1 UAV Alignment Device

As it stands, the catcher is an effective system. It could be improved by adding sensor feedback and constructing it out of a stronger material such as aluminum. Sensor feedback can easily be added either through potentiometers to measure the angle of the arms, or through limit switches at the closed position.
Additionally, the catcher only has restraints over two of the four feet of the UAV, which should be increased to all four. Doing this along with using a stronger material should greatly reduce the deflection of the arms during battery pack actuation. Most significantly, the decision was made not to include hot swapping, but this feature was considered during selection of the final design. This can be implemented with two concentric rings on the top of the feet of the UAV: one for power and one for ground. These will then contact similar rings on the bottom of the foot restraints to provide power during exchange. Everything mentioned above should be considered as minor alterations and not a shift in concept. It is felt that the decision to use an active alignment device was the right one and should not be changed.

5.4.2 Battery Exchange Mechanism

Similarly, the scissor lift needs slight modifications. The scissor lift should have a motor with a gearbox instead of a modified servo. This should significantly speed up the scissor lift and therefore the entire battery exchange. Additionally, by having an independent potentiometer instead of extracting it from the servo, the noise problems experienced should be eliminated, resulting in more accurate position control. In the more distant future, the addition of a second scissor lift on the same cart should be considered. Doing this would drastically decrease the amount of time the UAV spends on the base because a charged battery can be preloaded to one of the scissor lifts, which should bring the exchange time to under the ideal time of one minute. The first scissor lift would remove the used battery and move to the lower position. The cart would then have to move slightly to align the second scissor lift in order to place the preloaded new battery into the UAV. The rest of the charging process can then be completed while the UAV is airborne. The important aspect of the scissor lift that should not be changed is that it has a flat top. If the scissor lift did not have a flat top, then it would cause problems when aligning the UAV, as it has to fill the gap in the landing pad.

The horizontal movement of the cart should be very slightly modified. Instead of using limit switches, photo interrupters could be used to determine where the cart should stop so that the cart can approach from either direction without issue.

5.4.3 Battery Electrical Connections and Mating

The mating between the battery pack and the UAV/charger is the only part of this system that could benefit from a complete redesign. While the mechanical attachment system is effective, it allows the battery pack to swing, which has the potential to cause problems for the control of the UAV. If swinging is not a problem, then using touch latches is an inexpensive and simple way to have a mechanical connection. If the COTS male and female components of the touch latches are used with a new form of electrical contact, then
tight tolerances and a large amount of labor can be avoided. The pogo pins should not be used, as they required too much force, regardless of orientation. Instead of these pins, it is recommended that magnets be reconsidered. Magnets were quickly ruled out because there was no easy way to detach them. However, if small magnets were used only for the electrical connections instead of large magnets for the mechanical and electrical connection such as a design based upon the MacBook power cord, then this may work. Additionally, while it was discovered too late to evaluate for this prototype, switchable permanent magnets should be considered as well.

On a higher level, a solution is needed for the electrical contacts that requires much less force than the current configuration. In the end, the electrical connectors caused the majority of the problems and unreliability in the system. If a suitable way to make the electrical connection is found and the other recommendations are executed, then this prototype has the potential to be a fast and reliable way to autonomously exchange the batteries of a variety of different UAVs.

5.4.4 Software

The overall system could be improved through the inclusion of a graphical user interface (GUI). This GUI could provide useful information to an operator such as the charge state of each battery, the remaining lifetime of each battery, and information from the UAV. The most basic version of this GUI could include information about whether or not there is a UAV currently being serviced, the UAVs currently waiting to land, if any, and the state of charge of the batteries. This can be expanded by adding functionality base-side to calculate such figures as remaining battery time both while charging and while being used on a UAV. Many different resources exist that may be used to determine an adequate model for each of these situations. Another addition to the GUI would track the number of cycles for each battery, notifying an attendant when a battery needs to be removed from the system and exchanged to retain maximum efficiency. The GUI may also display any sensor data from the UAV itself. Additional possibilities include video for surveillance, sensor readings during data collection, and general location and orientation information for navigation purposes.

These additions may also be added without the use of a GUI, though to less effect in some cases. While a video feed may be advantageous if displayed to an onlooker, it is not useful in its real time form when there is no one to watch. This introduces the idea to have a log file that is generated during normal operation of the base station. This file may keep track of many things, including the number of cycles each battery has gone through, what charging station is empty, etc. The base station can read from this file at startup, keeping persistence between uses. This alleviates a problem of possibly running batteries for too many cycles, which could happen if this information is not kept and updated regularly.
Chapter 6: Conclusion

This project was a success when considering that the intended result was a prototype and proof of concept. After examining the results, the core weakness of the system was found to be the method chosen for creating an electrical connection. If this method is replaced by a better system, then the entire system becomes more reliable and meets or exceeds all design specifications. Additionally, by replacing the motor of the scissor lift and including a secondary scissor lift to allow for preloading of a charged battery, then this solution has the potential to have the UAV out of the air for a negligible period of time relative to its flight time.

Having identified the aforementioned areas for improvement, it is important to recognize the successes of this project as it already meets or exceeds most specifications. It successfully implemented an active planar alignment system that can easily accommodate various UAVs while being able to accept position errors of up to 12 inches and any amount of rotational error. Furthermore, while the electrical contacts were not successful, the mechanical solution for attaching the battery pack to the charger and UAV was inexpensive, while providing a solid connection that coped well with misalignments. Moreover, it did this while approximately meeting the desired weight limit of 100g added to the UAV. From a software perspective, the decision to induce slight movement both linearly and rotationally when moving the battery pack was a simplistic solution to reliably fix additional misalignment. Through testing, any unreliability not attributed to the electrical contacts was identified and addressed. Therefore, this project has demonstrated its potential for being a fast, reliable system that can perform a complete battery exchange.

With the completion of this project, MIT Lincoln Laboratory was provided with a successful prototype that has the potential to be implemented into a mobile platform to support a variety of hovering UAVs with minimal modifications. Additionally, the Lab is being provided with this report and all accompanying hardware, software, and models which document the research, procedures taken, and lessons learned from this project, which can be utilized to both refine this base as well as implement any future work in this area of research.
References


AscTec Pelican. Retrieved from https://picasaweb.google.com/asctec/AscTecPelican#5452579016283220050


%Appendix A: Justification MATLAB Code

function [ ] = ChargevsSwap( Tf, Tc, Tl, Te, Nb )
%ChargevsSwap Plots in-flight duty cycle based on number of batteries
% Tf = flight time, Tc = charge time, Tl = landing time, Te = battery exchange time, Nb = array of numbers of batteries to plot
% Assuming batteries start spent and start charging when first UAV takes off

Tb = Tl+Te; %Tbase = Tlanding + Texchange
Tw = (Tc)-(Nb-1)*(Tf+Tb); %Twaiting = (Tcharging + Texchange) - Nbatteries (Tflight + Tbase)
Tw(Tw < 0) = 0; %Twaiting cannot be less than 0
Ts = Tb+Tw; %Tservice = Tlanding + Texchange + Twaiting
Cc = Tf/(Tf+Tl+Tc); %Duty cycle of charge only = Tflight / (Tflight + Tland + Tcharge)
Cs = Tf/(Tf+Ts);
test = Cc * (Tf+Tl+Tc)/(Tf+Tl+Te); % show mult difference

figure1 = figure;
axes1 = axes('Parent',figure1,'XTick',[1 2 3 4 5 6 7 8 9 10]);
ylim(axes1,[0 1]);
box(axes1,'on');
hold(axes1,'all');
plot(Nb,Cs,'Parent',axes1,'MarkerFaceColor',[0 0 1],...
     'MarkerEdgeColor',[0 0 1],...
     'Marker','o',...
     'LineWidth',2,...
     'Color',[0 0 1],...
     'DisplayName','Swap and Recharge System');
plot(1, Cc,'Parent',axes1,'MarkerFaceColor',[0 .7 0],...
     'MarkerEdgeColor',[0 .7 0],...
     'Marker','o',...
     'LineStyle','none',...
     'DisplayName','Charge Only System');
xlabel('Number of Batteries','FontSize',12);
ylabel('% of Cycle Time in Flight','FontSize',12);
title('In Flight Duty Cycle','FontSize',14);
legend1 = legend(axes1,'show');
set(legend1,'Location','SouthEast');
end
# Appendix B: Design Requirements

<table>
<thead>
<tr>
<th>Category</th>
<th>Specification</th>
<th>Requirement</th>
<th>Goal</th>
<th>Ideal</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pelican landing pose</td>
<td>delta X-Y</td>
<td>± 6&quot;</td>
<td>± 12&quot;</td>
<td>± 12&quot;</td>
<td>Nominal from 0 of Pelican's delta - Pelican can land if displaced by amount</td>
</tr>
<tr>
<td></td>
<td>delta Yaw</td>
<td>± 30°</td>
<td>± 45°</td>
<td>± 180°</td>
<td>If additional structure added to Pelican</td>
</tr>
<tr>
<td></td>
<td>delta Pitch-Roll</td>
<td>± 15°</td>
<td>± 30°</td>
<td>± 45°</td>
<td></td>
</tr>
<tr>
<td></td>
<td>overall X-Y</td>
<td>&lt; 36&quot;</td>
<td>&lt; 30&quot;</td>
<td>&lt; 24&quot;</td>
<td>May exclude control computer if necessary</td>
</tr>
<tr>
<td></td>
<td>overall Z</td>
<td>&lt; 24&quot;</td>
<td>&lt; 18&quot;</td>
<td>&lt; 12&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ground mass</td>
<td>&lt; 30 lbs</td>
<td>&lt; 20 lbs</td>
<td>&lt; 10 lbs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>added mass to UAV</td>
<td>&lt; 100 g</td>
<td>&lt; 50 g</td>
<td>&lt; 0 g</td>
<td>Does not include batteries</td>
</tr>
<tr>
<td>device size</td>
<td>battery capacity</td>
<td>5000 mAh</td>
<td>≤ 5000 mAh</td>
<td>≤ 8000 mAh</td>
<td>More interested in circuitry that can handle this, not mechanical</td>
</tr>
<tr>
<td></td>
<td>Pelican discharge rate</td>
<td>75% battery discharge over 18 min</td>
<td></td>
<td></td>
<td>This is what we will test to</td>
</tr>
<tr>
<td></td>
<td>battery exchange time</td>
<td>&lt; 3 min</td>
<td>&lt; 2 min</td>
<td>&lt; 1 min</td>
<td>This factors into persistence</td>
</tr>
<tr>
<td></td>
<td>hot swap</td>
<td>hot swap capability required</td>
<td></td>
<td></td>
<td>Okay for future expansion</td>
</tr>
<tr>
<td></td>
<td>persistence</td>
<td>persistent flight</td>
<td></td>
<td></td>
<td>Can extend battery exchange time to accomplish this specification</td>
</tr>
<tr>
<td>battery charge &amp; exchange</td>
<td>balance functionality</td>
<td>required while charging</td>
<td></td>
<td></td>
<td>Needed to preserve battery quality, also allows for faster charge rates</td>
</tr>
<tr>
<td></td>
<td>base power consumption</td>
<td>&lt; 200 W</td>
<td>&lt; 180 W</td>
<td>&lt; 150 W</td>
<td>Includes control computer</td>
</tr>
<tr>
<td></td>
<td>power input</td>
<td>DC for custom circuits, AC/DC for COTS</td>
<td></td>
<td></td>
<td>Power conversion will be provided</td>
</tr>
<tr>
<td>software specs</td>
<td>communication</td>
<td>ROS message protocol via Ethernet or Wi-Fi</td>
<td></td>
<td></td>
<td>Lincoln to construct message types</td>
</tr>
<tr>
<td></td>
<td>logic</td>
<td>Initiate landing request given battery voltage</td>
<td></td>
<td></td>
<td>Demonstrates control capability of base</td>
</tr>
</tbody>
</table>
Appendix C: Torque Calculations

Catcher Servo Requirements

\[ C_L = 7.75 \text{ in} = 0.197 \text{ m} \] length of the arm of the four-bar

\[ \text{Servo Torque} = 582 \text{ oz-in} \] maximum servo torque

\[ \text{UAV Weight} = 1000 \text{ gm} \] fully loaded mass of UAV

\[ \mu_F = 0.25 \] coefficient of friction (found acrylic on acrylic to be 0.21)

\[ L_Q = 23 \text{ in} \] length of UAV lever arm

\[ \text{gap} = 6 \text{ in} \] distance between four-bar links

\[ \text{offset} = 0.5 \text{ in} \] distance from edge to first four-bar link

Positions of Forces relative to CG:

\[ Q := L_Q - \text{offset} = 22.5 \text{ in} \] distance from UAV to farthest link

\[ Y_2 := \text{gap} = 6 \text{ in} \]

\[ F_Q := \mu_F \cdot \text{UAV Weight} = 0.551 \text{ lb} \] force required to push UAV
Stress Analysis

Placing the UAV so it acts in the Y direction:

Examining the coupler link:

\[ \Sigma M_1 = 0 = F_2 Y_2 - FQ Y \]
\[ \Sigma F_y = 0 = F_{1y} + F_{2y} - FQ \]
\[ \Sigma F_x = 0 = F_{1x} + F_{2x} \text{ so } F_{1x} = -F_{2x} \text{ to be used later} \]

\[ F_{2y} = \frac{FQ}{Y_2} = 2.067 \text{ lb} \]
force on link 2 in Y direction

\[ F_{1y} = F_Q - F_{2y} = -1.516 \text{ lb} \]
force on link 1 in Y direction
Examining the rocker link:

There is no motor on the rocker and it is free to rotate. Therefore, F2x must be selected such that it cancels out the moment created by F2y so that the sum of the moments is 0. The reactant forces in the ground link are equal and opposite to F2x and F2y and are neglected in these calculations.

\[ \Sigma M = 0 = \ (F2y \cdot \cos(\alpha) - F2x \cdot \cos(90-\alpha)) \cdot CL = \ (F2y \cdot \cos(\alpha) - F2x \cdot \sin(\alpha)) \cdot CL \]

\[ F_{2x}(\alpha) := \frac{\cos(\alpha)}{\sin(\alpha)} \cdot F_{2y} \quad \text{force on link 2 in x direction based on angle} \]

From the carrier link, F1x = -F2x

\[ F_{1x}(\alpha) := -F_{2x}(\alpha) \quad \text{force on link 1 in x direction based on angle} \]

Examining the crank link:

\[ T(\alpha) := C_L \cdot \cos(\alpha) \cdot F_{1y} - C_L \cdot \sin(\alpha) \cdot F_{1x}(\alpha) \quad \text{torque required by motor to move UAV} \]

\[ T(10\text{deg}) = 67.305 \text{- oz-in} \]

\[ F_{2x}(10\text{deg}) = 11.722 \text{- lb} \]

\[ F_{1x}(10\text{deg}) = -11.722 \text{- lb} \]

Range = 9, 10.. 81

As can be seen from the graph above, the maximum required torque occurs when the four-bar linkage is fully collapsed with an \( \alpha \) of approximately 10 degrees.
Placing the UAV so it Acts in the X direction:

Stress Analysis

Examining the coupler link:

\[ \Sigma M_1 = 0 = FQ*Q + Fy2*Y2 \]
\[ \Sigma F_y = 0 = F1y + F2y \]
\[ \Sigma F_x = 0 = F1x + F2x - FQ \text{ this will be used later} \]

\[ F_{ax} = FQ \frac{Q}{Y2} = 2.067 \text{ lb} \] force on link 2 in Y direction

\[ F_{ay} = F_{2y} = 2.067 \text{ lb} \] force on link 1 in Y direction
Examining the rocker link:

There is no motor on the rocker and it is free to rotate. Therefore, F2x must be selected such that it cancels out the moment created by F2y so that the sum of the moments is 0. The reactant forces in the ground link are equal and opposite to F2x and F2y and are neglected in these calculations.

\[ \Sigma M = 0 = (F2y \cdot \cos(\alpha) - F2x \cdot \cos(90-\alpha)) \cdot CL = (F2y \cdot \cos(\alpha) - F2x \cdot \sin(\alpha)) \cdot CL \]

\[ \frac{F_{2x}(\alpha)}{\sin(\alpha)} = \frac{\cos(\alpha)}{F2y} \text{ force on link 2 in } x \text{ direction based on angle} \]

From the carrier link, F1x = -F2x

\[ F_{1x}(\alpha) = -F_{2x}(\alpha) + F_Q \text{ force on link 1 in } x \text{ direction based on angle} \]

Examining the crank link:

\[ T(\alpha) := C_L \cdot \cos(\alpha) \cdot F_{1y} - C_L \cdot \sin(\alpha) \cdot F_{1x}(\alpha) \text{ torque required by motor to move UAV} \]

\[ T(10\text{deg}) = 492.92\text{-oz-in} \]
\[ F_{2x}(10\text{deg}) = 11.72\text{-lb} \]
\[ F_{1x}(10\text{deg}) = -11.17\text{-lb} \]

Range := 9, 10 .. 81

As can be seen from the graph above, the maximum required torque occurs when the four-bar linkage is fully collapsed with an \( \alpha \) of approximately 10 degrees.
Scissor Lift Requirements

Load of Scissor Lift:

Battery Pack:
- Battery := 1 lb
- Battery\_Case := 79 gm

battery\_pack := Battery + Battery\_Case = 1.174 lb

Pushing the Latches:
- latches := 2
- Latches\_resistance := latches \cdot 1.5 lb = 3 lb

Pushing the Pins:
- pins := 8
- \mu_{\text{pin}} := 1.21 \text{ copper on copper}
- Contacts\_resistance := 120 gm \cdot pins \cdot \mu_{\text{pin}} = 2.561 lb

resistance := Contacts\_resistance + Latches\_resistance = 5.561 lb

Calculate Torque Required for Lead Screw:

\text{Link\_Length} := 6 in \quad \text{scissor} := 0.33 lb \quad \text{screw\_diameter} := 0.19 in \quad \mu_{\text{steel}} := 0.25 \quad \lambda := \frac{1}{24 in}

Weight := battery\_pack + resistance + scissor = 113.041 oz

\text{F}_{\text{lead}}(\text{theta}) := \frac{\text{Weight}}{2 \tan(\text{theta} \text{ deg})}

\text{T}_{\text{lead}}(\text{theta}) := \frac{\text{F}_{\text{lead}}(\text{theta}) \cdot \text{screw\_diameter}}{2} \left( \frac{1 + \pi \cdot \mu_{\text{steel}} \cdot \text{screw\_diameter}}{\pi \cdot \text{screw\_diameter} - \mu_{\text{steel}}^2} \right)

\text{Range} := 0 .. 20
\[ T_{\text{lead}}(\text{Range}) = \frac{100.124}{\text{in-oz}} \]

As can be seen above, the torque of the servo is more than sufficient to lift the battery pack, however, this does not take into account the friction in the scissor lift and therefore accounts for the high safety factor with this motor.

Rack and Pinion

\[
\begin{align*}
\text{pinion}_{\text{radius}} & := 0.5\text{in} \\
\text{servo} & := 1\text{lb} \\
\text{cart}_{\text{parts}} & := 2\text{lb} \\
\text{cart} & := \text{servo} + \text{cart}_{\text{parts}} = 3\text{lb} \\
\text{cart}_{\text{weight}} & := \text{batterypack} + \text{cart} = 4.174\text{lb} \\
\mu_{\text{frail}} & := 0.5 \\
\text{Cart}_{\text{torque}} & := \text{cart}_{\text{weight}} \times \text{pinion}_{\text{radius}} \times \mu_{\text{frail}} = 16.697\text{ oz-in} \\
\text{Cart}_{\text{SafetyFactor}} & := 2 \\
\text{Cart}_{\text{totalTorque}} & := \text{Cart}_{\text{torque}} \times \text{Cart}_{\text{SafetyFactor}} = 33.393\text{ oz-in}
\end{align*}
\]
Turntable

\[ \tau_{\text{stepper}} := 220 \text{oz}\cdot\text{in} \]

\[ \text{table\_radius} := \frac{20.5}{2} \text{in} \]

\[ \text{table\_weight} := 2.5 \text{lb} \]

\[ \mu_{\text{table}} := 0.15 \]

\[ \tau_{\text{table}} := \text{table\_radius} \left( \text{table\_weight} + \text{cart\_weight} \right) \mu_{\text{table}} = 164.184 \text{ oz}\cdot\text{in} \]

\[ \text{Table\_Safety} := \frac{\tau_{\text{stepper}}}{\tau_{\text{table}}} = 1.34 \]
Appendix D: Custom Battery Connectors

Transparent Female Connector to Show Electrical Contacts:

Male Connector with no Contacts or Wires:
Transparent Male Connector with no Contacts or Wires:

Transparent Male Connector with no Wires:
Appendix E: Additional Images

Landing Mechanism Open:

Landing Mechanism Closed:
Aligned Foot Extension:

Landing Mechanism Servo:
Battery Charging Station:
Photo Interrupters:
Scissor Lift Lead Screw:
Turntable:

Battery Charger:
Battery:
Appendix F: System State Flow Chart

UAV State Machine:

1. **UAV In Waiting Queue?**
   - Yes: Wait for Landing Requests from UAVs
   - No: Voltage Low?

2. **Request Received?**
   - Yes: Send Landing Confirmation
   - No: Place UAV In Waiting Queue

3. **Query Master Node for New UAV Nodes**
   - No: New Node?
     - Yes: Add to Subscription List
     - No: Landing Sequence
   - Yes: New Node?
     - Yes: Add to Subscription List
     - No: Landing Sequence

Flow Chart:
- Receive Voltage Information from UAVs
- Send Landing Confirmation
- Landing Sequence
Base Station State Machine:

Identify Fully-Charged Battery

Check Fully-Charged List

Any Items in List? No

Yes

Take First Entry

Query Chargers to Find Charged Battery

Found Charged Battery? No

Yes

Close Arms One at a Time

Open Arms

Close Arms One at a Time

Center UAV

Remove Spent Battery

Raise Scissor Lift to UAV

Lower Scissor Lift to Bottom Position

Rock Turntable / Shake Cart

Charge Spent Battery

Rotate to Open Station

Move Cart to Far End of Track

Raise Scissor Lift to Charger Height

Rock Turntable / Shake Cart

Retrieve Fully-Charged Battery

Rotate to Fully-Charged Station

Raise Scissor Lift to Charger Height

Lower Scissor Lift to Bottom Position

Rock Turntable / Shake Cart

Place Battery in UAV

Move Cart to Center of Track

Raise Scissor Lift to Charger Height

Move Scissor Lift to Resting Position

Rock Turntable / Shake Cart

Release UAV

Open Arms

Signal for Takeoff

Return to Idle State

Check Fully-Charged List

Any Items in List? No

Yes

Take First Entry
Appendix G: Code

/*
 * BaseStation.hpp
 * Created on: Aug 26, 2011
 * Author: ra23501
 */

#ifndef BASESTATION_HPP_
#define BASESTATION_HPP_

#include <ros/ros.h>
#include <ros/console.h>
#include <boost/thread/thread.hpp>
#include <geometry_msgs/Vector3.h>
#include <geometry_msgs/Quaternion.h>
#include <wpi_msgs/LandRequest.h>
#include <wpi_msgs/LandConfirm.h>
#include <wpi_msgs/RobotState.h>

#include "BatteryPack.hpp"
#include "MaestroController.hpp"

// define number of batteries in system
#ifndef NUM_BATTERIES
#define NUM_BATTERIES 8
#endif

// define pi
#ifndef PI
#define PI 3.14159265358979323846
#endif

// define epsilon
#define EPSILON 1E-6

// macros
#ifndef MIN
#define MIN(A,B) (((A)<(B))?(A):(B))
#endif

#ifndef MAX
#define MAX(A,B) (((A)>(B))?(A):(B))
#endif

#ifndef MINMAX
#define MINMAX(A,B,C) MIN((B),MAX((A),(C)))
#endif

#ifndef NORMALIZE
#define NORMALIZE(Z) atan2(sin(Z),cos(Z))
#endif

// enums
enum BaseState
{
  IDLE, LANDING, CENTERING, REMOVING_BATTERY, REPLACING_BATTERY, LAUNCHING
};

using namespace std;

class BaseStation
{
public:
  BaseStation();
  ~BaseStation();
private:
    // callbacks
    void callbackStateUpdate(const boost::shared_ptr<wpi_msgs::RobotState const> &robot_state_msg);
    void callbackLandRequestMsg(const boost::shared_ptr<wpi_msgs::LandRequest const> &land_request_msg);
    void callbackLandConfirmMsg(const boost::shared_ptr<wpi_msgs::LandConfirm const> &land_confirm_msg);

    // init functions
    void initParams(void);
    void initSubscribers(void);
    void initBatteries(void);

    // robot states
    void idleState(void);
    void landingState(void);
    void centeringState(void);
    void removingBatteryState(void);
    void replacingBatteryState(void);
    void launchingState(void);

    // base station thread
    void startBaseThread(void);
    void baseThread(void);
    void stopBaseThread(void);

    // listener thread
    void startListenThread(void);
    void listenThread(void);
    void stopListenThread(void);

    // thread helper functions
    void findNodes(void);
    void addNode(string name);
    float findRemainingTime(string name);

    // ros objects
    ros::NodeHandle nh_;  
    ros::NodeHandle private_nh_;  

    // ros param
    string uid_;  
    double base_thread_rate_;  
    double max_battery_voltage_;  
    double return_battery_voltage_;  
    double min_battery_voltage_;  

    // ros publishers
    std::vector<ros::Publisher> land_request_pub_;  
    std::vector<ros::Publisher> land_confirm_pub_;  

    // ros subscribers
    ros::Subscriber robot_state_sub_;  
    ros::Subscriber land_request_sub_;  
    ros::Subscriber land_confirm_sub_;  

    // threads
    boost::shared_ptr<boost::thread> robot_state_sub_thread_;  
    boost::shared_ptr<boost::thread> base_thread_;  
    boost::shared_ptr<boost::thread> listen_thread_;  
    boost::mutex mutex_;  

    // general containers
    wpi_msgs::RobotState robot_state_;  
    std::vector<std::string> all_UAVs_;  
    std::vector<std::string> waiting_UAVs_;  
    BaseState base_state_;  
    std::string current_UAV_;  
    MaestroController controller;  
    BatteryPack* allBatteries[NUM_BATTERIES];  
    std::vector<BatteryPack*> readyBatteries;  
    char openStation;
// volatile bools
volatile bool received_robot_update;
volatile bool received_land_request;
volatile bool received_positive_land_confirm;
volatile bool received_negative_land_confirm;
volatile bool UAV_landing;
};
#endif /* BASESTATION_HPP_ */

/*
 * MaestroController.hpp
 */
/*
 * Portions of this code including this header file and source files prefixed
 * with MaestroController_ contain code from at least one of libusb-1.0 and
 * the Pololu USB SDK. libusb-1.0 is licensed under terms of the LGPL found at
 * http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html and the Pololu SDK
 * is licensed under their license, found in the file PololuLicense.txt and at
 */
#ifndef MAESTRO_CONTROLLER_HPP
#define MAESTRO_CONTROLLER_HPP
#include <libusb.h>
#include <ros/ros.h>
// defines
#define VENDOR_POLOLU 0x1ffb
#define MICRO_MAESTRO_STACK_SIZE 32
#define MICRO_MAESTRO_CALL_STACK_SIZE 10
#define MINI_MAESTRO_STACK_SIZE 126
#define MINI_MAESTRO_CALL_STACK_SIZE 126
// for Maestro Mini 24
#define SERVO_COUNT 24
// stall is checked at 2Hz
#define STALL_OUT 2
#define NUM_BATTERIES 8
// logic high/low, also directions
#define STEPPER_CCW 2000*4
#define STEPPER_CW 1000*4
#define TRUE 2000
#define FALSE 1000
#define ON 2000*4
#define OFF 1000*4
#define ON_INP 1023
#define OFF_INP 0
// speeds for cart rack servo
#define CART_RIGHT 1600*4
#define CART_LEFT 1400*4
// LED color cutoffs
#define GREEN 770
#define ORANGE 255
// macros
#ifndef VTOSIG
#define VTOSIG(A) (A * 1023 / 5)
#endif
#ifndef SIGTOV
#define SIGTOV(A) (A * 5 / 1023)
#endif

65
#ifndef ISGREEN
#define ISGREEN(A) (A >= GREEN)
#endif

#ifndef ISORANGE
#define ISORANGE(A) (A >= ORANGE && A < GREEN)
#endif

#ifndef ISRED
#define ISRED(A) (A < ORANGE)
#endif

// enums
// product codes to validate USB devices
enum MaestroProduct
{
    MAESTRO_PROD_MICRO = 0x0089, MAESTRO_PROD_MINI_12 = 0x008A, MAESTRO_PROD_MINI_18 = 0x008B,
    MAESTRO_PROD_MINI_24 = 0x008C
};

// serial numbers for devices to discern them from each other
enum MaestroSerial
{
    MICRO_SERIAL = 7021, PRIMARY_SERIAL = 17565, SECONDARY_SERIAL = 17571
};

/*enum ScissorPotValues
{
    SCISSOR_REST_POT = ,
    SCISSOR_UAV_POT = ,
    SCISSOR_CHARGE_POT = ,
    SCISSOR_DOWN_POT = 142
};/*/ 

// These are the values to put in to bRequest when making a setup packet
// for a control transfer to the Maestro. See the comments and code in Usc.cs
// for more information about what these requests do and the format of the
// setup packet.
enum Request
{
    REQUEST_GET_PARAMETER = 0x81, REQUEST_SET_PARAMETER = 0x82, REQUEST_GET_VARIABLES = 0x83,
    REQUEST_SET_SERVO_VARIABLE = 0x84, REQUEST_SET_TARGET = 0x85,
    REQUEST_CLEAR_ERRORS = 0x86,
    REQUEST_GET_SERVO_SETTINGS = 0x87,
    REQUEST_GET_STACK = 0x88, REQUEST_GET_CALL_STACK = 0x89, REQUEST_SET_PWM = 0x8A,
    REQUEST_REINITIALIZE = 0x90, REQUEST_ERASE_SCRIPT = 0xA0,
    REQUEST_WRITE_SCRIPT = 0xA1,
    REQUEST_SET_SCRIPT_DONE = 0xA2, // value.low.b is 0 for go, 1 for stop, 2 for single-step
    REQUEST_RESTART_SCRIPT_AT_SUBROUTINE = 0xA3, REQUEST_RESTART_SCRIPT_AT_SUBROUTINE_WITH_PARAMETER = 0xA4,
    REQUEST_RESTART_SCRIPT = 0xA5, REQUEST_START_BOOTLOADER = 0xFF
};

// channels for the Maestro Micro controller
enum MicroChannels
{
    MICRO_TEST_SERVO = 0, MICRO_SECOND_SERVO = 1, MICRO_LIMIT = 4, MICRO_PHOTO = 5
};

// channels for the Maestro Mini 24 primary (servo) controller
enum PrimaryMiniChannels
{
    ARM_SERVO_0 = 0, ARM_SERVO_1 = 1, CART_0_SERVO = 2, CART_0_MOTOR = 3, CART_1_SERVO = 4, SCISSOR_POT = 5,
    CART_END_0 = 6, CART_END_1 = 7, CART_MID = 8, STEPPER_DIRECTION = 9, STEPPER_STEPS = 10, STEPPER_HOME = 15,
    BATTERY_STATION_0 = 11, BATTERY_STATION_1 = 12, BATTERY_STATION_2 = 13, BATTERY_STATION_3 = 14,
    BATTERY_STATION_4 = 15, BATTERY_STATION_5 = 16, BATTERY_STATION_6 = 17, BATTERY_STATION_7 = 18,
    ARM_SERVO_0_CURRENT = 19, ARM_SERVO_1_CURRENT = 20, CART_0_SERVO_CURRENT = 21, CART_1_SERVO_CURRENT = 22
};
// channels for the Maestro Mini 24 secondary (battery) controller
enum SecondaryMiniChannels
{
    BATTERY_START_RELAY_0 = 0, BATTERY_START_RELAY_1 = 1, BATTERY_START_RELAY_2 = 2, BATTERY_START_RELAY_3 = 3,
    BATTERY_START_RELAY_4 = 4, BATTERY_START_RELAY_5 = 5, BATTERY_START_RELAY_6 = 6, BATTERY_START_RELAY_7 = 7,
    BATTERY_LED_0 = 8, BATTERY_LED_1 = 9, BATTERY_LED_2 = 10, BATTERY_LED_3 = 11, BATTERY_LED_4 = 12,
    BATTERY_LED_5 = 13,
    BATTERY_LED_6 = 14, BATTERY_LED_7 = 15, BATTERY_PRESENT_0 = 16, BATTERY_PRESENT_1 = 17, BATTERY_PRESENT_2 = 18,
    BATTERY_PRESENT_3 = 19, BATTERY_PRESENT_4 = 20, BATTERY_PRESENT_5 = 21, BATTERY_PRESENT_6 = 22,
    BATTERY_PRESENT_7 = 23
};

// hard limits for the servos, by name
enum ServoLimits
{
    GWS_MAX = 2480 * 4, GWS_MIN = 592 * 4
};

// some commonly-used servo values
enum ServoValues
{
    SCISSOR_REST = 1295 * 4,
    SCISSOR_UAV = 1225 * 4,
    SCISSOR_CHARGE = 1605 * 4,
    SCISSOR_DOWN = 1520 * 4,
    SCISSOR_MAX_UP = 900 * 4, SCISSOR_MAX_DOWN = 2100 * 4
    /*
    GWS_HOME = 2480 * 4, SCISSOR_REST = 1545 * 4, // 87.5
    SCISSOR_UAV = 1500 * 4, // 85.0
    SCISSOR_CHARGE = 1850 * 4, // 104.5
    SCISSOR_DOWN = 1975 * 4, // 108.5
    SCISSOR_MAX_UP = 1200 * 4, SCISSOR_MAX_DOWN = 2300 * 4
    */

};

// Represents the current status of a channel.
struct ServoStatus
{
    // The position in units of quarter-microseconds.
    unsigned short position;

    // The target position in units of quarter-microseconds.
    unsigned short target;

    // The speed limit.
    unsigned short speed;

    // The acceleration limit.
    unsigned char acceleration;
};
typedef struct ServoStatus ServoStatus;

// Represents the variables that can be read from
// a Micro Maestro or Mini Maestro using REQUEST_GET_VARIABLES,
// excluding channel information, the stack, and the call stack.
struct MaestroVariables
{
    // The number of values on the data stack (0-32). A value of 0 means the stack is empty.
    unsigned char stackPointer;

    // The number of return locations on the call stack (0-10). A value of 0 means the stack is empty.
    unsigned char callStackPointer;

    // The error register. Each bit stands for a different error (see uscError).
    // If the bit is one, then it means that error occurred some time since the last
    // GET_ERRORS serial command or CLEAR_ERRORS USB command.
unsigned short errors;

    // The address (in bytes) of the next bytecode instruction that will be executed.
unsigned short programCounter;

    // 0 = script is running.
    // 1 = script is done.
    // 2 = script will be done as soon as it executes one more instruction
    // (used to implement step-through debugging features)
unsigned char scriptDone;

    // The performance flag register. Each bit represents a different flag.
    // If it is 1, then it means that the flag occurred some time since the last
    // getVariables request. This register is always 0 for the Micro Maestro
    // Because performance flags only apply to the Mini Maestros.
unsigned char performanceFlags;
};
typedef struct MaestroVariables MaestroVariables;

// Represents the non-channel-specific variables that can be read from
// a Micro Maestro using REQUEST_GET_VARIABLES.
struct MicroMaestroVariables
{
    // The number of values on the data stack (0-32). A value of 0 means the stack is empty.
    unsigned char stackPointer;

    // The number of return locations on the call stack (0-10). A value of 0 means the stack is empty.
    unsigned char callStackPointer;

    // The error register. Each bit stands for a different error (see uscError).
    // If the bit is one, then it means that error occurred some time since the last
    // GET_ERRORS serial command or CLEAR_ERRORS USB command.
    unsigned short errors;

    // The address (in bytes) of the next bytecode instruction that will be executed.
    unsigned short programCounter;

    // Meaningless bytes to protect the program from stack underflows.
    unsigned short buffer[3];

    // The data stack used by the script. The values in locations 0 through stackPointer-1
    // are on the stack.
    unsigned short stack[32];

    // The call stack used by the script. The addresses in locations 0 through
    // callStackPointer-1 are on the call stack. The next return will make the
    // program counter go to callStack[callStackPointer-1].
    unsigned short callStack[10];

    // 0 = script is running.
    // 1 = script is done.
    // 2 = script will be done as soon as it executes one more instruction
    // (used to implement step-through debugging features)
unsigned char scriptDone;

    // Meaningless byte to protect the program from call stack overflows.
    unsigned char buffer2;

    // Container for channel information. The struct is 1 byte too large (should be 7, not 8)
    ServoStatus ServoSettings[6];
};
typedef struct MicroMaestroVariables MicroMaestroVariables;

// Represents the variables that can be read from a Mini Maestro using REQUEST_GET_VARIABLES.
struct MiniMaestroVariables
{
    // The number of values on the data stack (0-32). A value of 0 means the stack is empty.
    unsigned char stackPointer;

    // The number of return locations on the call stack (0-10). A value of 0 means the stack is empty.
    unsigned char callStackPointer;
unsigned short errors;
// The address (in bytes) of the next bytecode instruction that will be executed.
unsigned short programCounter;

unsigned char scriptDone;

unsigned char performanceFlags;

typedef struct MiniMaestroVariables MiniMaestroVariables;

typedef struct MiniMaestroVariables MiniMaestroVariables;

using namespace std;

class MaestroController
{
public:
    MaestroController(void);
    ~MaestroController(void);

    // system actions
    void initializeSystem(void);
    void centerUAV(void);
    void removeBatteryUAV(void);
    void chargeSpentBattery(char openStation);
    void getChargedBattery(char chargedStation);
    void replaceBatteryUAV(void);
    void retractArms(void);
    void disableAll(void);

    // battery functions that the station must call
    bool batteryPresent(char station);
    char doneCharging(void);

private:
    // stepper functions
    void homeStepper(void);
    void moveToStation(char target);
    char getStepperLocation(void);
    void moveCCW(char stations);
    void moveCW(char stations);
    void wiggle(ros::Rate timer);
    void shiftCW(void);
    void shiftCCW(void);

    // cart functions
    void centerCart(void);
    void moveCartLeft(void);
    void moveCartRight(void);
    int moveScissorLift(short height, bool up);

    // servo functions
    int moveServos(short* serial, short* channel, short* value, char numServos);
    void setTarget(short serial, short value, short channel);
    void setSpeed(short serial, short value, short channel);
    void setAcceleration(short serial, short value, short channel);
    void disableServo(short serial, short channel);

    // battery functions
    int startCharger(char station);
// low-level functions
void initDevices(void);
void closeDevices(void);
void openDevices(bool print);
int controlTransfer(short serial, unsigned char RequestType, unsigned char Request, short Value, short Index);
int controlTransfer(short serial, unsigned char RequestType, unsigned char Request, short Value, short Index,
                    void * data, short length);

// input/status-related functions
void getChannelValues(short* serial, short* channel, short* positions, short numServos);
void displayStatus(short serial);
void getVariables(short serial, MaestroVariables variables, short* stack, short* callStack, ServoStatus* servos);
void getVariables(short serial, ServoStatus* servos);
void getVariables(short serial, short* stack, bool isStack);
void getVariables(short serial, short* stack, short* callStack, ServoStatus* servos);
void getVariablesMiniMaestro(short serial, short* stack, short* callStack, ServoStatus* servos);

// containers
libusb_device* primary_dev, *secondary_dev;
libusb_device_handle* primary_hand, *secondary_hand;
char stepperAt, stepsPerStation;

#endif // MAESTRO_CONTROLLER_HPP

/*
 * BatteryPack.hpp
 * 
 *  Created on: Aug 26, 2011
 *      Author: ra23501
 */

#ifndef BATTERYPACK_HPP_
#define BATTERYPACK_HPP_
#include "ros/ros.h"
#define MAX_TIME 1200
#define MAX_VOLTAGE 12.6

class BatteryPack
{
public:
    int cycles;
    float voltage; int time;
    char location; // -1 is UAV, otherwise is bay number
    BatteryPack();
    BatteryPack(int cyc, int time, float volts, char loc);
    ~BatteryPack();
};
#endif /* BATTERYPACK_HPP_ */

/*
 * BaseStation.cpp
 * 
 *  Created on: Aug 26, 2011
 *      Author: ra23501
 */

#include "BaseStation.hpp"
#define ROS_NODE "base_station"
// Public methods
BaseStation::BaseStation(void) :
   nh_(), private_nh_("~")
{
   // init params
   BaseStation::initParams();

   // init subscribers
   BaseStation::initSubscribers();

   // init batteries
   //BaseStation::initBatteries();

   // start base thread
   BaseStation::startBaseThread();

   // start listener thread
   BaseStation::startListenThread();
}

BaseStation::~BaseStation()
{
   // stop all threads
   BaseStation::stopBaseThread();
   BaseStation::stopListenThread();
}

// main function
int main(int argc, char **argv)
{
   // initialize ros node handle
   ros::init(argc, argv, ROS_NODE);

   // print start to log file
   ROS_INFO("Started node %s", ROS_NODE);
   ROS_DEBUG("Started node %s", ROS_NODE);

   // create class instance
   BaseStation base_station;
   MaestroController controller;

   controller.disableAll();
   //controller.initializeSystem();
   //ros::Duration(5).sleep();
   controller.centerUAV();
   //ros::Duration(5).sleep();
   //ROS_ERROR("About to move cart");
   controller.retractArms();

   // ros spin
   ros::spin();

   // print termination to log file
   ROS_INFO("Stopped node %s", ROS_NODE);
   ROS_ERROR("Node stopped");

   // return success
   return (0);
}

#include "BaseStation.hpp"
// callback for robot state update
void BaseStation::callbackStateUpdate(const boost::shared_ptr<wpi_msgs::RobotState const> &robot_state_msg)
{
    mutex_.lock();
    received_robot_update_ = true;
    robot_state_ = *robot_state_msg;
    current_UAV_ = robot_state_msg->uid;
    ROS_DEBUG("Received state update.
    ");
    mutex_.unlock();
}

// callback for land request message
void BaseStation::callbackLandRequestMsg(const boost::shared_ptr<wpi_msgs::LandRequest const> &land_request_msg)
{
    mutex_.lock();
    if (UAV_landing_)
    {
        // this will only occur once per UAV as it wants to land, as it only sends out one request
        waiting_UAVs_.push_back(land_request_msg->uid);
    }
    else
    {
        received_land_request_ = true;
        current_UAV_ = land_request_msg->uid;
        UAV_landing_ = true;
        ROS_DEBUG("Received land request.
        ");
    }
    mutex_.unlock();
}

// callback for land confirm message
void BaseStation::callbackLandConfirmMsg(const boost::shared_ptr<wpi_msgs::LandConfirm const> &land_confirm_msg)
{
    mutex_.lock();
    if (land_confirm_msg->confirm)
    {
        received_positive_land_confirm_ = true;
        ROS_DEBUG("Received positive confirm.
        ");
    }
    else
    {
        received_negative_land_confirm_ = true;
        ROS_DEBUG("Received negative confirm.
        ");
    }
    mutex_.unlock();
}

// init params
void BaseStation::initParams(void)
{
    // ros params
    private_nh_.param("uid", uid_, string("base"));
    private_nh_.param("base_thread_rate", base_thread_rate_, 20.0);
    private_nh_.param("max_battery_voltage", max_battery_voltage_, 12.6);
    private_nh_.param("min_battery_voltage", min_battery_voltage_, 9.5);
    private_nh_.param("return_battery_voltage", return_battery_voltage_, 10.5);

    // init params
    robot_state_.uid = uid;
    robot_state_.translation = geometry_msgs::Vector3();
}
robot_state_.rotation = geometry_msgs::Quaternion();
robot_state_.rotation.w = 1.0;
robot_state_.battery_voltage = max_battery_voltage_; 
base_state_ = LAUNCHING;

// init volatile bools
received_robot_update_ = true;
received_land_request_ = false;
received_positive_land_confirm_ = false;
received_negative_land_confirm_ = false;
UAV_landing_ = true;
}

// init subscribers
void BaseStation::initSubscribers(void)
{
    robot_state_sub_ = nh_.subscribe(string("/robot_state"), 10, &BaseStation::callbackStateUpdate, this);
    land_request_sub_ = nh_.subscribe(string("land_request"), 10, &BaseStation::callbackLandRequestMsg, this);
    land_confirm_sub_ = nh_.subscribe(string("land_confirm"), 10, &BaseStation::callbackLandConfirmMsg, this);
}

// init batteries
void BaseStation::initBatteries(void)
{
    int numOpen = 0;
    openStation = NUM_BATTERIES;
    for (int i = 0; i < NUM_BATTERIES; i++)
    {
        allBatteries[i] = new BatteryPack(0, 0, 0, i);
        if (controller.batteryPresent(i))
        {
            readyBatteries.push_back(allBatteries[i]);
        } else {
            openStation = i;
            allBatteries[i]->location = -1;
            numOpen++;
        }
    }
    if(openStation == NUM_BATTERIES) {
        ROS_ERROR("Too many batteries or communication error, no open station.");
    }
    if(numOpen > 1) {
        ROS_ERROR("Too few batteries or dislodged battery.");
    }
}

/**
 * BaseStation_state.cpp
 *
 *  Created on: Aug 31, 2011
 *      Author: ra23501
 */

// includes
#include "BaseStation.hpp"

// idle state
void BaseStation::idleState(void)
{
    // int ready;
    // // poll internal batteries to see which are done charging
    // if (((ready = controller.doneCharging()) != NUM_BATTERIES)
    // {
    //     for (int i = 0; i < NUM_BATTERIES; i++)
    //     {
    //         if (allBatteries[i]->location == ready)
    //         {
    //             readyBatteries.push_back(allBatteries[i]);
    //         }
    //     }
    //     }
    // }
}
if (all_UAVs_.size())
{
    base_state_ = LANDING;
}
}

// landing state
void BaseStation::landingState(void)
{
    //TODO: add code to check for landing
    // can check for translation == 0, or can add to the message to see if it has powered down motors
    if (robot_state_.translation.z == 0)
        base_state_ = CENTERING;
}

// centering state
void BaseStation::centeringState(void)
{
    controller.centerUAV();
    base_state_ = REMOVING_BATTERY;
}

// removing battery state
void BaseStation::removingBatteryState(void)
{
    controller.removeBatteryUAV();
    controller.chargeSpentBattery(openStation);
    for (int i = 0; i < NUM_BATTERIES; i++)
    {
        // update the location of the battery that is now charging
        if (allBatteries[i]->location == -1)
        {
            allBatteries[i]->location = openStation;
            break;
        }
    }
    base_state_ = REPLACING_BATTERY;
}

// replacing battery state
void BaseStation::replacingBatteryState(void)
{
    char ready = readyBatteries.at(0)->location;
    for (int i = 0; i < NUM_BATTERIES; i++)
    {
        // update the location of the battery that will be in the UAV
        if (allBatteries[i]->location == ready)
        {
            allBatteries[i]->location = -1;
            openStation = ready;
            break;
        }
    }
    readyBatteries.erase(readyBatteries.begin());
    controller.getChargedBattery(ready);
    controller.replaceBatteryUAV();
    if (robot_state_.battery_voltage > 12)
        base_state_ = LAUNCHING;
}

// launching state
void BaseStation::launchingState(void)
{
    //controller.retractArms();
    //TODO: add code to clear UAV for launch
    base_state_ = IDLE;
    UAV_landing_ = false;
}
// includes
#include "BaseStation.hpp"

// start base thread
void BaseStation::startBaseThread(void)
{
    base_thread_ = boost::shared_ptr < boost::thread > (new
    boost::thread(boost::bind(&BaseStation::baseThread, this)));
}

// base thread
void BaseStation::baseThread(void)
{
    // set thread rate
    ros::Rate thread_rate(base_thread_rate_);

    // while node active
    while (ros::ok())
    {
        // base state machine
        mutex_.lock();
        switch (base_state_)
        {
        case (IDLE):
            idleState();
            break;
        case (LANDING):
            landingState();
            break;
        case (CENTERING):
            centeringState();
            break;
        case (REMOVING_BATTERY):
            removingBatteryState();
            break;
        case (REPLACING_BATTERY):
            replacingBatteryState();
            break;
        case (LAUNCHING):
            launchingState();
            break;
        default:
            break;
        }
        mutex_.unlock();

        // regulate thread rate
        thread_rate.sleep();
    }
}

// stop base thread
void BaseStation::stopBaseThread(void)
{
    base_thread_ -> join();
}

// start listen thread
void BaseStation::startListenThread(void)
{
    listen_thread_ = boost::shared_ptr < boost::thread
    > (new boost::thread(boost::bind(&BaseStation::listenThread, this)));
}

// listens for requests and status updates from all UAVs in communication
// publishes landing requests/confirmations in applicable cases
void BaseStation::listenThread(void)
{
    // set thread rate
    ros::Rate thread_rate(base_thread_rate_);

    // while node active
    while (ros::ok())
    {
        // base state machine
        mutex_.lock();
        findNodes();
        if (received_land_request_) // one UAV wants to land
        {
            // reset bools
            received_land_request_ = false;

            // publish confirm message
            wpi_msgs::LandConfirm land_confirm_msg;
            land_confirm_msg.header.stamp = ros::Time::now();
            land_confirm_msg.uid = uid_;
            land_confirm_msg.confirm = true;
            for (unsigned int i = 0; i < land_confirm_pub_.size(); i++)
            {
                if (land_confirm_pub_.at(i).getTopic().find(current_UAV_) != string::npos)
                {
                    land_confirm_pub_.at(i).publish(land_confirm_msg);
                    break;
                }
            }
        }
        else if ((waiting_UAVs_.size() > 0 && !UAV_landing_)) // UAVs are waiting and the base platform is open
        {
            // reset bools
            received_land_request_ = false;
            received_positive_land_confirm_ = false;
            UAV_landing_ = true;

            // publish confirm message
            wpi_msgs::LandConfirm land_confirm_msg;
            land_confirm_msg.header.stamp = ros::Time::now();
            land_confirm_msg.uid = uid_;
            land_confirm_msg.confirm = true;
            for (unsigned int i = 0; i < land_confirm_pub_.size(); i++)
            {
                if (land_confirm_pub_.at(i).getTopic().find(waiting_UAVs_.at(0)) != string::npos)
                {
                    land_confirm_pub_.at(i).publish(land_confirm_msg);
                    waiting_UAVs_.erase(waiting_UAVs_.begin());
                    break;
                }
            }

            // change state
            base_state_ = LANDING;
        }
        else if (received_positive_land_confirm_ && !UAV_landing_) // a UAV agreed to land and the base is open
        {
            // reset bools
            received_positive_land_confirm_ = false;
            UAV_landing_ = true;

            // change state
            base_state_ = LANDING;
        }
        else if (received_robot_update_)
        {
            // reset bools
            received_robot_update_ = false;
            if (robot_state_.battery_voltage <= return_battery_voltage_ && !UAV_landing_) // UAV has low voltage, request it to land
            {
                // publish confirm message
                wpi_msgs::LandConfirm land_confirm_msg;
                land_confirm_msg.header.stamp = ros::Time::now();
                land_confirm_msg.uid = uid_;
                land_confirm_msg.confirm = true;
                for (unsigned int i = 0; i < land_confirm_pub_.size(); i++)
                {
                    if (land_confirm_pub_.at(i).getTopic().find(robot_state_.doctor_.current_UAV) != string::npos)
                    {
                        land_confirm_pub_.at(i).publish(land_confirm_msg);
                        break;
                    }
                }

                // change state
                base_state_ = LANDING;
            }
        }
    }
}
// publish request message
wpi_msgs::LandRequest land_request_msg;
land_request_msg.header.stamp = ros::Time::now();
land_request_msg.uid = uid;
for (unsigned int i = 0; i < land_request_pub_.size(); i++)
{
    if (land_request_pub_.at(i).getTopic().find(current_UAV_) != std::string::npos)
    {
        land_request_pub_.at(i).publish(land_request_msg);
        UAV_landing_ = true; // UAVs are programmed to always land when requested from the base
        break;
    }
}
mutex_.unlock();

// regulate thread rate
thread_rate.sleep();

}// stop listen thread
void BaseStation::stopListenThread(void)
{
    listen_thread_->join();
}

#include "BaseStation.hpp"

// queries the ros master to obtain a list of all possible nodes to connect to
// and connects to those that have the /uid field, signaling a UAV
void BaseStation::findNodes(void)
{
    std::vector<std::string> nodes;
    std::string name;
    ros::NodeHandle nh;
    std::string param;
    unsigned int diffCount = 0;
    ros::master::getNodes(nodes);
    for (unsigned int i = 0; i < nodes.size(); i++)
    {
        diffCount = 0;
        name = nodes.at(i);
        if (nh.getParam(name + string("/uid"), param))
        {
            for (unsigned int i = 0; i < all_UAVs_.size(); i++)
            {
                if (param.compare(all_UAVs_.at(i)) != 0)
                {
                    diffCount++;
                }
            }
            if (diffCount == all_UAVs_.size())
            {
                addNode(param);
            }
        }
    }
}

// adds the node to the list of publishers, as it is a node not previously seen
void BaseStation::addNode(string name)
{
    all_UAVs_.push_back(name);
    land_request_pub_.push_back(nh_.advertise<wpi_msgs::LandRequest>(name + "/land_request", 10, true));
    land_confirm_pub_.push_back(nh_.advertise<wpi_msgs::LandConfirm>(name + "/land_confirm", 10, true));
}

#include "BatteryPack.hpp"

BatteryPack::BatteryPack()
{
    this->cycles = 0;
    this->time = MAX_TIME;
    this->voltage = MAX_VOLTAGE;
    this->location = 0;
}

BatteryPack::BatteryPack(int cyc, int time, float volts, char loc)
{
    this->cycles = cyc;
    this->time = time;
    this->voltage = volts;
    this->location = loc;
}

BatteryPack::~BatteryPack()
{
}

#include "MaestroController.hpp"

MaestroController::MaestroController(void) {
    initDevices();
    openDevices(false);
}

MaestroController::~MaestroController(void) {
    closeDevices();
}

#include "MaestroController_actions.hpp"

void MaestroController::initializeSystem(void)
{
    stepsPerStation = 0;
}
short serials[] = {PRIMARY_SERIAL};
short servos[] = {ARM_SERVO_0};
short targets[] = {1874 * 4};
short numServo;
ROS_ERROR("Initialize systems");
disableAll();
numServo = moveServos(serials, servos, targets, 1);
ros::Duration(2).sleep();
disableServo(PRIMARY_SERIAL, ARM_SERVO_0);
servos[0] = ARM_SERVO_1;
targets[0] = 1956 * 4;
numServo = moveServos(serials, servos, targets, 1);
ros::Duration(2).sleep();
disableServo(PRIMARY_SERIAL, ARM_SERVO_1);
movescissorLift(SCISSOR_DOWN, false);
ros::Duration(1).sleep();
homeStepper();
centerCart();
movescissorLift(SCISSOR_UAV, true);
ros::Duration(1).sleep();
movescissorLift(SCISSOR_REST, false);
}

// actuate the arms to center the UAV on the platform
// pre conditions: UAV has landed inside the arms' area
// post condition: UAV is centered over the platform and is retained vertically by the arms
void MaestroController::centerUAV(void)
{
short serials[] = {PRIMARY_SERIAL};
short servos[] = {ARM_SERVO_0};
short targets[] = {1170 * 4};
short numServo;
ROS_ERROR("center UAV");
numServo = moveServos(serials, servos, targets, 1);
ros::Duration(2).sleep();
disableServo(PRIMARY_SERIAL, ARM_SERVO_0);
servos[0] = ARM_SERVO_1;
targets[0] = 1200 * 4;
numServo = moveServos(serials, servos, targets, 1);
ros::Duration(2).sleep();
disableServo(PRIMARY_SERIAL, ARM_SERVO_1);
}

// remove the battery from the UAV, assume it always works
// pre conditions: UAV has been centered over the scissor lift
// post conditions: UAV is battery-less, spent battery is on the scissor lift, which is in the lowest state
void MaestroController::removeBatteryUAV(void)
{
short numServo;
if ((numServo = moveScissorLift(SCISSOR_UAV, true)) != -1)
{
  ROS_ERROR("Servo %d stalled out.", numServo);
}
if ((numServo = moveScissorLift(SCISSOR_DOWN, false)) != -1)
{
  ROS_ERROR("Servo %d stalled out.", numServo);
}
}

// load the battery recently taken from the UAV into the open charging slot
// makes sure battery has started to charge before moving on
// pre conditions: battery has started charging, scissor lift is in lowest state and at one end of the
// track, scissor lift is empty
void MaestroController::chargeSpentBattery(char openStation)
{
  char location = getStepperLocation();
  if (location == openStation)
  {
    moveCartLeft();
  }
}
/*else if (location == ((openStation + 4) % 8)) // opposite station, as cart can move to both ends
  {moveCartRight();}*/
else
{
  moveToStation(openStation); // optimizes for smallest movement, must check direction again
  if (location == openStation)
  {
    moveCartLeft();
  } /*else if (location == ((openStation + 4) % 8))
  {moveCartRight();}*/
}
while (startCharger(openStation) != 0) // keeps trying until charging has successfully started
{
  if (moveScissorLift(SCISSOR_CHARGE, true) != -1)
  {
    ROS_ERROR("Servo stalled out.");
  }
  if (moveScissorLift(SCISSOR_DOWN, false) != -1)
  {
    ROS_ERROR("Servo stalled out.");
  }
}

// obtain the battery that was first to be done charging
// pre conditions: scissor lift is in lowest state and is empty
// post conditions: scissor lift is in lowest state with a charged battery, cart is centered and turntable is homed

void MaestroController::getChargedBattery(char chargedStation)
{
  char location = getStepperLocation();
  if (location == chargedStation)
  {
    moveCartLeft();
  } /*else if (location == ((chargedStation + 4) % 8))
  {moveCartRight();}*/
else
{
  moveToStation(chargedStation);
  if (location == chargedStation)
  {
    moveCartLeft();
  } /*else if (location == ((chargedStation + 4) % 8))
  {moveCartRight();}*/
}
while (batteryPresent(chargedStation)) // keep trying to dislodge battery until it is no longer detected by the charger
{
  if (moveScissorLift(SCISSOR_CHARGE, true) != -1)
  {
    ROS_ERROR("Servo stalled out.");
  }
  if (moveScissorLift(SCISSOR_DOWN, false) != -1)
  {
    ROS_ERROR("Servo stalled out.");
  }
}
centerCart(); // in preparation for placing in the UAV
homeStepper(); // home stepper for correct alignment between scissor lift and upper platform
}
// place the battery in the UAV
// pre conditions: scissor lift is in lowest state with a charged battery, cart is centered, turntable is homed
// post conditions: battery is placed in UAV, scissor lift is in resting (flush with landing plate) state

void MaestroController::replaceBatteryUAV(void)
{
    // cart is already at the center, just need to lift it and then return to resting position
    // assumes that the battery engages first try, can be made to check with UAV if it actually engaged later
    if (moveScissorLift(SCISSOR_UAV, true) != -1)
    {
        ROS_ERROR("Servo stalled out.");
    }
    if (moveScissorLift(SCISSOR_REST, false) != -1)
    {
        ROS_ERROR("Servo stalled out.");
    }
}

// remove the arms from retaining the UAV in preparation for takeoff
// pre conditions: battery has been replaced in the UAV
// post conditions: retaining arms are retracted and UAV has or is in the process of taking off

void MaestroController::retractArms(void)
{
    // temporary testing code
    /*
    disableAll();
    ROS_ERROR("lower scissor lift");
    moveScissorLift(SCISSOR_DOWN, false);
    ros::Duration(1).sleep();
    
    ROS_ERROR("center cart");
    centerCart();
    disableAll();
    ROS_ERROR("raise scissor lift");
    moveScissorLift(SCISSOR_UAV, true);
    disableAll();
    ros::Duration(2).sleep();
    ROS_ERROR("flush scissor lift");
    moveScissorLift(SCISSOR_REST, false);
    disableAll();
    ros::Duration(5).sleep();
    */
    /*
    homeStepper();
    ros::Duration(2).sleep();
    moveToStation(3);
    ros::Duration(2).sleep();
    moveToStation(5);
    */
    moveScissorLift(SCISSOR_UAV, true);
    ros::Duration(1).sleep();
    moveScissorLift(SCISSOR_DOWN, false);
    ros::Duration(1).sleep();
    moveCartLeft();
    ros::Duration(1).sleep();
    moveScissorLift(SCISSOR_CHARGE, true);
    ros::Duration(1).sleep();
    moveScissorLift(SCISSOR_DOWN, false);
    ros::Duration(1).sleep();
    moveToStation(3);
    ros::Duration(1).sleep();
    moveScissorLift(SCISSOR_CHARGE, true);
    ros::Duration(1).sleep();
    moveScissorLift(SCISSOR_DOWN, false);
    ros::Duration(1).sleep();
    centerCart();
    ros::Duration(1).sleep();
    moveToStation(4);
}
ros::Duration(1).sleep();
moveScissorLift(SCISSOR_UAV, true);
ros::Duration(1).sleep();
moveScissorLift(SCISSOR_REST, false);
ros::Duration(1).sleep();

//end of testing code
short serials[] = {PRIMARY_SERIAL};
short servos[] = {ARM_SERVO_0};
short targets[] = {1874 * 4};
short numServo;
ROS_ERROR("retract arms");
numServo = moveServos(serials, servos, targets, 1);
ros::Duration(2).sleep();
disableServo(PRIMARY_SERIAL, ARM_SERVO_0);
servos[0] = ARM_SERVO_1;
targets[0] = 1956 * 4;
numServo = moveServos(serials, servos, targets, 1);
ros::Duration(2).sleep();
disableServo(PRIMARY_SERIAL, ARM_SERVO_1);
}

void MaestroController::disableAll(void)
{
    disableServo(PRIMARY_SERIAL, ARM_SERVO_0);
    disableServo(PRIMARY_SERIAL, ARM_SERVO_1);
    disableServo(PRIMARY_SERIAL, CART_0_SERVO);
    disableServo(PRIMARY_SERIAL, CART_0_MOTOR);
}

#include "MaestroController.hpp"

bool MaestroController::batteryPresent(char station)
{
    short batteryHere[1];
    short serial[] = {SECONDARY_SERIAL};
    short channel[] = {BATTERY_PRESENT_0};
    switch (station)
    {
    case (0):
        channel[0] = BATTERY_PRESENT_0;
        break;
    case (1):
        channel[0] = BATTERY_PRESENT_1;
        break;
    case (2):
        channel[0] = BATTERY_PRESENT_2;
        break;
    case (3):
        channel[0] = BATTERY_PRESENT_3;
        break;
    case (4):
        channel[0] = BATTERY_PRESENT_4;
        break;
    case (5):
        channel[0] = BATTERY_PRESENT_5;
        break;
    case (6):
        channel[0] = BATTERY_PRESENT_6;
        break;
    case (7):
        channel[0] = BATTERY_PRESENT_7;
        break;
    }
getChannelValues(serial, channel, batteryHere, 1);
return batteryHere[0] == ON;
}

// attempts to start charging the battery in the specified station, returns -1 on no battery found
int MaestroController::startCharger(char station)
{
  ros::Rate wait(50);
  ros::Rate sample(3);
  short serial[] = {SECONDARY_SERIAL};
  short channel[] = {station};
  short LED_color[1];
  char isCharging = 0;
  bool present = batteryPresent(station - BATTERY_START_RELAY_0);
  if (present)
  {
    wait.reset();
    setTarget(serial[0], ON, channel[0]);
    wait.sleep();
    setTarget(serial[0], OFF, channel[0]);
    wait.sleep();
    sample.reset();
    for (int i = 0; i < 3; i++) // sample 3 times over one second, blink at 2Hz
    {
      getChannelValues(serial, channel, LED_color, 1);
      if (ISORANGE(LED_color[0]))
        isCharging++;
      sample.sleep();
    }
    if (isCharging == 0)
    {
      return -1; // charged or disconnected/error
    }
    else if (isCharging < 3) // LEDs will be solid orange when charging, if not solid, is still waiting, so
    press again
    {
      wait.reset();
      setTarget(serial[0], ON, channel[0]);
      wait.sleep();
      setTarget(serial[0], OFF, channel[0]);
    }
    return 0;
  }
  else
  return -1;
}

// polls the channels for the stop charge signal from each of the chargers
// if a battery is done charging, its station number is returned
char MaestroController::doneCharging(void)
{
  short serial[] = {SECONDARY_SERIAL, SECONDARY_SERIAL, SECONDARY_SERIAL, SECONDARY_SERIAL,
                   SECONDARY_SERIAL, SECONDARY_SERIAL, SECONDARY_SERIAL, SECONDARY_SERIAL};
  short channels[] = {BATTERY_LED_0, BATTERY_LED_1, BATTERY_LED_2, BATTERY_LED_3, BATTERY_LED_4,
                      BATTERY_LED_5, BATTERY_LED_6, BATTERY_LED_7};
  short values[NUM_BATTERIES];
  char green_check[NUM_BATTERIES];
  ros::Rate sample(3);
  for (int i = 0; i < 3; i++) // sample 3 times over one second, blink at 2Hz
  {
    getChannelValues(serials, channels, values, NUM_BATTERIES);
    for (int k = 0; k < NUM_BATTERIES; k++)
    {
      if (ISGREEN(values[k]))
      {
        if (++green_check[k] == 3)
        {
          return k;
        }
      }
    }
  }
  return -1; // no battery found
void MaestroController::centerCart(void)
{
    short cartPositions[2/*3*/];
    short serials[] = {PRIMARY_SERIAL, /*PRIMARY_SERIAL,*/ PRIMARY_SERIAL};
    short channels[] = {CART_END_0, /*CART_END_1,*/ CART_MID};
    ros::Rate delay(60);
    getChannelValues(serials, channels, cartPositions, 2/*3*/);
    if (cartPositions[0] == ON_INP) // left end
    {
        setTarget(PRIMARY_SERIAL, CART_RIGHT, CART_0_MOTOR);
        while (cartPositions[1/*2*/] != ON_INP)
        {
            getChannelValues(&(serials[1/*2*/]), &(channels[1/*2*/]), &(cartPositions[1/*2*/]), 1);
            delay.sleep();
        }
        setTarget(PRIMARY_SERIAL, STOP, CART_0_MOTOR);
        disableServo(PRIMARY_SERIAL, CART_0_MOTOR);
    } /*else if (cartPositions[1] == ON) // right end
    {
        setTarget(PRIMARY_SERIAL, CART_LEFT, CART_0_MOTOR);
        while (cartPositions[2] == OFF)
        {
            getChannelValues(&(serials[2]), &(channels[2]), &(cartPositions[2]), 1);
        }
        setTarget(PRIMARY_SERIAL, STOP, CART_0_MOTOR);
    }*/
    else if (cartPositions[1/*2*/] == ON) // middle
    {
        return;
    } else
    {
        setTarget(PRIMARY_SERIAL, CART_RIGHT, CART_0_MOTOR);
        {
            setTarget(PRIMARY_SERIAL, CART_LEFT, CART_0_MOTOR);
            while (cartPositions[2] == OFF)
            {
                getChannelValues(&(serials[2]), &(channels[2]), &(cartPositions[2]), 1);
            }
            setTarget(PRIMARY_SERIAL, STOP, CART_0_MOTOR);
        }*/
        else /*if (cartPositions[1/*2*/] == ON) // middle
        {
            return;
        }
// moves the cart to the left end of the track, causing leftwards motion until it is on the left side
void MaestroController::moveCartLeft(void)
{
    short cartPositions[1];
    short serials[] = {PRIMARY_SERIAL};
    short channels[] = {CART_END_0};
    getChannelValues(serials, channels, cartPositions, 1);
    ros::Rate delay(60);
    if (*cartPositions == ON_INP) // left end
    {
        return;
    }
    else
    {
        setTarget(PRIMARY_SERIAL, CART_LEFT, CART_0_MOTOR);
        while (*cartPositions != ON_INP)
        {
            getChannelValues(serials, channels, cartPositions, 1);
            delay.sleep();
        }
        setTarget(PRIMARY_SERIAL, STOP, CART_0_MOTOR);
        disableServo(PRIMARY_SERIAL, CART_0_MOTOR);
    }
}

// moves the cart to the right end of the track, causing rightwards motion until it is on the right side
void MaestroController::moveCartRight(void)
{
    short cartPositions[1];
    short serials[] = {PRIMARY_SERIAL};
    short channels[] = {CART_END_1};
    getChannelValues(serials, channels, cartPositions, 1);
    if (*cartPositions == ON) // right end
    {
        return;
    }
    else
    {
        setTarget(PRIMARY_SERIAL, CART_RIGHT, CART_0_MOTOR);
        while (*cartPositions == OFF)
        {
            getChannelValues(serials, channels, cartPositions, 1);
        }
        setTarget(PRIMARY_SERIAL, STOP, CART_0_MOTOR);
    }
}

// commands the scissor lift servo to move to the specified height
int MaestroController::moveScissorLift(short height, bool up)
{
    short serial[] = {PRIMARY_SERIAL};
    short channel[] = {CART_0_SERVO};
    short value[1];
    ros::Rate timer(30);
    if (up)
    {
        value[0] = SCISSOR_MAX_UP;
    }
    else
    {
        value[0] = SCISSOR_MAX_DOWN;
        ROS_ERROR("Move scissor");
    }
    int ret = moveServos(serial, channel, value, 1);
    int count = 0;
    if (height == SCISSOR_DOWN)
    {
        channel[0] = 23;
        while (value[0] != ON_INP)
        {
            getChannelValues(serial, channel, value, 1);
        }
    }
}
if (count == 3)
{
    wiggle(timer);
    count = 0;
}
else
{
    count++;
    timer.sleep();
    timer.sleep();
    timer.sleep();
    timer.sleep();
    timer.sleep();
    timer.sleep();
    timer.sleep();
    timer.sleep();
}
}
disableServo(PRIMARY_SERIAL, CART_0_SERVO);
return ret;

if (height == SCISSOR_UAV)
{
    channel[0] = 22;
    while (value[0] != ON_INP)
    {
        getChannelValues(serial, channel, value, 1);
        if (count == 3)
        {
            wiggle(timer);
            count = 0;
        }
        else
        {
            count++;
            timer.sleep();
            timer.sleep();
            timer.sleep();
            timer.sleep();
            timer.sleep();
            timer.sleep();
            timer.sleep();
            timer.sleep();
        }
    }
    ros::Rate t_wait(10);
    t_wait.reset();
    t_wait.sleep();
    disableServo(PRIMARY_SERIAL, CART_0_SERVO);
    return ret;
}

channel[0] = SCISSOR_POT;
long sum = 0;
count = 1;
bool in_pos = false;
while (1)
{
    getChannelValues(serial, channel, value, 1);
    if ((count % 10) == 0 && count != 0)
    {
        sum /= 10;
        int tar_pot = (int)(((float)height) * 0.0571));
        ROS_ERROR("target is \%d, current avg is \%ld", tar_pot, sum);
        if (sum >= tar_pot - 5 && sum <= tar_pot + 5)
        {
            ROS_ERROR("In range.");
            value[0] = height;
            channel[0] = CART_0_SERVO;
            ROS_ERROR("Slowing down scissor");
        }
    }
ROS_ERROR("Sending value %d", value[0]);
ret = moveServos(serial, channel, value, 1);
ros::Rate delay(3);
	//tar_pot += 20;
while (!in_pos)
{
	disableServo(PRIMARY_SERIAL, CART_0_SERVO);
delay.sleep();
channel[0] = SCISSOR_POT;
getChannelValues(serial, channel, value, 1);
if (value[0] >= tar_pot - 5 && value[0] <= tar_pot + 5)
{
	in_pos = true;
}
else
{
	ROS_ERROR("Error is %d, value is %d, target is %d, outside dead band", value[0] - tar_pot, value[0], tar_pot);
	int diff = tar_pot - value[0];
	ROS_ERROR("Difference is %d, height is %d, continuing to move", diff, height);

tar_pot = height + (diff * 50);
channel[0] = CART_0_SERVO;
ROS_ERROR("Sending value %d", value[0]);
ret = moveServos(serial, channel, value, 1);
wiggle(timer);
delay.sleep();
delay.sleep();
}
ROS_ERROR("Disabling scissor");
disableServo(PRIMARY_SERIAL, CART_0_SERVO);
channel[0] = SCISSOR_POT;
break;
}
sum = 0;
}
else
{
	sum += value[0];
}
if (count == 20)
{
	wiggle(timer);
count = 1;
}
else
{
count++;
}
return ret;
}
variables.callStackPointer = 0;
variables.errors = 0;
variables.performanceFlags = 0;
variables.programCounter = 0;
variables.scriptDone = 0;
variables.stackPointer = 0;
ServoStatus* servos;
if (serial[0] == MICRO_SERIAL)
{
    servos = new ServoStatus[6];
}
else
{
    servos = (ServoStatus*)calloc(24, sizeof(ServoStatus));
}
short stack[variables.stackPointer];
short call_stack[variables.callStackPointer];
bool serialSame = true;
for (int i = 1; i < numServos; i++)
{
    serialSame = (serial[i] == serial[i - 1]);
}
if (serialSame) // if the controllers specified are all the same, only one call must be made
{
    getVariables(serial[0], variables, stack, call_stack, servos);
    for (int i = 0; i < numServos; i++)
    {
        positions[i] = servos[channel[i]].position;
    }
}
else // otherwise, make a call for each entry
{
    for (int i = 0; i < numServos; i++)
    {
        getVariables(serial[i], variables, stack, call_stack, servos);
        positions[i] = servos[channel[i]].position;
    }
}

void MaestroController::displayStatus(short serial)
{
    MaestroVariables variables;
    variables.callStackPointer = 0;
    variables.errors = 0;
    variables.performanceFlags = 0;
    variables.programCounter = 0;
    variables.scriptDone = 0;
    variables.stackPointer = 0;
    ServoStatus* servos;
    if (serial == MICRO_SERIAL)
    {
        servos = new ServoStatus[6];
    }
    else
    {
        servos = (ServoStatus*)calloc(24, sizeof(ServoStatus));
    }
    short stack[variables.stackPointer];
    short call_stack[variables.callStackPointer];
    getVariables(serial, variables, stack, call_stack, servos);
    ROS_ERROR("# target tspeed taccel tpos");
    if (serial == MICRO_SERIAL)
    {
        for (int i = 0; i < 6; i++)
        {
            ROS_ERROR("%d\t%hu\t%hu\t%hhu\t%hu", i, servos[i].target, servos[i].speed, servos[i].acceleration, servos[i].position);
        }
    }
for (int i = 0; i < 24; i++)
{
    ROS_ERROR("%d\t%hu\t%hu\t%hu\t%hu\t%hu", i, servos[i].target, servos[i].speed, servos[i].acceleration, servos[i].position);
}
ROS_ERROR("errors: %d", variables.errors);
ROS_ERROR("program counter: %d", variables.programCounter);

// function to get all variables from the controller
// individual calls differ based on the controller, which this handles by checking Mini/Micro
// pre conditions: USB has been initialized and at least one Maestro device has been identified
void MaestroController::getVariables(short serial, MaestroVariables variables, short* stack, short* callStack, ServoStatus* servos)
{
    if (serial == MICRO_SERIAL)
    {
        getVariablesMicroMaestro(serial, variables, stack, callStack, servos);
    } else
    {
        getVariablesMiniMaestro(serial, variables);
        getVariablesMiniMaestro(serial, servos);
        getVariablesMiniMaestro(serial, stack, true);
        getVariablesMiniMaestro(serial, callStack, false);
    }
}

// gets only the standard variables from the controller
// different calls for Mini/Micro
// pre conditions: USB has been initialized and at least one Maestro device has been identified
void MaestroController::getVariables(short serial, MaestroVariables variables)
{
    if (serial == MICRO_SERIAL)
    {
        ServoStatus servos[6];
        short stack[variables.stackPointer];
        short callStack[variables.callStackPointer];
        getVariablesMicroMaestro(serial, variables, stack, callStack, servos);
    } else
    {
        getVariablesMiniMaestro(serial, variables);
    }
}

// gets only the channel information from the controller
// different calls for Mini/Micro
// pre conditions: USB has been initialized and at least one Maestro device has been identified
void MaestroController::getVariables(short serial, ServoStatus* servos)
{
    if (serial == MICRO_SERIAL)
    {
        MaestroVariables variables;
        variables.callStackPointer = 0;
        variables.errors = 0;
        variables.performanceFlags = 0;
        variables.programCounter = 0;
        variables.scriptDone = 0;
        variables.stackPointer = 0;
        short stack[variables.stackPointer];
        short callStack[variables.callStackPointer];
        getVariablesMicroMaestro(serial, variables, stack, callStack, servos);
    }
else
{
    getVariablesMiniMaestro(serial, servos);
}
}

// gets only the stack or callstack from the controller
// different calls for Mini/Micro
// boolean determines whether to get stack (true) or callstack (false)
// pre conditions: USB has been initialized and at least one Maestro device has been identified
void MaestroController::getVariables( short serial, short* stack, bool isStack)
{
    if ( isStack ) // stack case
    {
        if ( serial == MICRO_SERIAL)
        {
            MaestroVariables variables;
            variables.callStackPointer = 0;
            variables.errors = 0;
            variables.performanceFlags = 0;
            variables.programCounter = 0;
            variables.scriptDone = 0;
            variables.stackPointer = 0;
            ServoStatus servos[6];
            short callStack[variables.callStackPointer];
            getVariablesMicroMaestro(serial, variables, stack, callStack, servos);
        }
    }
    else
    {
        getVariablesMiniMaestro(serial, stack, true);
    }
}
else // callstack case
{
    if ( serial == MICRO_SERIAL)
    {
        MaestroVariables variables;
        variables.callStackPointer = 0;
        variables.errors = 0;
        variables.performanceFlags = 0;
        variables.programCounter = 0;
        variables.scriptDone = 0;
        variables.stackPointer = 0;
        ServoStatus servos[6];
        short callStack[variables.stackPointer];
        getVariablesMicroMaestro(serial, variables, callStack, stack, servos);
    }
    else
    {
        getVariablesMiniMaestro(serial, stack, false);
    }
}

// queries the Micro controller for all variable information
// can only be used on the Micro, as the structures are a certain size
// pre conditions: USB has been initialized and at least one Maestro device has been identified
void MaestroController::getVariablesMicroMaestro( short serial, MaestroVariables variables, short* stack, short* callStack, ServoStatus* servos)
{
    char* array = (char*)calloc(sizeof(MicroMaestroVariables), sizeof(char));
    controlTransfer(serial, 0xC0, REQUEST_GET_VARIABLES, 0, 0, array, sizeof(MicroMaestroVariables));
    char* pointer = array;
    MicroMaestroVariables tmp = *(MicroMaestroVariables*)pointer;
    variables.callStackPointer = tmp.callStackPointer;
    variables.scriptDone = tmp.scriptDone;
    variables.stackPointer = tmp.stackPointer;
    variables.programCounter = tmp.programCounter;
variables.scriptDone = tmp.scriptDone;
variables.performanceFlags = 0;

for (int i = 0; i < 6; i++)
{
    // the ServoStatus struct is rounded to 8 bytes by sizeof() instead of the 7 it takes up, must account
    // for the error
    servos[i] = *((ServoStatus*)((byte *)pointer + sizeof(MicroMaestroVariables) - (sizeof(ServoStatus) * (6 - i)) - i));
}

stack = new short[variables.stackPointer];
for (int i = 0; i < variables.stackPointer; i++)
{
    stack[i] = *(tmp.stack + i);
}

callStack = new short[variables.callStackPointer];
for (int i = 0; i < variables.callStackPointer; i++)
{
    callStack[i] = *(tmp.callStack + i);
}

// obtain just the variable information from the Mini controller
// pre conditions: USB has been initialized and at least one Maestro device has been identified
void MaestroController::getVariablesMiniMaestro(short serial, MaestroVariables variables)
{
    try
    {
        MiniMaestroVariables tmp;
        controlTransfer(serial, 0xC0, REQUEST_GET_VARIABLES, 0, 0, &tmp, (short)sizeof(MiniMaestroVariables));

        variables.stackPointer = tmp.stackPointer;
        variables.callStackPointer = tmp.callStackPointer;
        variables.errors = tmp.errors;
        variables.programCounter = tmp.programCounter;
        variables.scriptDone = tmp.scriptDone;
        variables.performanceFlags = tmp.performanceFlags;
    }
    catch (int e)
    {
        ROS_ERROR("Error getting variables from device.");
    }
}

// obtain just the channel information from the Mini controller
// pre conditions: USB has been initialized and at least one Maestro device has been identified
void MaestroController::getVariablesMiniMaestro(short serial, ServoStatus* servos)
{
    try
    {
        char* servoSettingsArray = (char*)calloc(SERVO_COUNT * sizeof(ServoStatus), sizeof(char));
        controlTransfer(serial, 0xC0, REQUEST_GET_SERVO_SETTINGS, 0, 0, servoSettingsArray, SERVO_COUNT
                     * sizeof(ServoStatus));
        char* pointer = servoSettingsArray;
        for (int i = 0; i < SERVO_COUNT; i++)
        {
            // must account for sizeof() returning a rounded value (8 bytes) for ServoStatus (7 bytes)
            servos[i] = *((ServoStatus*)((byte *)pointer + ((sizeof(ServoStatus) - 1) * i)));
        }
    }
    catch (int e)
    {
        ROS_ERROR("Error getting channel settings from device.");
    }
}

// obtain just the stack or callstack information from the Mini controller
// stack (true) or callstack (false) is determined by the boolean isStack
// pre conditions: USB has been initialized and at least one Maestro device has been identified
void MaestroController::getVariablesMiniMaestro(short serial, short* stack, bool isStack)
{
    if (isStack) // stack case
    {
        try
        {
            stack = new short[MINI_MAESTRO_STACK_SIZE];
            short* pointer = stack;
            controlTransfer(serial, 0xC0, REQUEST_GET_STACK, 0, 0, pointer, (short)(sizeof(short) * MINI_MAESTRO_STACK_SIZE));
        }
        catch (int e)
        {
            ROS_ERROR("Error getting stack from device.");
        }
    }
    else // callstack case
    {
        try
        {
            stack = new short[MINI_MAESTRO_CALL_STACK_SIZE];
            short* pointer = stack;
            controlTransfer(serial, 0xC0, REQUEST_GET_CALL_STACK, 0, 0, pointer, (short)(sizeof(short) * MINI_MAESTRO_CALL_STACK_SIZE));
        }
        catch (int e)
        {
            ROS_ERROR("Error getting call stack from device.");
        }
    }
}

#include "MaestroController.hpp"

// start up USB interfacing
void MaestroController::initDevices(void)
{
    if (libusb_init(NULL) < 0)
        ROS_ERROR("Error initializing");
}

// close any USB interfacing and stop communications
void MaestroController::closeDevices(void)
{
    libusb_exit(NULL);
}

// searches for all available USB devices
// if any are found that carry the vendor code for Pololu, they are added to a list of controller devices
// as one of the known devices. If an unknown device is found, it is labelled as a Micro
// can print out identification information for all UAB devices if boolean print is true
// pre conditions: USB devices have been initialized
void MaestroController::openDevices(bool print)
{
    libusb_device* dev, **devs;
    libusb_device_descriptor desc;
    libusb_device_handle* hand;
    int i = 0;
    bool search = false;

    if (libusb_get_device_list(NULL, &devs) < 0)
        ROS_ERROR("No devices found");

    while ((dev = devs[i++]) != NULL)
if (libusb_get_device_descriptor(dev, &desc) < 0) {
    ROS_ERROR("failed to get device descriptor");
    return;
}

if (desc.idVendor == VENDOR_POLOLU) {
    libusb_open(dev, &hand);
    // useful for when a device times out
    // resets the device and looks for devices again
    if (libusb_reset_device(hand) == LIBUSB_ERROR_NOT_FOUND) {
        libusb_close(hand);
        search = true;
    }
}

if (search) // if a device is reset, the devices must be found again
{
    libusb_free_device_list(devs, 1);
    if (libusb_get_device_list(NULL, &devs) < 0)
        ROS_ERROR("No devices found");
}

i = 0; // reset from previous while loop
while ((dev = devs[i++]) != NULL) {
    if (libusb_get_device_descriptor(dev, &desc) < 0) {
        ROS_ERROR("failed to get device descriptor");
        return;
    }
    if (desc.idVendor == VENDOR_POLOLU) {
        libusb_device_handle* temp_hand;
        libusb_open(dev, &temp_hand);
        int length = 0;
        char buffer[100];
        unsigned char * p = (unsigned char*)buffer;
        // get serial number to use for identification
        length = libusb_get_string_descriptor_ascii(temp_hand, desc.iSerialNumber, p, 100);
        string serial_number = "";
        for (int i = 0; i < length; i++) {
            serial_number += (char)buffer[i];
        }
        int ser_num = atoi(serial_number.c_str());
        ROS_ERROR("Serial number is %d", ser_num);
        switch (ser_num) {
            case (MICRO_SERIAL):
                primary_dev = dev;
                libusb_open(primary_dev, &primary_hand);
                ROS_ERROR("Micro");
                break;
            case (PRIMARY_SERIAL):
                primary_dev = dev;
                libusb_open(primary_dev, &primary_hand);
                ROS_ERROR("Primary");
                break;
            case (SECONDARY_SERIAL):
                secondary_dev = dev;
                libusb_open(secondary_dev, &secondary_hand);
                ROS_ERROR("Secondary");
                break;
        }
    }
}
default:
    primary_dev = dev;
    libusb_open(primary_dev, &primary_hand);
    ROS_ERROR("Micro default");
    break;
}
}
}
if (print) // print out information on all devices if chosen
{
    ROS_ERROR("%04x:%04x (bus %d, device %d)\n", desc.idVendor, desc.idProduct, libusb_get_bus_number(dev), libusb_get_device_address(dev));
}
libusb_free_device_list(devs, 1); // device list is no longer needed

// send a simple command to the controller
// most useful with servo settings and outputs
// pre conditions: target device has been opened
int MaestroController::controlTransfer(short serial, unsigned char RequestType, unsigned char Request, short Value, short Index)
{
    int ret = 0;
    switch (serial)
    {
    case (MICRO_SERIAL):
        ret = libusb_control_transfer(primary_hand, RequestType, Request, Value, Index, (unsigned char*)0, 0, (short)5000);
        break;
    case (PRIMARY_SERIAL):
        ret = libusb_control_transfer(primary_hand, RequestType, Request, Value, Index, (unsigned char*)0, 0, (short)5000);
        break;
    case (SECONDARY_SERIAL):
        ret = libusb_control_transfer(secondary_hand, RequestType, Request, Value, Index, (unsigned char*)0, 0, (short)5000);
        break;
    }
    if (ret)
    {
        ROS_ERROR("Control transfer failed.");
    }
    return ret;
}

// used for more complex communications with the controller, usually involving obtaining information from it
// mostly used for obtaining variable information, such as inputs and current servo states
// pre conditions: target device has been opened
int MaestroController::controlTransfer(short serial, unsigned char RequestType, unsigned char Request, short Value, short Index, void * data, short length)
{
    int ret = 0;
    switch (serial)
    {
    case (MICRO_SERIAL):
        ret = libusb_control_transfer(primary_hand, RequestType, Request, Value, Index, (unsigned char*)data, length, (short)5000);
        break;
    case (PRIMARY_SERIAL):
        ret = libusb_control_transfer(primary_hand, RequestType, Request, Value, Index, (unsigned char*)data, length, (short)5000);
        break;
    case (SECONDARY_SERIAL):

ret = libusb_control_transfer(secondary_hand, RequestType, Request, Value, Index, (unsigned char*)data, length, (short)5000);
break;
}
//TODO: remove or fix to fit all applications
if (ret == (length - 24))
{
    ROS_ERROR("SERVOS GET!");
    return ret;
}

#include "MaestroController.hpp"

// takes in any number of servos and sets their targets as specified
// contains testing for if a servo is stalling, stopping it after a defined period of time
int MaestroController::moveServos(short* serial, short* channel, short* value, char numServos)
{
    ros::Rate try_rate(2);
    short curr_pos[numServos];//, prev_pos[numServos];
    //char stall_test[numServos];
    //bool indv_moved[numServos];
    //bool moved = false;
    ROS_ERROR("Moving servo");
    //while (!moved)
    //{
        /*for (int i = 0; i < numServos; i++)
        {
            prev_pos[i] = curr_pos[i];
        }*/
        getChannelValues(serial, channel, curr_pos, numServos);
        for (int i = 0; i < numServos; i++)
        {
            ROS_ERROR("Moving servo %d", i);
            //displayStatus(serial[i]);
            //      if (curr_pos[i] == prev_pos[i] && !indv_moved[i]) // test for stall condition
            //      {
            //        ROS_ERROR("Current position %d is same as previous position %d for servo %d", curr_pos[i],
            //        prev_pos[i], i);
            //        if ((++(stall_test[i])) >= STALL_OUT)
            //        {
            //            // modify target by a small amount to let up on stall position
            //            ROS_ERROR("Setting servo to new position %d, currently at %d.", value[i], curr_pos[i]);
            //            // ROS_ERROR("Stall value is %d for servo %d", stall_test[i], i);
            //            // setTarget(serial[i], value[i], channel[i]);
            //            // return i;
            //            // }
            //        }
            //    } else
            //    {
            //        stall_test[i] = 0;
            //    }
            ros::Rate delay(10);
            if (curr_pos[i] == 0)
                curr_pos[i] = 5600;
            ROS_ERROR("Target is %d, current is %d", value[i], curr_pos[i]);
            int diff = value[i] - curr_pos[i];
            ROS_ERROR("Diff is %d, current %d, target %d", diff, curr_pos[i], value[i]);
            bool neg = false;
            if (diff < 0)
                {
                    "MaestroController_servos.cpp"
                    * Created on: Sep 14, 2011
                    * Author: ra23501
                    */
                    #include "MaestroController.hpp"

                    // takes in any number of servos and sets their targets as specified
                    // contains testing for if a servo is stalling, stopping it after a defined period of time
                    int MaestroController::moveServos(short* serial, short* channel, short* value, char numServos)
                    {
                        ros::Rate try_rate(2);
                        short curr_pos[numServos];//, prev_pos[numServos];
                        //char stall_test[numServos];
                        //bool indv_moved[numServos];
                        //bool moved = false;
                        ROS_ERROR("Moving servo");
                        //while (!moved)
                        //{
                            /*for (int i = 0; i < numServos; i++)
                            {
                                prev_pos[i] = curr_pos[i];
                            }*/
                            getChannelValues(serial, channel, curr_pos, numServos);
                            for (int i = 0; i < numServos; i++)
                            {
                                ROS_ERROR("Moving servo %d", i);
                                //displayStatus(serial[i]);
                                //      if (curr_pos[i] == prev_pos[i] && !indv_moved[i]) // test for stall condition
                                //      {
                                //        ROS_ERROR("Current position %d is same as previous position %d for servo %d", curr_pos[i],
                                //        prev_pos[i], i);
                                //        if ((++(stall_test[i])) >= STALL_OUT)
                                //        {
                                //            // modify target by a small amount to let up on stall position
                                //            ROS_ERROR("Setting servo to new position %d, currently at %d.", value[i], curr_pos[i]);
                                //            // ROS_ERROR("Stall value is %d for servo %d", stall_test[i], i);
                                //            // setTarget(serial[i], value[i], channel[i]);
                                //            // return i;
                                //            // }
                                //        }
                                //    } else
                                //    {
                                //        stall_test[i] = 0;
                                //    }
                                ros::Rate delay(10);
                                if (curr_pos[i] == 0)
                                    curr_pos[i] = 5600;
                                ROS_ERROR("Target is %d, current is %d", value[i], curr_pos[i]);
                                int diff = value[i] - curr_pos[i];
                                ROS_ERROR("Diff is %d, current %d, target %d", diff, curr_pos[i], value[i]);
                                bool neg = false;
                                if (diff < 0)
                                    {
diff *= -1;
    neg = true;
}

if (diff > 200) && curr_pos[i] != 0) {
    ROS_ERROR("Large change");
    int end = value[i];

    for (int j = 0; j < diff / 200; j++) {
        if (neg) {
            value[i] = curr_pos[i] - (200 * (j + 1));
        } else {
            value[i] = curr_pos[i] + (200 * (j + 1));
        }
        ROS_ERROR("In loop %d, target is %d", j, value[i]);
        setTarget(serial[i], value[i], channel[i]);
        delay.sleep();
    }
    value[i] = end;
}

setTarget(serial[i], value[i], channel[i]);
// indv_moved[i] = curr_pos[i] == value[i];
// moved = true;
// for (int i = 0; i < numServos; i++)
// {
//     moved = moved && indv_moved[i];
// }
// try_rate.sleep(); // regulate sample rate to 2 Hz
// }
return -1;

// set the target of the specified servo with some limit checking
void MaestroController::setTarget(short serial, short value, short channel) {
    // TODO: change this to fit all servos if needed
    if (value > GWS_MAX)
        value = GWS_MAX;
    if (value < GWS_MIN)
        value = GWS_MIN;

    // if (value == GWS_MAX || value == GWS_MIN)
    // {
    //     ROS_ERROR("Bounds hit, setting to limit");
    // }
    controlTransfer(serial, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, value, channel);
}

// limit the speed of the servo to the specified value
void MaestroController::setSpeed(short serial, short value, short channel) {
    controlTransfer(MICRO_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_SERVO_VARIABLE, value, channel);
}

// limit the acceleration of the servo to the specified value
void MaestroController::setAcceleration(short serial, short value, short channel) {
    controlTransfer(MICRO_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_SERVO_VARIABLE, value, (short)(channel | 0x80));
}

void MaestroController::disableServo(short serial, short channel) {
    controlTransfer(serial, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, 0, channel);
void MaestroController::homeStepper()
{
    stepperAt = getStepperLocation();
    if (stepperAt == STEPPER_HOME - 11)
    {
        ROS_ERROR("At home");
        return;
    }
    else if (stepperAt == NUM_BATTERIES)
    {
        ROS_ERROR("Not home, moving CCW 1 to find heading");
        while (stepperAt == NUM_BATTERIES)
        {
            shiftCCW();
            stepperAt = getStepperLocation();
            if (stepperAt == 0)
            {
                shiftCW();
                shiftCW();
                shiftCW();
                while ((stepperAt = getStepperLocation()) == NUM_BATTERIES)
                {
                    shiftCW();
                }
            }
            ROS_ERROR("Currently at %d, moving to home", stepperAt);
        }
        moveToStation(STEPPER_HOME - 11);
    }
    else
    {
        ROS_ERROR("Currently at %d, moving to home", stepperAt);
        moveToStation(STEPPER_HOME - 11);
    }
}

// move the stepper to the desired station, given current and target
// optimizes movement for limits of CCW motion on station 0 and CW motion on station 7
// assuming cart can move to either end of the track as well
void MaestroController::moveToStation(char target)
{
    if (stepperAt == target)
        return;

    int change = target - stepperAt;
    ROS_ERROR("Need to move %d stations", change);

    if (change < 0)
    {
        moveCCW(target);
    }
    else
    {
        moveCW(target);
    }
    stepperAt = target;
// obtain the current location of the stepper motor based on photo interrupters
// returns the current location or an out of bounds value if not at a single location
char MaestroController::getStepperLocation()
{
    short serials[] = {PRIMARY_SERIAL, PRIMARY_SERIAL, PRIMARY_SERIAL, PRIMARY_SERIAL, PRIMARY_SERIAL,
                      PRIMARY_SERIAL};
    short channels[] = {BATTERY_STATION_0, BATTERY_STATION_1, BATTERY_STATION_2, BATTERY_STATION_3,
                        BATTERY_STATION_4,
                        BATTERY_STATION_5, BATTERY_STATION_6, BATTERY_STATION_7};
    short positions[NUM_BATTERIES];
    getChannelValues(serials, channels, positions, NUM_BATTERIES);
    for (int i = 0; i < NUM_BATTERIES; i++)
    {
        if (positions[i] == ON_INV)
        {
            return i;
        }
    }
    return NUM_BATTERIES;
}

// move the stepper motor counter clockwise to a certain station
void MaestroController::moveCCW(char dest)
{
    // set direction to CCW
    ros::Rate timer(60);
    int count = 0;
    ROS_ERROR("Moving CCW");
    controlTransfer(PRIMARY_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, STEPPER_CCW,
                    STEPPER_DIRECTION);
    while (getStepperLocation() != dest)
    {
        if (count == 2)
        {
            controlTransfer(PRIMARY_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, OFF, STEPPER_STEPS);
            timer.sleep();
            controlTransfer(PRIMARY_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, ON, STEPPER_STEPS);
            timer.sleep();
            count = 0;
        }
        else
        {
            timer.sleep();
            timer.sleep();
            count++;
        }
    }
    ros::Duration(0.3).sleep();
    controlTransfer(PRIMARY_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, STEPPER_CW,
                    STEPPER_DIRECTION);
    while (getStepperLocation() != dest)
    {
        controlTransfer(PRIMARY_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, OFF, STEPPER_STEPS);
        timer.sleep();
        controlTransfer(PRIMARY_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, ON, STEPPER_STEPS);
        timer.sleep();
        timer.sleep();
        timer.sleep();
        timer.sleep();
        timer.sleep();
    }
}
// move the stepper motor counter clockwise to a certain station
void MaestroController::moveCW(char dest)
{
    ros::Rate timer(60);
    int count = 0;
    ROS_ERROR("Moving CW");
    controlTransfer(PRIMARY_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, STEPPER_CW,
                    STEPPER_DIRECTION);
    while (getStepperLocation() != dest)
    {
        if (count == 2)
        {
            controlTransfer(PRIMARY_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, OFF, STEPPER_STEPS);
            timer.sleep();
            controlTransfer(PRIMARY_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, ON, STEPPER_STEPS);
            timer.sleep();
            count = 0;
        }
        else
        {
            timer.sleep();
            timer.sleep();
            count++;
        }
    }
    ros::Duration(0.3).sleep();
    controlTransfer(PRIMARY_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, STEPPER_CCW,
                    STEPPER_DIRECTION);
    while (getStepperLocation() != dest)
    {
        controlTransfer(PRIMARY_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, OFF, STEPPER_STEPS);
        timer.sleep();
        controlTransfer(PRIMARY_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, ON, STEPPER_STEPS);
        timer.sleep();
        timer.sleep();
        timer.sleep();
        timer.sleep();
        timer.sleep();
    }
}

void MaestroController::wiggle(ros::Rate timer)
{
    /*controlTransfer(PRIMARY_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, STEPPER_CW,
                     STEPPER_DIRECTION);
    controlTransfer(PRIMARY_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, OFF, STEPPER_STEPS);
    timer.sleep();
    controlTransfer(PRIMARY_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, ON, STEPPER_STEPS);
    timer.sleep();
    controlTransfer(PRIMARY_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, OFF, STEPPER_STEPS);
    timer.sleep();
    controlTransfer(PRIMARY_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, ON, STEPPER_STEPS);
    timer.sleep();
    controlTransfer(PRIMARY_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, OFF, STEPPER_STEPS);
    timer.sleep();
    controlTransfer(PRIMARY_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, ON, STEPPER_STEPS);
    timer.sleep();
    moveCCW(stepperAt);
    controlTransfer(PRIMARY_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, STEPPER_CCW,
                    STEPPER_DIRECTION);
    controlTransfer(PRIMARY_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, OFF, STEPPER_STEPS);
    timer.sleep();
    controlTransfer(PRIMARY_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, ON, STEPPER_STEPS);
    timer.sleep();
    controlTransfer(PRIMARY_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, OFF, STEPPER_STEPS);
    timer.sleep();
    controlTransfer(PRIMARY_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, ON, STEPPER_STEPS);
    timer.sleep();
    controlTransfer(PRIMARY_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, ON, STEPPER_STEPS);
void MaestroController::shiftCW(void) {
  ros::Rate timer(30);
  controlTransfer(PRIMARY_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, STEPPER_CW,
                   STEPPER_DIRECTION);
  controlTransfer(PRIMARY_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, OFF, STEPPER_STEPS);
  timer.sleep();
  controlTransfer(PRIMARY_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, ON, STEPPER_STEPS);
  timer.sleep();
}

void MaestroController::shiftCCW(void) {
  ros::Rate timer(30);
  controlTransfer(PRIMARY_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, STEPPER_CCW,
                   STEPPER_DIRECTION);
  controlTransfer(PRIMARY_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, OFF, STEPPER_STEPS);
  timer.sleep();
  controlTransfer(PRIMARY_SERIAL, LIBUSB_REQUEST_TYPE_VENDOR, REQUEST_SET_TARGET, ON, STEPPER_STEPS);
  timer.sleep();
}
#ifndef __PELICAN_SIMULATOR_H__
#define __PELICAN_SIMULATOR_H__

// includes
#include <ros/ros.h>
#include <boost/thread/thread.hpp>
#include <geometry_msgs/Vector3.h>
#include <geometry_msgs/Quaternion.h>
#include <wpi_msgs/LandRequest.h>
#include <wpi_msgs/LandConfirm.h>
#include <wpi_msgs/RobotState.h>

// define pi
#ifndef PI
#define PI 3.14159265358979323846
#endif

// define epsilon
#define EPSILON 1E-6

// macros
#ifndef MIN
#define MIN(A,B) (((A)<(B))?(A):(B))
#endif

#ifndef MAX
#define MAX(A,B) (((A)>(B))?(A):(B))
#endif

#ifndef MINMAX
#define MINMAX(A,B,C) MIN((B),MAX((A),(C)))
#endif

#ifndef NORMALIZE
#define NORMALIZE(Z) atan2(sin(Z),cos(Z))
#endif

// enums
eenum PelicanState
{
  CHARGING,
  LAUNCHING,
  FLYING,
  FLYING_WITH_LOW_BATTERY,
  LANDING
};

// namespaces
using namespace std;

// class
class PelicanSimulator
|
public:
  // constructor
  PelicanSimulator();
// destructor
~PelicanSimulator();

private:

// callbacks
void callbackLandRequestMsg(const boost::shared_ptr<wpi_msgs::LandRequest const> &land_request_msg);
void callbackLandConfirmMsg(const boost::shared_ptr<wpi_msgs::LandConfirm const> &land_confirm_msg);

// init functions
void initParams(void);
void initPublishers(void);
void initSubscribers(void);

// robot states
void chargingState(void);
void launchingState(void);
void flyingState(void);
void flyingWithLowBatteryState(void);
void landingState(void);

// pelican thread
void startPelicanThread(void);
void pelicanThread(void);
void stopPelicanThread(void);

// robot state pub thread
void startRobotStatePubThread(void);
void robotStatePubThread(void);
void stopRobotStatePubThread(void);

// ros objects
ros::NodeHandle nh_
ros::NodeHandle private_nh_

// ros param
string uid_
double robot_state_pub_rate_
double pelican_thread_rate_
double flight_height_
double flight_speed_
double max_battery_voltage_
double return_battery_voltage_
double min_battery_voltage_
double battery_charge_rate_
double battery_discharge_rate_

// ros publishers
ros::Publisher robot_state_pub_
ros::Publisher land_request_pub_
ros::Publisher land_confirm_pub_

// ros subscribers
ros::Subscriber land_request_sub_
ros::Subscriber land_confirm_sub_

// threads
boost::shared_ptr<boost::thread> robot_state_pub_thread_
boost::shared_ptr<boost::thread> pelican_thread_
boost::mutex mutex_

// general containers
wpi_msgs::RobotState robot_state_
PelicanState pelican_state_

// volatile bools
volatile bool received_land_request_
volatile bool received_positive_land_confirm_
volatile bool received_negative_land_confirm_
/**
 * \file pelican_simulator.cpp
 * \author Brian J. Julian
 * \version 0.1
 * \date 30 Aug 2011
 * */

#include <pelican_simulator.h>

#define ROS_PKG "pelican_simulator"
define ROS_NODE "pelican_simulator"

PelicanSimulator::PelicanSimulator(void)
    : nh_(),
      private_nh_("~")
{
    // init params
    initParams();

    // init publishers
    initPublishers();

    // init subscribers
    initSubscribers();

    // start pelican thread
    startPelicanThread();

    // start publishing robot state
    startRobotStatePubThread();
}

PelicanSimulator::~PelicanSimulator(void)
{
    // stop all threads
    stopPelicanThread();
    stopRobotStatePubThread();
}

int main(int argc, char **argv)
{
    // initialize ros node handle
    ros::init(argc, argv, ROS_NODE);

    // print start to log file
    ROS_INFO("Started node %s", ROS_NODE);

    // create class instance
    PelicanSimulator pelican_simulator;

    // ros spin
    ros::spin();
// print termination to log file
ROS_INFO("Stopped node %s", ROS_NODE);

// return success
return(0);
}

/**
 * \file pelican_simulator_callback.cpp
 * \author Brian J. Julian
 * \version 0.1
 * \date 30 Aug 2011
 */

// includes
#include <pelican_simulator.h>

// callback for land request message
void PelicanSimulator::callbackLandRequestMsg(const boost::shared_ptr<wpi_msgs::LandRequest const>& land_request_msg)
{
  mutex_.lock();
  received_land_request_ = true;
  mutex_.unlock();
}

// callback for land confirm message
void PelicanSimulator::callbackLandConfirmMsg(const boost::shared_ptr<wpi_msgs::LandConfirm const>& land_confirm_msg)
{
  mutex_.lock();
  if(land_confirm_msg->confirm)
  {
    received_positive_land_confirm_ = true;
  }
  else
  {
    received_negative_land_confirm_ = true;
  }
  mutex_.unlock();
}

/**
 * \file pelican_simulator_init.cpp
 * \author Brian J. Julian
 * \version 0.1
 * \date 30 Aug 2011
 */

// includes
#include <pelican_simulator.h>

// init params
void PelicanSimulator::initParams(void)
{
  // ros params
  private_nh_.param("uid", uid_, string("robot_name"));
  private_nh_.param("robot_state_pub_rate", robot_state_pub_rate_, 1.0);
}
private_nh_.param("pelican_thread_rate", pelican_thread_rate_, 20.0);
private_nh_.param("flight_height", flight_height_, 10.0);
private_nh_.param("flight_speed", flight_speed_, 1.0);
private_nh_.param("max_battery_voltage", max_battery_voltage_, 12.6);
private_nh_.param("min_battery_voltage", min_battery_voltage_, 9.5);
private_nh_.param("return_battery_voltage", return_battery_voltage_, 9.9);
private_nh_.param("battery_charge_rate", battery_charge_rate_, 0.5);
private_nh_.param("battery_discharge_rate", battery_discharge_rate_, 0.1);

// init params
robot_state_.uid = uid_;
robot_state_.translation = geometry_msgs::Vector3();
robot_state_.rotation = geometry_msgs::Quaternion();
robot_state_.rotation.w = 1.0;
robot_state_.battery_voltage = return_battery_voltage_;
pelican_state_ = CHARGING;

// init volatile bools
received_land_request_ = false;
received_positive_land_confirm_ = false;
received_negative_land_confirm_ = false;
}

// init publishers
void PelicanSimulator::initPublishers(void)
{
    robot_state_pub_ = nh_.advertise<wpi_msgs::RobotState>("robot_state", 10, true);
    land_request_pub_ = nh_.advertise<wpi_msgs::LandRequest>("land_request", 10, true);
    land_confirm_pub_ = nh_.advertise<wpi_msgs::LandConfirm>("land_confirm", 10, true);
}

// init subscribers
void PelicanSimulator::initSubscribers(void)
{
    land_request_sub_ = nh_.subscribe(uid_ + string("/land_request"), 10,
           &PelicanSimulator::callbackLandRequestMsg, this);
    land_confirm_sub_ = nh_.subscribe(uid_ + string("/land_confirm"), 10,
           &PelicanSimulator::callbackLandConfirmMsg, this);
}

// charging state
void PelicanSimulator::chargingState(void)
{
    // motion
    // none

    // battery
    robot_state_.battery_voltage += battery_charge_rate_/pelican_thread_rate_;
    robot_state_.battery_voltage = MIN(robot_state_.battery_voltage, max_battery_voltage_);

    // state transitions
    if(robot_state_.battery_voltage > (max_battery_voltage_ - EPSILON))
    {
// reset volatile bools
received_land_request_ = false;
received_positive_land_confirm_ = false;
received_negative_land_confirm_ = false;

// change state
pelican_state_ = LAUNCHING;
}

// launching state
void PelicanSimulator::launchingState(void)
{
    // motion
    robot_state_.translation.z += flight_speed_/pelican_thread_rate_;  
    robot_state_.translation.z = MIN(robot_state_.translation.z, flight_height_);
    // battery
    robot_state_.battery_voltage -= battery_discharge_rate_/pelican_thread_rate_;  
    robot_state_.battery_voltage = MAX(robot_state_.battery_voltage, min_battery_voltage_);
    // state transition
    if(robot_state_.translation.z > (flight_height_ - EPSILON))
    {
        // change state
        pelican_state_ = FLYING;
    }
    else if(received_land_request_)
    {
        // reset volatile bool
        received_land_request_ = false;
        // publish land confirm msg
        wpi_msgs::LandConfirm land_confirm_msg;
        land_confirm_msg.header.stamp = ros::Time::now();
        land_confirm_msg.uid = uid_;  
        land_confirm_msg.confirm = true;
        land_confirm_pub_.publish(land_confirm_msg);
        // change state
        pelican_state_ = LANDING;
    }
}

// flying state
void PelicanSimulator::flyingState(void)
{
    // motion
    // none
    // battery
    robot_state_.battery_voltage -= battery_discharge_rate_/pelican_thread_rate_;  
    robot_state_.battery_voltage = MAX(robot_state_.battery_voltage, min_battery_voltage_);
    // state transition
    if(received_land_request_)
    {
        // reset volatile bool
        received_land_request_ = false;
        // publish land confirm msg
        wpi_msgs::LandConfirm land_confirm_msg;
        land_confirm_msg.header.stamp = ros::Time::now();
        land_confirm_msg.uid = uid_;  
        land_confirm_msg.confirm = true;
        land_confirm_pub_.publish(land_confirm_msg);
        // change state
pelican_state_ = LANDING;
}

else if(robot_state_.battery_voltage < return_battery_voltage_)
{
    // reset volatile bools
    received_positive_land_confirm_ = false;
    received_negative_land_confirm_ = false;

    // publish land request msg
    wpi_msgs::LandRequest land_request_msg;
    land_request_msg.header.stamp = ros::Time::now();
    land_request_msg.uid = uid_;
    land_request_pub_.publish(land_request_msg);

    // change state
    pelican_state_ = FLYING_WITH_LOW_BATTERY;
}

// flying with low battery state
void PelicanSimulator::flyingWithLowBatteryState(void)
{
    // motion
    // none

    // battery
    robot_state_.battery_voltage -= battery_discharge_rate_/pelican_thread_rate_;
    robot_state_.battery_voltage = MAX(robot_state_.battery_voltage, min_battery_voltage_);

    // state transition
    if(received_land_request_)
    {
        // reset volatile bool
        received_land_request_ = false;

        // publish land confirm msg
        wpi_msgs::LandConfirm land_confirm_msg;
        land_confirm_msg.header.stamp = ros::Time::now();
        land_confirm_msg.uid = uid_;
        land_confirm_msg.confirm = true;
        land_confirm_pub_.publish(land_confirm_msg);

        // change state
        pelican_state_ = LANDING;
    }

    else if(received_positive_land_confirm_)
    {
        // reset volatile bool
        received_positive_land_confirm_ = false;

        // change state
        pelican_state_ = LANDING;
    }

    else if(received_negative_land_confirm_)
    {
        // reset volatile bool
        received_negative_land_confirm_ = false;

        // publish land request msg
        wpi_msgs::LandRequest land_request_msg;
        land_request_msg.header.stamp = ros::Time::now();
        land_request_msg.uid = uid_;
        land_request_pub_.publish(land_request_msg);
    }
}

// landing state
void PelicanSimulator::landingState(void)
{
// motion
robot_state_.translation.z = flight_speed_/pelican_thread_rate_;  
robot_state_.translation.z = MAX(robot_state_.translation.z, 0.0);

// battery
robot_state_.battery_voltage = battery_discharge_rate_/pelican_thread_rate_;  
robot_state_.battery_voltage = MAX(robot_state_.battery_voltage, min_battery_voltage_);

// state transition
if(robot_state_.translation.z < EPSILON)
{
   // change state
   pelican_state_ = CHARGING;
}

#include <pelican_simulator.h>

// start pelican thread
void PelicanSimulator::startPelicanThread(void)
{
   pelican_thread_ = boost::shared_ptr<boost::thread>(new
   boost::thread(boost::bind(&PelicanSimulator::pelicanThread, this)))
;
}

// pelican thread
void PelicanSimulator::pelicanThread(void)
{
   // set thread rate
   ros::Rate thread_rate(pelican_thread_rate_);

   // while node active
   while(ros::ok())
   {
      // pelican state machine
      mutex_.lock();
      switch(pelican_state_)
      {
      case(CHARGING):
         chargingState();
         break;
      case(LAUNCHING):
         launchingState();
         break;
      case(FLYING):
         flyingState();
         break;
      case(FLYING_WITH_LOW_BATTERY):
         flyingWithLowBatteryState();
         break;
      case(LANDING):
         landingState();
         break;
      default:
         break;
      }  
   }
mutex_.unlock();

// regulate thread rate
thread_rate.sleep();
}

// stop pelican thread
void PelicanSimulator::stopPelicanThread(void)
{
    pelican_thread_->join();
}

// start robot state pub thread
void PelicanSimulator::startRobotStatePubThread(void)
{
    robot_state_pub_thread_ = boost::shared_ptr<boost::thread>(new
                        boost::thread(boost::bind(&PelicanSimulator::robotStatePubThread, this)));
}

// robot state pub thread
void PelicanSimulator::robotStatePubThread(void)
{
    // set thread rate
    ros::Rate thread_rate(robot_state_pub_rate_);

    // while node active
    while(ros::ok())
    {
        // publish robot state
        mutex_.lock();
        robot_state_.header.stamp = ros::Time::now();
        robot_state_pub_.publish(robot_state_);
        mutex_.unlock();

        // regulate thread rate
        thread_rate.sleep();
    }
}

// stop robot state pub thread
void PelicanSimulator::stopRobotStatePubThread(void)
{
    robot_state_pub_thread_->join();
}

Header header
string uid
bool confirm

Header header
string uid

Header header
string uid
gallery_msgs/Vector3 translation
gallery_msgs/Quaternion rotation
float64 battery_voltage