Team #6

University of Pennsylvania
School of Engineering and Applied Science
Department of Electrical and Systems Engineering

ESE Senior Design

R.A.V.E.N.
Remote Autonomous Vehicle Explorer Network

William Etter
etterw@seas.upenn.edu

Paul Martin
pdmartin@seas.upenn.edu

May 6, 2011

Advisor: Professor Rahul Mangharam

Phase 2 Report

University of Pennsylvania
School of Engineering and Applied Science
Department of Electrical and Systems Engineering

Authors: William Etter & Paul Martin

R.A.V.E.N. – Remote Autonomous Vehicle Explorer Network

Abstract

As robotic platforms and unmanned aerial vehicles (UAVs) increase in sophistication and complexity, the ability to determine the spatial orientation and placement of the platform in real time (also known as localization) becomes an important issue. What is a relatively simple task for a human operator to complete becomes a daunting process for an autonomous platform.

Current methods to achieve localization in UAV systems require computation-intensive sensor systems on the platforms themselves or pre-installed in the location of interest. This not only increases the system cost, but also decreases the overall applicable range of the platform. In addition, no system to control multiple units simultaneously for search and rescue (SAR) exists. This prevents more capable and advanced platforms from being used in situations where they could provide the most benefit.

The system presented uses an on-board near-infrared camera to track beacons on user-controlled quadrotor units. By fusing this data with orientation data that is broadcasted over a wireless network from leader to follower, an accurate position relative to other quadrotors can be determined. Provided with this accurate position relative to the leader, a slave quadrotor can autonomously follow based on the amplitude of its error in three-dimensional space.

The resulting system demonstrates a relative localization scheme with centimeter accuracy at under 3 meters distance and less accurate localization information at larger distances. Under normal conditions, a second follower quadrotor can respond appropriately to its error displacements and track the movements of the lead quadrotor.

Contents

List of Figures

List of Tables

1 Purpose and Scope

Search and rescue (SAR) and surveillance operations require fast responses and broad geographic scopes. In some cases, helicopters and airplanes can be used to monitor a situation or locate a missing person, but in situations of limited aerial sight or where more precise information must be gathered, a more specialized approach is required. Unmanned aerial vehicles (UAVs) have seen increasing use in surveillance environments. Specifically, quadrotor helicopter platforms provide a convenient platform for agile, robust locomotion and long range of operation. Quadrotor helicopters are currently being used by different groups including law enforcement and emergency personnel to respond quickly to traffic accidents or survey an urban landscape. However, whereas a team of individuals might help each other to find a missing person or explore a dangerous situation, current unmanned surveillance systems use only one unit and thus do not benefit from the collaboration and redundancy of a multi-agent team. To combat this, we designed a multiple quadrotor system for search and rescue. The goals of this project were

- To offer a greater breadth of search than current UAV surveillance systems
- To create a wireless telemetry and vision-based leader-follower system to allow for mutli-UAV relative localization

2 Introduction

According to a 2008 report, during urban search and rescue missions there is a narrow 72 hour time frame to rescue a victim before the probability of finding and restoring a victim to full health drops to nearly zero [1]. Unmanned aerial vehicles (UAVs) have recently become a viable platform for surveillance and exploration tasks where human presence is dangerous, impossible, or inadequate. Several commercial quadrotor aircraft (a popular four-rotor vertical take-off and landing (VTOL) vehicle) have been successfully used as surveillance equipment by groups such as United States and Canadian police forces, and it is not difficult to imagine other applications for this burgeoning technology–exploration of radioactive and hazardous material environments, naval search and rescue, or surveying a building on fire, to name a few. Despite the agility and speed of the quadrotor platform, current implementations of the VTOL configuration do not allow for extended flight times (usually upwards of twenty minutes). This short flight time limits the overall flight range as well as the amount of time available to thoroughly explore a specific area. To combat this drawback to quadrotor platforms, we designed a system for intelligently controlling multiple quadrotor UAVs. This system uses a lead, human-controlled quadrotor and one or more quadrotors that track and follow the lead unit autonomously. By using this system, we improve the execution time required to complete missions by utilizing multiple platforms simultaneously. This approach increases both the breadth of search during the mission (since a larger area may be covered in the same amount of time, overcoming the current issues due to battery-life) as well as the platform effectiveness.

When dispatching quadrotor units to a location, the challenge of determining the platform's placement in space with respect to surrounding objects and other platforms (localization) is increasingly important as the simplicity and familiarity of the environment diminish. Methods currently used

require pre-installed systems or a base station, limiting both range and effectiveness in a wide array of surroundings. A cost-effective and manageable solution to this requirement is to use a single human operator to conduct the advanced requirements of simultaneous localization and mapping (SLAM) and effectively "follow" the human-controlled unit through the environment. That is, quadrotors using our system will assume that the human operator is aware of his/her surroundings and follow the "clear" flight path generated. When the desired location is reached, more advanced maneuvers such as searching or environment testing can be achieved with a greater number of vehicles (decreasing total mission time). Overall, this system eliminates the requirement of standard localization equipment or a human operator for each quadrotor, effectively managing resources during a critical mission.

The system we designed has three main subsystems. The first is the lead quadrotor UAV. This is a human-controlled unit that receives commands from and transmits information to the second subsystem–the base station. The base station is composed of a graphical user interface (GUI) that allows the user to see in real-time the view from the lead quadrotor, while controlling it with a wireless radio control unit. Any pertinent environmental information (e.g. infrared detection of heat signatures, radioactivity levels, etc. depending on the sensors installed) is transmitted via wireless RF from the lead quadrotor to the base station. The final subsystem is composed of one or more follower quadrotor UAVs. These are identical to the lead UAV with the exception of additional vision processing hardware. Each follower quadrotor receives inertial measurement unit (IMU) data from the lead and other follower quadrotors. This information, coupled with object tracking information from the front-facing cameras (which provides distance and respective position), allows the follower quadrotors to perform relative localization based on of the coordinates of the lead unit. This data is also sent to the base station where a graphical representation of the quadrotor vehicle network is displayed in addition to the sensor data.

3 Discussion of Previous Work

Localization and environment awareness for robotic and unmanned vehicles are not new subjects by any means. These conditions for automation and advanced control have proven to be difficult challenges to overcome in terms of equipment and operating environment requirements. Previous work that has been done to meet these conditions has resulted in implementations using varying technologies and platforms with strong localization advantages. However, although individual systems have shown strong promise in particular areas, they possess disadvantages that prevent them from being reasonably used in a wide range of applications and environments. The methods commonly used include the Global Positioning System (GPS), laser rangefinders, stationary cameras, received signal strength indication (RSSI), sonar/infrared (IR) sensors, and on-board cameras.

One of the most common methods in use on robotic platforms is the Global Positioning System. Relying on signals from satellites orbiting the Earth, the system triangulates the position of a platform and outputs latitude, longitude, and (on some equipment) altitude. One example of this system installed on a quadrotor platform was accomplished by a group of students at MIT [2]. The quadrotor demonstrated the ability to hold positions as well as autonomously move to waypoints sent remotely over a radio connection. Although this system is inexpensive, has both a small foot-

print and power requirement, and proved useful in an outdoor environment there are limitations to its usefulness. First, the accuracy of GPS systems is limited to several meters. This prevents highly intricate maneuvers and positioning from being available to systems using GPS. In addition, signal blockage due to the environment greatly decreases the operation of the system. For example, usage indoors is prevented since the limited signal strength is not enough to accurately determine the position of the platform. Therefore, this is not a system that should be used in a SAR type situation where urban scenarios are possible.

An on-board system that is gaining popularity in the robotics field is the scanning laser rangefinder, which is also known as light detecting and ranging (LIDAR). Using a rotating laser assembly and a vision capture device (such as a camera optical sensor), the environment surrounding a platform can be mapped out in terms of relative distance. These systems are very accurate and can create three-dimensional layouts of locations. Due to their high refresh rate (sensor update speed) and field of view (FOV), several academic institutions have implemented this system on robotic platforms ranging from ground-based bomb explorers to autonomous quadrotors exploring indoor environments [3] [4]. However, although accurate and powerful, laser rangefinders are expensive, heavy, and power-intensive sensors that are not designed for constrained platform environments such as a quadrotor. Additional thrust and computation requirements decrease already-limited flight times and make this system not particularly useful in SAR situations where as much flight time as possible is needed for increased range of search.

The most powerful systems used for general localizations is the Vicon 3D motion capture system. This system uses multiple, stationary cameras to determine in real time the location of reflectors placed on objects. This results in a very robust and accurate system that can precisely determine the orientation and movement of platforms with attached reflectors. The University of Pennsylvania General Robotics, Automation, Sensing and Perception (GRASP) Laboratory has developed advanced controls for quadrotor platforms using the Vicon system [5]. With feedback from the cameras, Penn's GRASP Lab has been able to develop intricate and innovating controls for their fleet of quadrotors (including perching and flips). The two important downsides to this system include the equipment and the range. The Vicon system requires a large financial investment to obtain the necessary cameras and servers to run the software. This makes it unfeasible for a project aimed at a low-cost localization method. The Vicon system also has a limit to the applicable range, requiring the platform to be within a room where the motion capture system has been installed. Therefore, it is not appropriate for SAR scenarios. A less expensive method similar to the Vicon system is a stationary camera system that uses vision tracking and ranging to determine the location of the platform. Using this design the University of Essex has developed swarMAV, a multiple micro air vehicle (MAV) system designed for collaborative flocking and problem solving [6]. The swarMAV system uses a separate base station unit with infrared ranging accurate to within 1 millimeter. This, however, presents the same limitation inherent in the setup used by Penn's GRASP lab–operation is limited to within range of the base station.

Two inexpensive methods for relative localization that have been used (primarily in conjunction with other techniques) include received signal strength indication and sonar/infrared sensors. Received signal strength indication uses the the strength of a radio frequency (RF) signal to determine approximate distance from the source of the signal. Using this method, robotic platforms are able

to easily implement a basic relative localization scheme that uses equipment already installed. Although techniques using this method have been tested, results have been limited in usable information due to noise, signal blockage, and signal reflection that causes inaccurate measurements [7]. Sonar and infrared sensors are two other inexpensive methods for limited localization. Sonar sensors use ultrasonic pulses and listen for return echoes to determine distance from objects, while infrared sensors emit infrared light and measure the strength of the reflection. Both of these sensors need to be combined with additional sensors or localization techniques for a platform to be autonomous in an environment. However, due to their light weight, low cost, and low power requirements they are useful additional sensors to a platform that needs to work in a wide range of scenarios.

Lastly, on-board camera systems remove the range limitations previously stated in other methods and allow a greater number of applicable environments. On the platform itself, these systems use processed images to visually examine the surroundings, removing the need for expensive, fixed equipment or a base station. An example of this system has been implemented by students at ETH Zurich. Their PixHawk quadrotor uses cameras for simultaneous localization and mapping (SLAM) as well as detection of objects [8]. Currently the PixHawk system has not been extended to relative localization between quadrotors, however this method of SLAM offers potential in multi-quadrotor systems and uses lower cost equipment with improved results.

4 Strategic Plan and Structure

This section contains detailed information regarding the theory driving the system, hardware and software requirements, and the model used to test algorithms before implementing them on physical hardware.

4.1 Theory of Operation

The algorithm that regulates quadrotor helicopter stability is not complicated and follows straightforward logic paths for correcting yaw, pitch, roll, and altitude. Correcting for absolute position requires a more complex method of localization. Quadrotor controls assume the configuration shown in Figure 1.
With this orientation in mind, the following controls were used:

- A positive pitch is effected by decreasing motor 4 and increasing motor 2.
- A positive roll is effected by decreasing motor 1 and increasing motor 3.
- A positive yaw is effected by decreasing one pair of counter rotating rotors (3 and 1 or 4 and 2) and increasing the other. This will depend on which pairs are installed on which motors.

This governs only the stability of the system, which is a foundation for the more complex algorithms. The integration of vision detection, flight trajectory, and the end user interface are depicted in more depth in Figure 2. The lead quadrotor, under the control of a human operator, is depicted

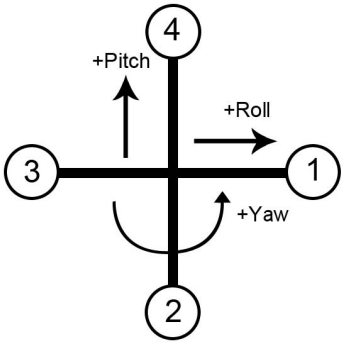


Figure 1: Layout of Four Motors and Attitude Convention

as flying in a "safe zone" which the follower quadrotors can travel within. Using the inertial measurement unit data from the quadrotor directly in front as well as the spatial location and distance information obtained from the camera vision system, the follower quadrotors are able to determine relative location and direction of travel of the quadrotor being followed. This allows the follower quadrotors to allocate resources to the requirements of the mission rather then performing individual localization.

Once one pair of quadrotors (consisting of a leader and follower) was created successfully, the nature of the system allowed for expandability (see Figure 2). The addition of a second follower causes the first follower to take on the role of the new quadrotor's leader.

$$L_1(human\ operator) \Leftarrow F_1$$
$$F_1 = L_2 \Leftarrow F_2$$
$$F_2 = L_3 \Leftarrow F_3$$
$$\dots$$
$$L = Leader \qquad F = Follower \qquad X \Leftarrow Y = Quadrotor\ Y\ follows\ Quadrotor\ X$$

In this way, successive quadrotors follow the previous followers which as a whole follow the flight of the lead quadrotor under human control.

4.2 System Specifications

The multi-quadrotor system was tested against very particular specifications to ensure that the system as a whole remains a viable solution to the problem presented. In particular, the system was designed to

- Retain a flight time adequate for search and rescue operations despite added computation and weight. In particular, the addition of vision processing hardware, high-power wireless communication, and wireless video transmission was to not reduce the flight time to a point such that the system is no longer useful. This was tested and calculated upon completion of all subsystems.

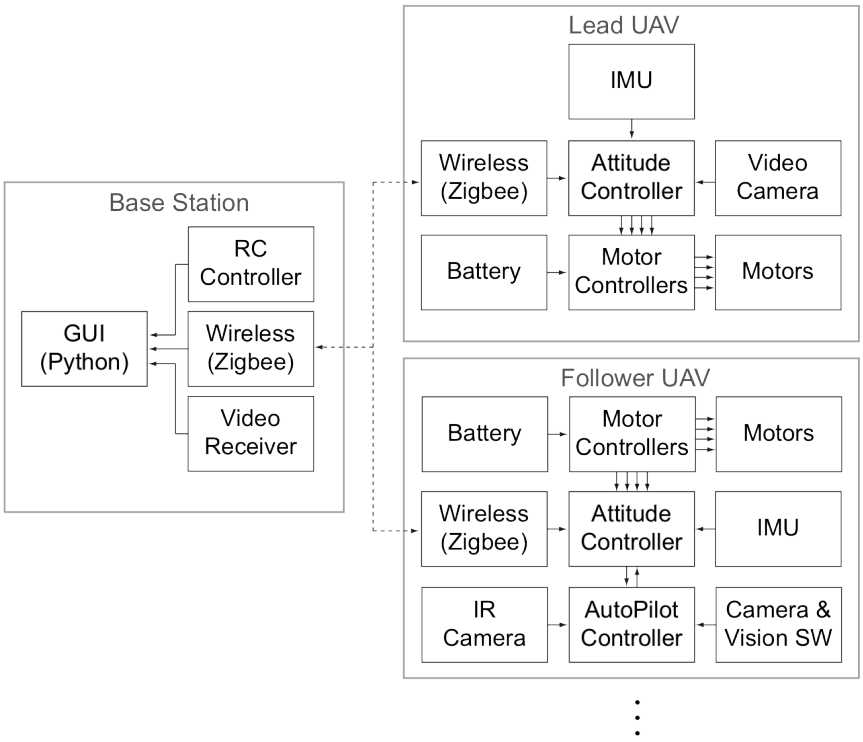


Figure 2: Block diagram for the multi-quadrotor system as a whole. The ellipsis under the follower quadrotor indicates that the system is designed to be expandable.

- Allow for low-maintenance control of multi-quadrotor groups in a way that is *easier* than multiple user-controlled units. The added follower quadrotors should place no additional burden on the user.
- Provide increased breadth of search compared to a single UAV system. With the addition of each quadrotor unit there must be a corresponding increase in the breadth of search.

4.3 Hardware and Software Requirements

Quadrotor UAVs require particular hardware and software in order to remain stable and perform tasks efficiently. Lightweight and highly capable hardware components were chosen, while code was optimized for efficiency and feedback.

4.3.1 Hardware Requirements and Design Approach

Strict hardware specifications were followed in the construction of each quadrotor unit. In particular, items chosen had high bandwidth (to improve stabilization) and low weight (to improve flight time). These requirements also drove the price of the system up, and therefore care was taken to find solutions within budget to retain the viability and capabilities of the final project.

Potential hardware concerns for the system for which additional care was taken in both selection and implementation of the components include (but are not limited to)

- Bandwidth, capability, and accuracy of the inertial measurement unit
- Efficiency and total thrust generated by the motors
- Update frequency of the electronic speed controllers
- Processing power and capability of the flight microcontroller
- Processing power and capability of the camera system
- Size, weight, and structural integrity of the quadrotor frame
- Range and bandwidth of the communication system
- Size, weight, and capacity of the power source

Inertial Measurement Unit

The most important piece of equipment to a quadrotor system is the inertial measurement unit (IMU). This device measures the orientation of the quadrotor in three dimensions as well as the accelerations that the platform experiences. The IMU used in this system is the *CHRobotics CHR-6dm AHRS* IMU. This board uses a dedicated 32-bit ARM Cortex processor that obtains the values from the sensor array. The sensors available on this device include a three-axis accelerometer, three-axis gyroscope, and three-axis magnetometer. The three-axis accelerometer is used to sense accelerations in the x, y, and z axes and can be used to determine orientation with respect to ground by using the earth's gravitational acceleration. However, when the quadrotor itself is accelerating these values are less precise and are therefore used in conjunction with the gyroscopes. A three-axis rate gyroscope is needed to determine change in angles. This is needed for accurate response in the derivative terms of the PID. An alternative method would be to determine change in angle by subtracting the current angle from a past angle using only the accelerometer, but this is slow and inaccurate. A three-axis magnetometer is used for drift correction. By using the magnetometer the quadrotor's absolute orientation with respect to the earth's magnetic field can be measured. For instance, the yaw measurement is 0 degrees when motor four is pointed due north. The processor runs an Extended Kalman Filter (EKF) on the raw sensor data to obtain absolute yaw, pitch, and roll values. The IMU has been programmed to communicate with a baud rate of 115,200 and is connected to the flight microcontroller, where the data is used for stabilization and flight trajectories.

Motors and Props

To achieve flight, motor and prop combinations were selected for the quadrotor system. Brushless DC (BLDC) motors are more efficient and last longer than brushed motors, offering lower cost over the life of the quadrotor and a greater thrust to power ratio. A downside to using this type of motor is that it requires a particular form of dedicated motor controller and is often more expensive than a brushed motor. To ensure that the flight time of each quadrotor unit remains adequately high to allow for broad range of search, care was taken to pick a BLDC motor that offers generous thrust for sufficiently low power. To accomplish this, multiple motors were tested with various propellers to determine the combination that provides the best thrust per current output. The test platform consisted of a laser-cut ABS plastic lever on which the motor was attached. One unit

of thrust produced by the motor resulted in a thrust proportional to the motor thrust and the ratio of the lengths of the lever on either side of the pivot point. Figures 3 and 4 show the results of these tests using an 8" x 3.8 pitch propeller and a 10" x 4.5 pitch propeller. Note that the ordinate shows the *net* thrust, calculated by subtracting the weight of each motor/propeller combination. Each motor-prop combination was only tested up to 3 amps. This is due to the assumption that the power consumption of the final quadrotor will be around or less than 3 amps (determined from previous experience with quadrotor platforms).

Based on these results, both leader and follower quadrotor units were designed to use KDA Hacker Style motors. The estimated flight time for these motors, given a 2.2 amp·hour battery and an 800g total platform mass is calculated as:

$$Time = 2.2\ amp \cdot hour \times \frac{60\ minutes}{1\ hour} \times \frac{1}{(4\ motors \times 2.8\ amps)} = 11.78\ minutes$$

This is shorter than some commercial systems that boast flight times as long as 20 minutes, but is sufficient to demonstrate the advantages of our system, and is not unexpected given the added weight to our system from the additional vision processing and wireless hardware. Specifications of the KDA Hacker Style motors can be found in Table 1.

Table 1: KDA Hacker Style Brushless Motor Specifications

Voltage	$11.1V$
RPM/V	1130
Max Current	15 A
Weight	$43g$

Electronic Speed Controller
Since this system uses BLDC motors, more advanced motor controllers were required than simple H-bridge controllers. Brushless electronic speed controllers (ESCs) were used to allow motor control using a convenient PWM signal. Each ESC produces the necessary sinusoids to drive the brushless motors at the necessary current. Using back-EMF, the motor controller is able to determine the current phase of the motor and apply the correct voltage in sequence and thereby commutate the motor. An issue for quadrotors with ESCs is the motor update rate. Remote control aircraft ESCs are designed to accept PWM signals at 50Hz (the standard for RC units is a 50Hz signal with duty cycles between 1ms and 2ms). This update frequency is adequate for basic maneuvers such as taking off, hovering, and landing. However, to maintain solid stability and perform more complicated flight maneuvers, the ESCs need to update the motors at a frequency greater than 50Hz. The ESCs (Table 2) used in this system have been modified to accept PWM frequencies up to $\approx$450Hz. This will allow the system stable quadrotor platforms on which to compute flight trajectories and vision detection.

Flight Microcontroller
The flight microcontroller is responsible for quadrotor flight stabilization as well as controlling

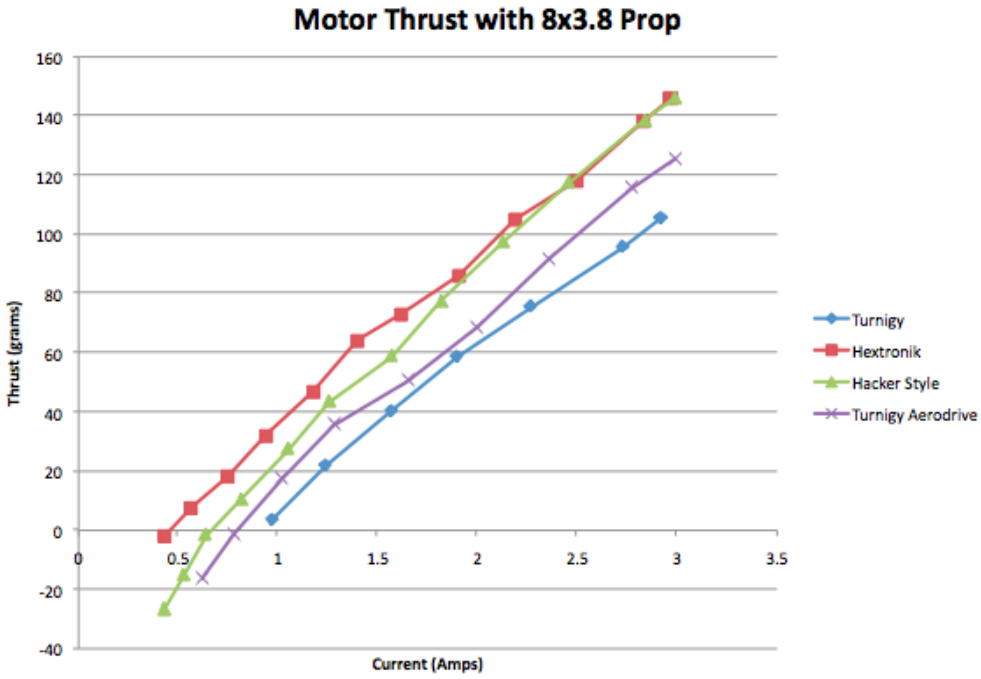


Figure 3: Thrust test for 8.0x3.8 propeller (8.0” diameter and 3.8” pitch).

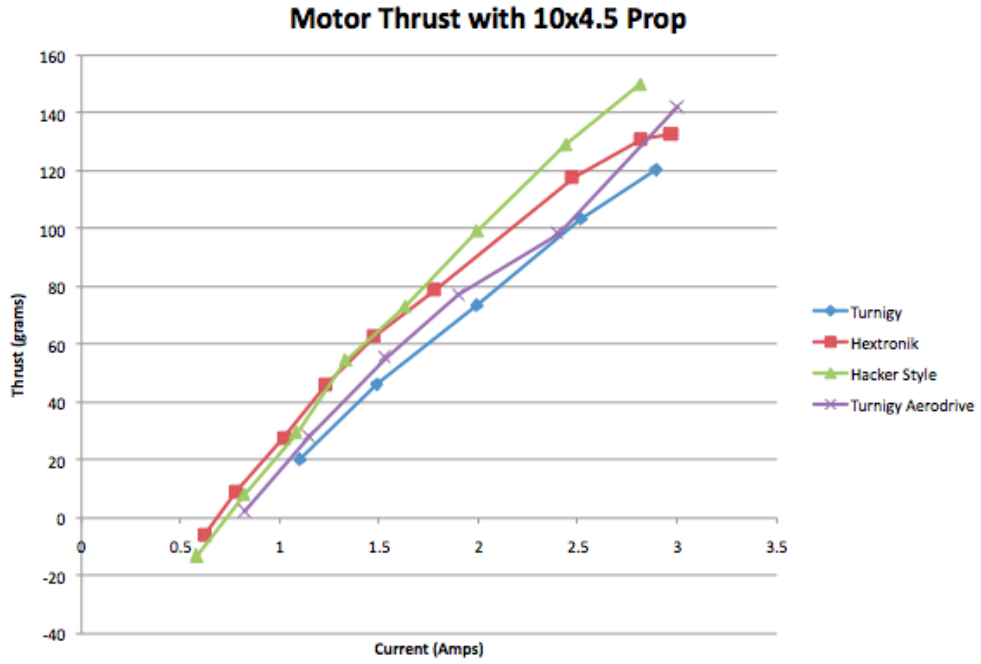


Figure 4: Thrust test for 10.0x4.5 propeller (10.0” diameter and 4.5” pitch).

Table 2: HobbyKing 18-20A Brushless ESC Specifications

Voltage	$11.1V$
BEC	$5V$ at $2A$
Max Current	$18\ A$
Max Current Burst	$20\ A$
Weight	$19g$
Dimensions	45 x 25 x 4 mm

wireless communication. On the lead quadrotor, the flight microcontroller takes inputs directly from a remote control (RC) unit. On the follower quadrotors, the inputs are received from the camera system. The flight microcontroller is connected to the inertial measurement unit and receives data regarding the attitude of the quadrotor. Using these values and software PID control, PWM outputs to the four brushless motor controllers are varied in order to maintain stability and achieve flight in the desired manner. The flight microcontroller used for this system is the open-source *ArduPilot Mega*. This microcontroller allows for eight input and eight output PWM channels, several I/O channels with ADC, as well as four serial communication channels to send/receive data for telemetry or for communication with additional sensors. These capabilities provide room for improvement and growth to the platform.

Camera System
An important system required for this quadrotor platform is the camera vision system. Fiducial (distinct figures from which orientation can be determined) and/or beacon markers (lights or shapes from which direction and distance can be determined) installed on the platform allow follower quadrotors to compute the relative location of the leading quadrotor. Using functions such as pixel detection and frame differencing, the direction (when used in conjunction with the inertial measurement unit data) as well as the distance (i.e. pixel detection of the beacon of known size) can be used together to obtain detailed information regarding the leading quadrotor's current and future position/movement. The overall accuracy of the vision system is dependent on the resolution and video processing capabilities of the camera used. Therefore, the most capable camera within the budget allowance will provide the best experimental results in allowing the followers to track the flight path of the leader.

Frame
A frame that is light, sturdy, and capable of housing the components securely was required for this system. The frame had to be be able to stand up to take-off and landing maneuvers, vibrations caused by the rotation of the motors/props, and have ample room to hold electronic equipment and sensors. Constructed of aircraft plywood and fiberglass landing struts, the selected frame (Table 3) for the quadrotor system is designed to meet these requirements and is large enough to effectively hold future expansions to the onboard equipment. This frame (Q4-600) was designed and manufactured by *Sigma*.

Communication
To allow for data communication between the quadrotors and the base station (such as IMU data,

Table 3: Quadrotor Frame Specifications

Main Body	Air Laminate Aircraft Plywood
Landing Struts	Fiberglass
Weight	220 g with terrace, 180 g without terrace
Height	185 mm
Ground Clearance	100 mm
Wingspan	500 mm

visual data, GPS, and other systems) a wireless communication system was used. Requirements for wireless communication included long ranges and high data bandwidth. The hardware used in this project was the *XBee Pro* wireless platform. This system is running at 900MHz and 50mW (+17dBm), allowing up to 156 Kbps data rates and range greater than one mile. The wireless board with an external antenna was used on the base station, and a wireless board with wire antenna was used on the quadrotor to save weight.

Power Source
Power source characteristics that were important to the system included the weight, amp·hours rating, and output current. The power source used in this system is a 3-cell ($11.1V$) Lithium Polymer (LiPo) battery rated for 2200 mAh and 25C (Table 4). These ratings indicate that the battery can provide 25×2.2 amps, or a little over 50 amps, of continuous current as well as three-second peaks of up to 100 amps. To charge the batteries, a high-end LiPo battery charger with on-board balancer is used. Balancing the LiPo cells equalizes the voltage and results in both longer run-time and battery life. The charger is able to handle several types of batteries (LiPo, LiIon, LiFe, NiCd, and NiMH) as well as 1 to 6 cells. The charger can charge from 0.1 to 5 amps and discharge (to give a fresh charge to a battery) from 0.1 to 1 amp.

Table 4: 3-Cell LiPo Battery Specifications

Voltage	11.1 V
Cells	3
Capacity	2200 mAh
Max Continuous Discharge	25C (50A)
Max Burst Discharge	50C (100A)
Weight	152g
Dimensions	24.4 x 35 x 103 mm

4.3.2 Software Requirements and Design Approach

Three distinct software regimes were used in this project: (1) Control, communication, vision processing, and flight extrapolation performed on each quadrotor, (2) communication and graphical user interface on the base station, and (3) a simulated model to safely design flight extrapolation algorithms before porting to the physical hardware. The first two are enumerated in more detail as

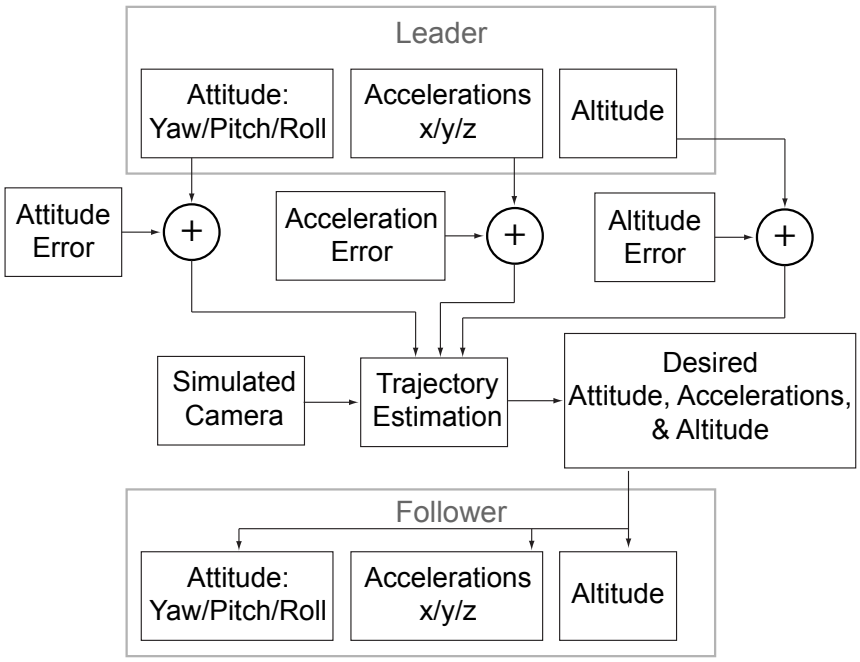


Figure 5: Block diagram for simulation, showing simulated noise and logic flow.

follows:

Quadrotor Stabilization, Control, and Communication:

- Attitude and altitude stabilization
- Robust wireless communication
- Vision processing
- Flight trajectory estimations

Base Station Communication and Interface:

- Robust wireless communication
- Graphical user interface and display of gathered data

MATLAB model

Figure 5 shows the general flow of the MATLAB model. This model takes in a set of coordinates through which the lead quadrotor will fly. At discretized points in time, the lead quadrotor will transmit attitude information (both orientation and three axes of acceleration) to all follower quadrotors. These data are subject to a variable amount of noise and loss rate, as would be seen in the real system. On receiving attitude data, the follower quadrotor takes a snapshot using its simulated camera and, based on both object tracking information generated from vision processing

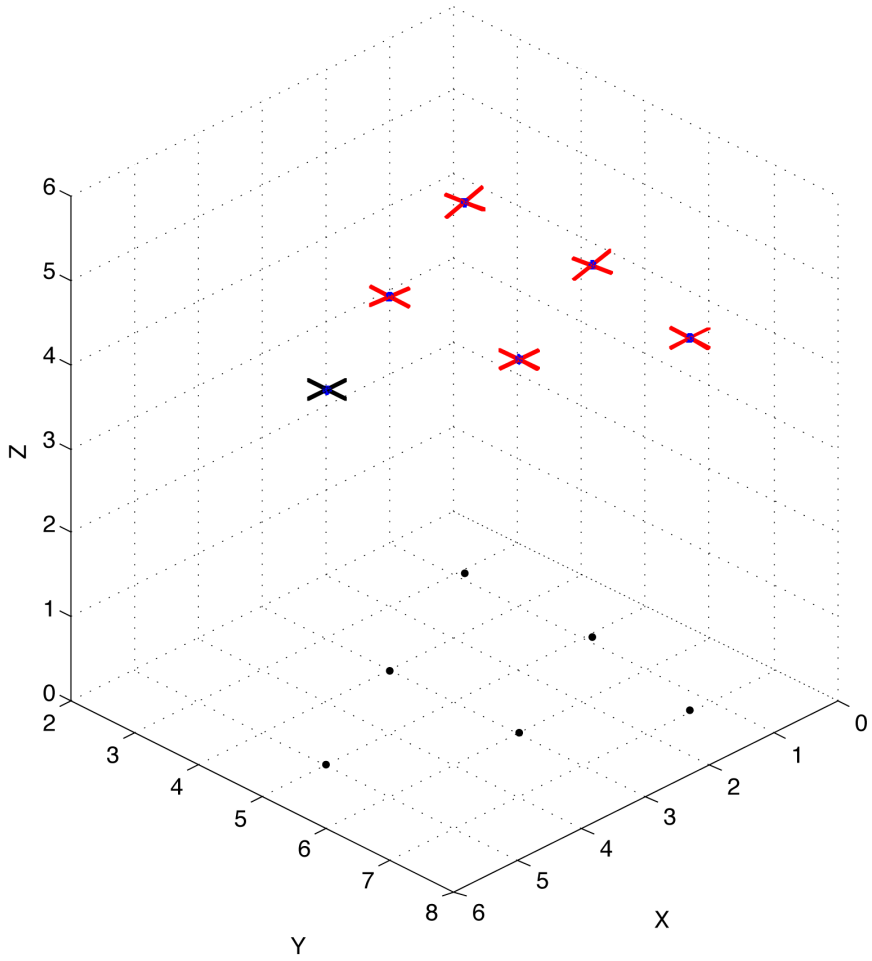


Figure 6: Recursively defined flocking model in MATLAB, showing 2 followers following one leader, and three followers following those two.

and dead-reckoning from transmitted attitude information, adds to a buffer of the leaders estimated trajectory. Then, the follower quadrotor assumes an overall thrust and orientation to arrive at the next point in the leader's trajectory. Simulated perspective views can be seen in Figure 6.

Before implementing vision tracking algorithms on an autonomous UAV with no user input, it was useful to model the system in software. Towards this end, a linear MATLAB model was used to simulate quadrotor physics, object-tracking cameras, RF communication, and proportional–integral–derivative (PID) controls. The model simulates individual quadrotor objects as a point mass, allowing easy creation of additional quadrotor objects along with arbitrary definition of leader-to-follower coordinates and thus an arbitrary flock formation as shown in Figure 6.

Controlling the Leader

The lead quadrotor was controlled by a user-provided event array, indicating desired attitude and thrust for a specified time. This event script, shown in Equation 1, simulated an RC controller and allowed the trajectory of the lead quadrotor to be changed easily and realistically. The attitude was

specified using Euler angles, with ψ, θ, and ϕ as yaw, pitch, and roll, respectively.

$$events = \begin{bmatrix} t_1 & thrust_1 & \psi_1 & \theta_1 & \phi_1 \\ t_2 & thrust_2 & \psi_2 & \theta_2 & \phi_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ t_n & thrust_n & \psi_n & \theta_n & \phi_n \end{bmatrix} \tag{1}$$

Virtual PID

To achieve the desired attitude commanded in the matrix in Equation 1, a virtual PID controller was used. With a time step δt of 0.1 seconds, the values for K_P, K_I, and K_D that resulted in a realistic transient response (decidedly 1 second rise time with less than 10% overshoot) were empirically determined to be 20, 0, and 0.6. These gains molded the response of the angular accelerations, $\dot{p}$, $\dot{q}$, and $\dot{r}$, which were then integrated to determine the angular velocities and the angles themselves.

Virtual Camera

To accurately simulate the conditions faced when implementing this system on physical hardware, it was useful to limit the information available to the follower quadrotor(s). To simulate an on-board camera, the leader's coordinates in world space were transformed via rotation about the unit's angles, as shown in Equation 2, where x_L and x_F indicate the x coordinate of the leader and follower, respectively.

$$\begin{bmatrix} x_{cam} \\ y_{cam} \\ z_{cam} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\phi) & -sin(\phi) \\ 0 & sin(\phi) & cos(\phi) \end{bmatrix} \times \begin{bmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{bmatrix} \times \begin{bmatrix} cos(\psi) & -sin(\psi) & 0 \\ sin(\psi) & cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_L - x_F \\ y_L - x_F \\ z_L - x_F \end{bmatrix} \tag{2}$$

By simulating a virtual camera with the linear transformation given in Equation 2 we can determine whether or not a given quadrotor is in the field of view (FOV) for any of the follower quadrotors. This is trivial, and assuming a camera with aspect ratio 1:1 can be determined by Equation 3, where x is the forward-facing axis, y is the right-facing axis, and z is the up-facing axis.

$$\theta_{object} = tan^{-1}\frac{2z_{cam}}{x_{cam}} = tan^{-1}\frac{2y_{cam}}{x_{cam}} \tag{3}$$

PID In-flight Stabilization of Quadrotor

PID, or proportional-integral-derivative controllers, are loop feedback controls that aim to correct for a given error. Often, the effect of a PID is modeled as a step response. One PID control is needed for each parameter to be corrected. In this case, PID loops for yaw, pitch, and roll are

needed to meet the minimum requirements for a stable system. Additionally, PID loops are used to keep the altitude and planar position in check. The proportional (P), integral (I), and derivative (D) terms are defined below.

$$P = desired - actual$$
$$I_{current} = I_{previous} + actual$$
$$D = current\_error - past\_error$$

This gives a software implementation of the derivative term, but in practice it is faster and more reliable to use the direct output of the rate gyroscopes on the inertial measurement unit. Each of the three terms has a gain K_P, K_I, or K_D that is tuned to minimize rise time to the step input and to minimize oscillation and overshoot. For each of the three Euler angles the sum comprised of the gains and the corresponding terms is computed. For example

$$pitch\_sum = K_P * P + K_I * I + K_D * D$$

At the end of the PID loop, the appropriate summations are added and subtracted from each of the four motors as seen in the code snippet below. The thrustFloor term corresponds to the desired overall thrust.

```
PWM_MOT1 = thrustFloor - roll_sum + yaw_sum;
PWM_MOT2 = thrustFloor + pitch_sum - yaw_sum;
PWM_MOT3 = thrustFloor + roll_sum + yaw_sum;
PWM_MOT4 = thrustFloor - pitch_sum - yaw_sum;
```

The entire PID loop must be completed at a relatively high frequency (approximately 200Hz or greater) to ensure fast motor response. Without a fast PID response the quadrotor will not properly stabilize

4.4 Test and Demonstration

4.4.1 Test

Verifying the various hardware and software components of the system was not overly difficult: Such metrics as bandwidth, flight time, and wireless throughput were logged and measured in software.

Verifying that the system created offers a benefit over a single UAV system was nontrivial. An ideal comparison of the two systems would entail direct comparison between the proposed multi-quadrotor system with a system using only a single quadrotor and a system using two, independently controlled quadrotors. A quantitative comparison in this regard is difficult to obtain without the presence of expensive localization equipment. Instead, towards measuring the efficacy of the flocking system, the accuracy of the localization and tracking software was tabulated.

In short, those metrics used to verify the success of the system were:

1. Flight time
2. Wireless throughput
3. Payload
4. Localization accuracy

4.4.2 Demonstration

The final demonstration of this project showed a quadrotor tracking a lead, user-controlled quadrotor using only on-board sensors and localization techniques. The lead unit was controlled to fly within the bounds of an enclosed canopy, and the canopy was surrounded by a protective net to prevent the blades from coming dangerously close to a person. In addition to a functional display of the working system, pre-recorded flight videos of the system operating in a confined area such as a hallway, along with an animation of what the user saw during the entirety of this episode were shown. Plots were provided in order to demonstrate the accuracy of the physical system as well as the expandability of the algorithms in simulation.

4.5 Schedule

Due to the complexity of this project, care was taken to maintain the project schedule as small slips can compound and cause considerable delays. In order to ensure that the project remained on-task, a Gantt chart was developed. The compressed schedule is shown in Figure 7. The expanded schedule is found in Appendix 1.

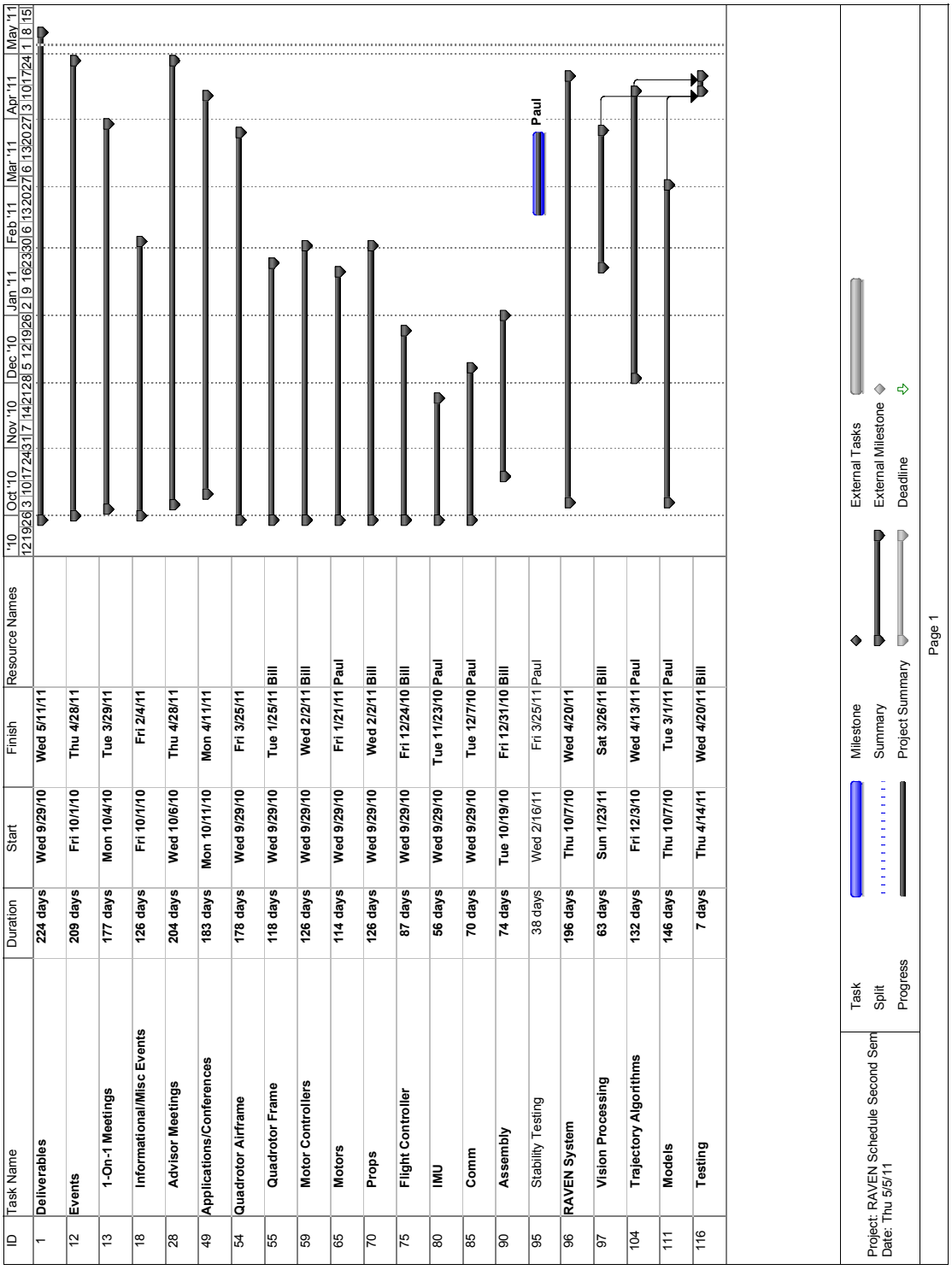


Figure 7: Compressed Gantt chart showing project schedule.

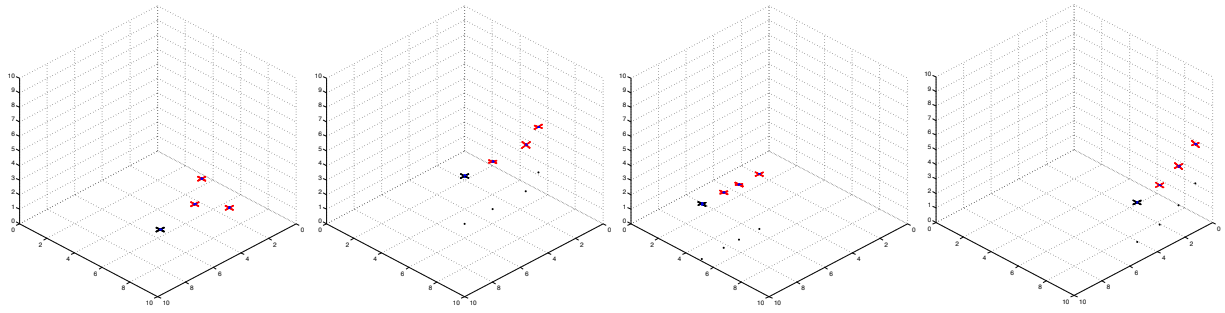


Figure 8: Snapshots of the MATLAB leader-follower model with three followers responding to initial thrust, a forward pitch of 10 degrees, a backwards pitch of 10 degrees, and then a roll left of 10 degrees. Times are (from left to right) 0, 3, 8, and 13 seconds.

A project scheduling issue that was found is the inventory availability of components used in the system. One overseas supplier placed the brushless motors and motor controllers on backorder for approximately three weeks. However, these items arrived and were successfully implemented in the airframe for the quadrotors.

5 Results

The R.A.V.E.N. system proved a very successful platform; two quadrotor helicopters were fully constructed and programmed. Independently, each quadrotor flew in a stable manner, and together a very accurate 3D topology was obtained, allowing relative localization and flocking to be implemented with high speed and little overhead. What follows is a summary not only of the final physical system but of the insight gained from the initial software model as well.

5.1 System Model

With the MATLAB model, the system algorithms can be tested against specific real-world circumstances before they are implemented on the physical system. Specifically, potentially problematic non-idealities that exist in the real system include lost wireless packets from the leader to the follower containing IMU data, noisy IMU data, and noisy or unavailable camera data.
Noisy attitude information from the IMU will affect the stability of each individual quadrotor, but reducing or accounting for this noise is not directly tied with the success of the leader-follower algorithm used. However, the current infrared camera setup does not calculate the relative yaw of the leader and therefore must rely on accurate RF transmission of the leader's yaw angle. Simulated lost packets and erroneous yaw data are lumped into a pseudo-random error between $-\alpha$ and $+\alpha$, where α is the maximum error (in degrees) in the calculation and transmission of yaw. Additionally, a pseudo-random error is introduced on all three axes of the infrared camera, allowing the user to specify in centimeters the maximum camera error.

Each follower quadrotor is given specific following coordinates with respect to the leader. In order to achieve these desired coordinates, a second PID loop is used on top of the stability PID. Gains are empirically chosen first to match altitude (z) by commanding a thrust and then to match x

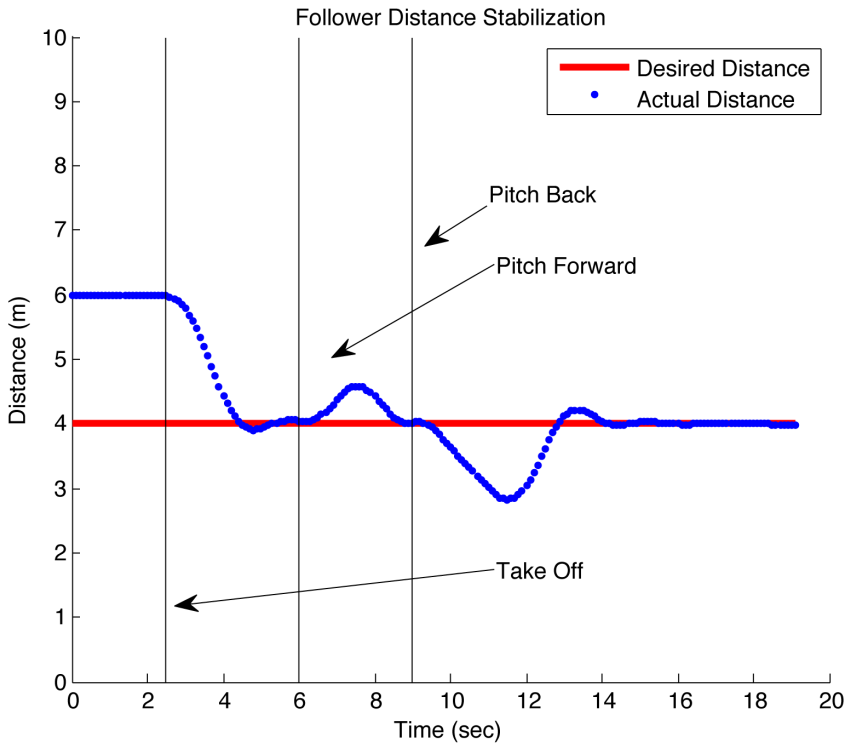


Figure 9: Accuracy of tracking algorithm along forward-facing vector with scripted "leader" events

and y by commanding a pitch and a roll. Information about the leader yaw is not parsed in the camera transform, nor is it on the experimental platform. Instead, the model creates a virtual RF link between the leader and the follower(s), allowing the follower access to the yaw information of the leader with some pseudo-random noise. Values used for the tracking PID are: $K_P = 23$, $K_I = 0$, $K_D = 18$, $K_{P-THRUST} = 100$, $K_{I-THRUST} = 4$, and $K_{D-THRUST} = 6$. A non-zero $K_{I-THRUST}$ is needed to offset steady state error in altitude matching. A timestamped visualization of the model is shown in Figure 8.

For the following simulations, the event matrix (shown in Equation 1) is defined to give the lead quadrotor a 10° pitch forward, a 10° pitch backward, a 10° roll left, a 10° roll right, and finally a 90° yaw and 10° pitch forward. Only one follower is used, and the desired x, y, and z positions with respect to the leader are 4, 0, and 0 meters. The time step used for all simulations is $\delta t = 0.1$ seconds. Figure 9 shows the response of the follower's tracking PID loop with the initial thrusts and pitches of the leader as defined by the event matrix.

By defining θ_{object} as the angle that the object seen by the camera makes with the forward-facing vector of the follower, we can determine if the object is within the camera's FOV. This allows the model to simulate the leader-follower tracking algorithm for various FOV values as well as handle cases in which line-of-sight is lost.

Figure 10 shows the result of plotting versus time the total error (calculated as the sum at each time step of the instantaneous error, shown in Equation 4, where N is the total number of followers)

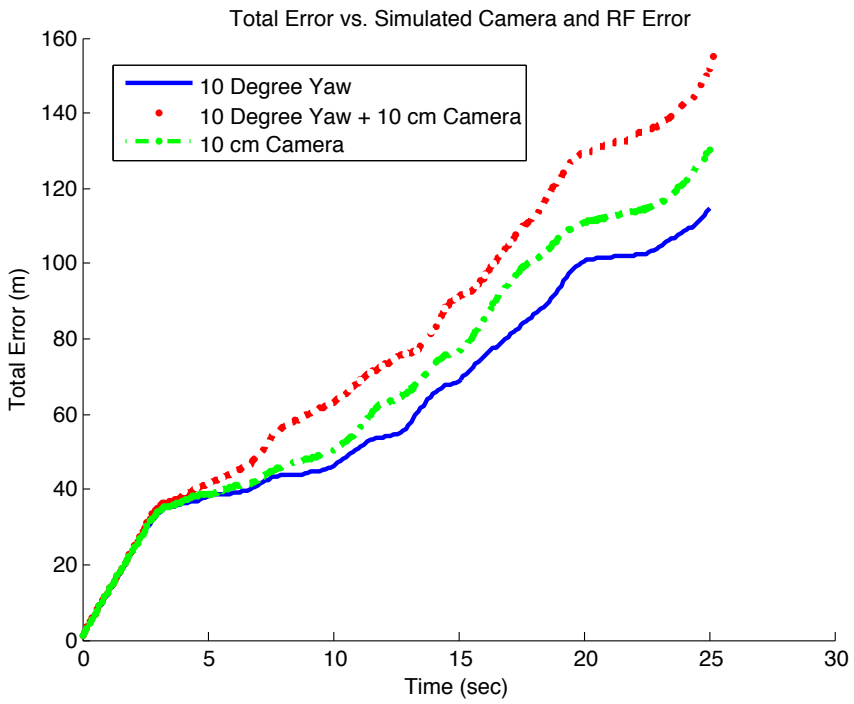


Figure 10: Accuracy (total error) vs. simulated pseudo-random errors

for the cases of a ± 10 degree yaw error, a ± 10 centimeter camera error, and the accumulation of both errors. As shown, the success of the algorithm is affected more by a 10 cm camera error than a 10 degree yaw error, and naturally the aggregate of the two is worse still. However, though the camera sees the leader with only ± 10 cm accuracy, the aggregate error over 30 seconds at $\delta t = 0.1$ intervals is only around 40 meters off of the ideal case, meaning that the worst case error is 40 meters / 10 steps = 4 meters total in all three axes, or roughly 1.3 meters in each of x, y, and z for the worst case.

$$error = \sqrt{\sum_{i=1}^{i=N} [(x_{des} - x)^2 + (y_{des} - y)^2 + (z_{des} - z)^2]} \tag{4}$$

More dramatic than the effect of simulated errors on the system is the effect that the FOV of the follower's camera has on the success of the system. For the same event matrix used for the previous test, the total error is plotted versus time for various simulated field of views. A quadrotor unit is commanded to hover if it loses sight of the leader, and it will continue following should line-of-sight return. Figure 11 shows the points at which the follower quadrotor loses sight for 20°, 30°, and 50° field of views. As shown, a follower with a 20° FOV already loses sight after the initial thrust and does not recover. A follower with a 30° FOV successfully completes thrust, pitch, and roll following (for the 10° event cases), but loses sight during the 90° yaw and does not recover. However, for a follower with a 50° FOV, the leader remains within line-of-sight. Many commercial cameras are within this FOV constraint, but lower FOV cameras may still be sufficient if the maximum yaw rate and angles are constrained.

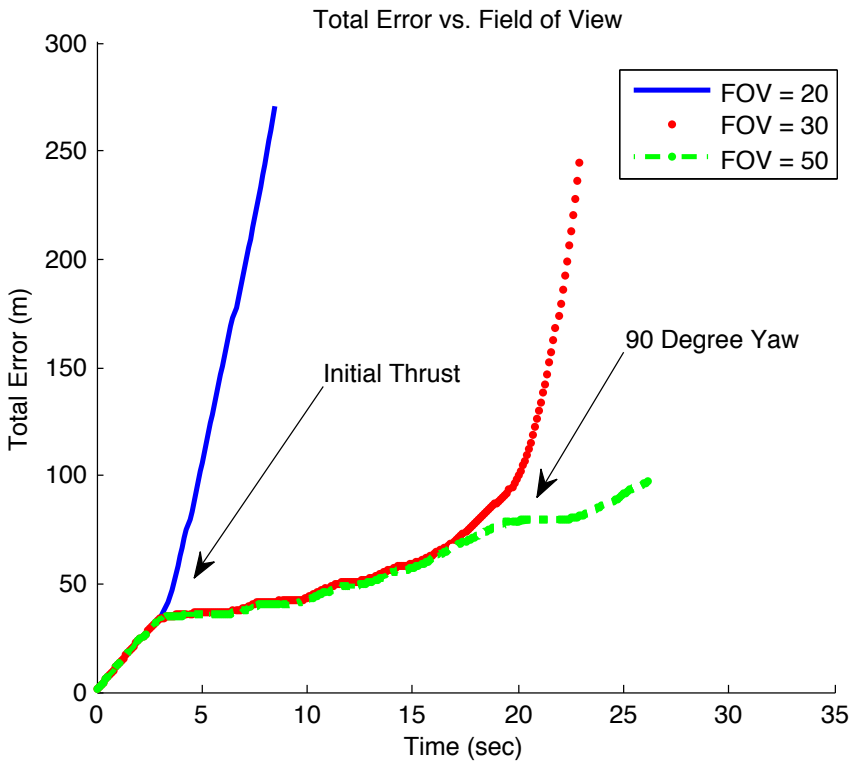


Figure 11: Accuracy (total error) vs. field of view of follower camera, given scripted "leader" events

5.2 Graphical User Interface

Because the end product–a multi-quadrotor search and rescue system–targets an end-user, we developed an appropriate user interface. The initial design and implementation of this user interface included real-time display of yaw, pitch, and roll angles in addition to battery level and altitude for a leader and a single follower. Figure 12 shows the initial design in MATLAB. Eventually, the MATLAB user interface proved too slow, and a second user interface was designed using Python that allowed for real-time streaming of data such as the 3D localization metrics and wireless throughput data. The second and final user interface can be seen in Figure 13.

5.3 Unit Performance

Each quadrotor unit met the desired specifications. The final quadrotor unit pair can be seen in Figure 14. Each R.A.V.E.N. quadrotor was designed for and successfully met the following performance marks:

1. Flight time $>$ 10 minutes. The estimate flight time of each final unit is around 14 minutes
2. Payload for additional sensors. The estimated payload of each unit is around 800 grams–enough to carry 4 additional batteries of the same size.
3. Stable flight. Each unit self-stabilizes in the presence of external disturbances such as wind.

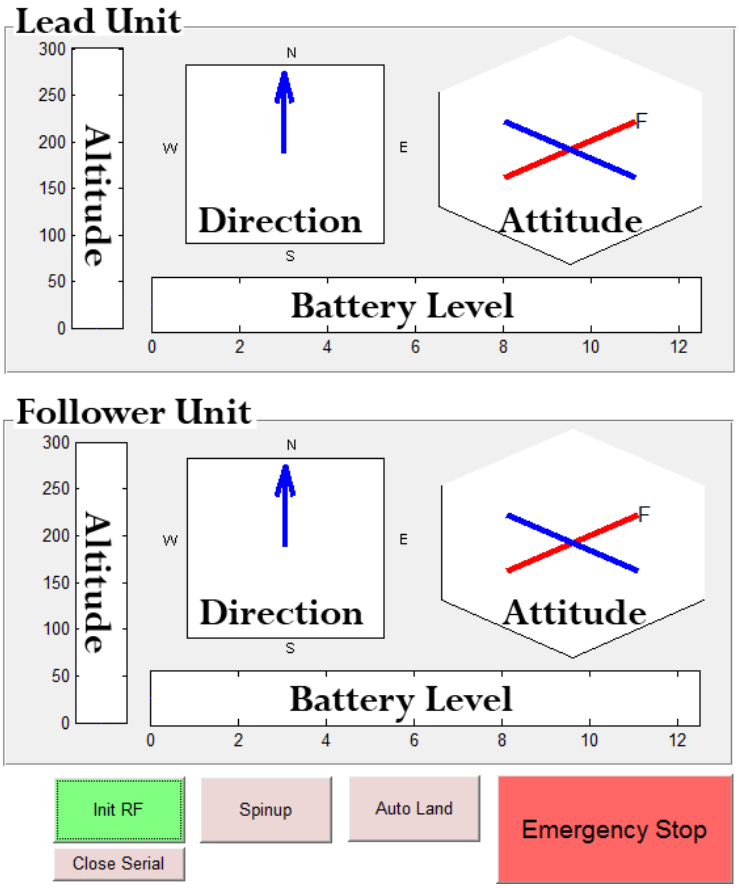


Figure 12: Initial system GUI designed in MATLAB

5.4 Network Architecture

The system network architecture is designed such that the individual components are insusceptible to common sources of interference and maintain a level of robustness against data loss. That is, the three main data communication channels (the remote control for the leader, the video transmission from the leader, and the inter-node data) are separated in channel and/or frequency. The RC and video transmissions are directional point-to-point, however the primary data setup is an ad-hoc network between the nodes. This provides system stability as the network is decentralized and not dependent on the operation of any individual node. This setup also allows for more complex configurations, such as packet routing back to the base station in situations when certain nodes are out of range.

5.5 Sensor Fusion

An integral component of the quadrotor leader-follower system is the fusion of data from the separate sensors on the dedicated microcontroller. This provides two main data types—node tracking and environmental data. The node tracking data is created from the combination of the IR camera and inertial measurement unit (IMU) data from each leader/follower pair. That is, the IR point data from the camera is transformed using the attitude data of both the follower and leader nodes. This

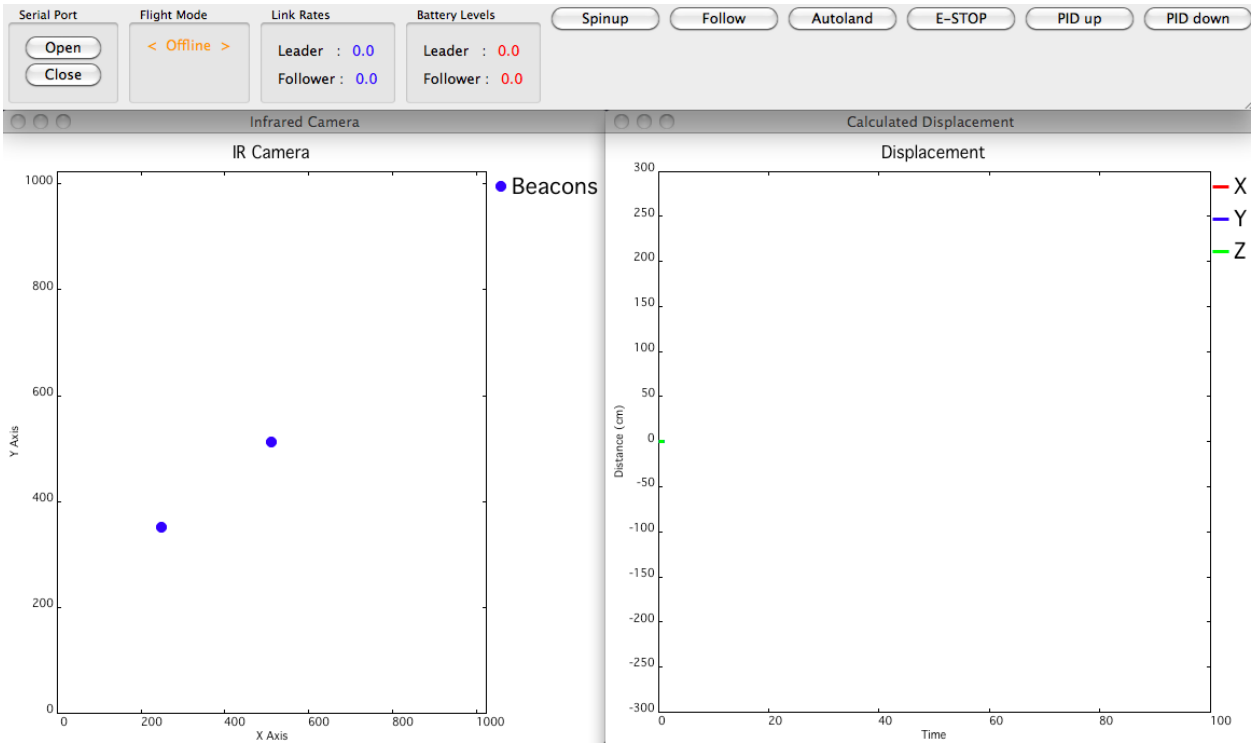


Figure 13: Final system GUI designed in Python, showing battery information, wireless throughput metrics, beacon locations, and 3D localization data.

provides the orientation-corrected IR point-width information which relates the relative spatial positioning of the quadrotors. The position of the IR points in relation to the major axes provides vertical and horizontal information (up/down and left/right) while the fixed width between the IR points provides the distance information (forward/backward). In this way, the data from three sensors (one of which is physically separated from the others) is combined to generate the three-dimensional tracking data required for flight following (Figure 15). An example of the infrared beacons as perceived by the near-infrared camera can be seen in Figure 16.
Environmental data is also processed by the sensor microcontroller. The range data from the ultrasonic sonar sensors directed along each axis of the quadrotor provides distance data to obstacles. When processed, this data provides flight feedback to prevent the nodes from maneuvering into blocked locations. Additionally, the suite of sensors installed on the nodes provide collaborative information to system users. For example when combined with the sonar data from multiple nodes the information from a thermopile array can provide location information for sources of heat such as people or fires and atmospheric sensors (such as carbon-monoxide sensors) can map locations presenting hazards. This results in the system assisting rescuers with mission-critical data that increases their effectiveness and efficiency.

5.6 Communication and Protocols

The platform created has three distinct wireless communication links. First, a 2.4 GHz commercial RC controller is used to command a total thrust as well as desired yaw, pitch, and roll angles.

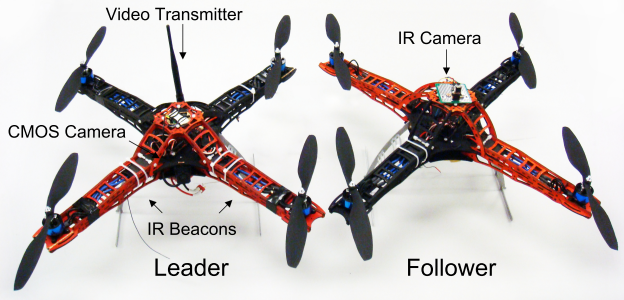


Figure 14: Experimental Quadrotor Platform (Leader and Follower Units)

Second, a 1.2 GHz commercial wireless video camera and transmitter is used on the lead quadrotor unit alone for the purpose of maintaining a first-person point of view when flying at distance. Finally, a 900 MHz XBee Pro Module (IEEE 802.15.4) provides the communication link between quadrotor units and between each unit and the base station. Each wireless link operates at a different band to minimize interference, and the ZigBee communication layer was chosen to operate at 900 MHz to increase the wireless range of critical mission data. Broadcasting at 50mW, this communication link has a maximum LOS range of approximately 9,000 meters which provides adequate range in more constricted environments.
The 900 MHz ZigBee link is set up in API mode, allowing data packets to be sent to specific addresses and not as a broadcast. This link communicates using two distinct packet structures—one to relay attitude information to followers, providing necessary information needed to complete the tracking algorithms discussed, and one to relay (at a slower frequency) battery level, the relative locations of swarm units with respect to the leader, and any pertinent environmental data collected. The former transmits at roughly 50 Hz, while the latter transmits at 2 Hz.

5.7 Sensor Data

The PixArt IR camera was tested using multiple IR beacons moving in three-dimensional space. When running at full speed the camera was able to return IR tracking results over the I^2C bus at approximately 500Hz. Figure 16 illustrates a complex movement of the IR beacons in three-dimensional space (a sweeping motion away from the IR camera sensor combined with a lateral shift). This data and frequency is more than adequate for flight tracking, as this is almost twice as fast as the PID loop running on the ATmega1280 for stabilization. In addition, the FOV was experimentally determined to be approximately 45° horizontally and 32° vertically. This is close to the results found in MATLAB of a 50° FOV that resulted in limited error. However, since this is limited in comparison to the model additional changes would have to be made to improve this camera's effectiveness. This could include increasing the following distance as well as mounting the camera on a stabilizing mount (in this case a micro-servo supplied with controls based on the platform attitude) to offset changes in pitch.

5.8 Sensor Fusion - Tracking Sensor Data

In addition to the flight control system installed on the units, at the core of the leader-follower system is the vision tracking system. On the lead quadrotor the CMOS camera is only used by the user

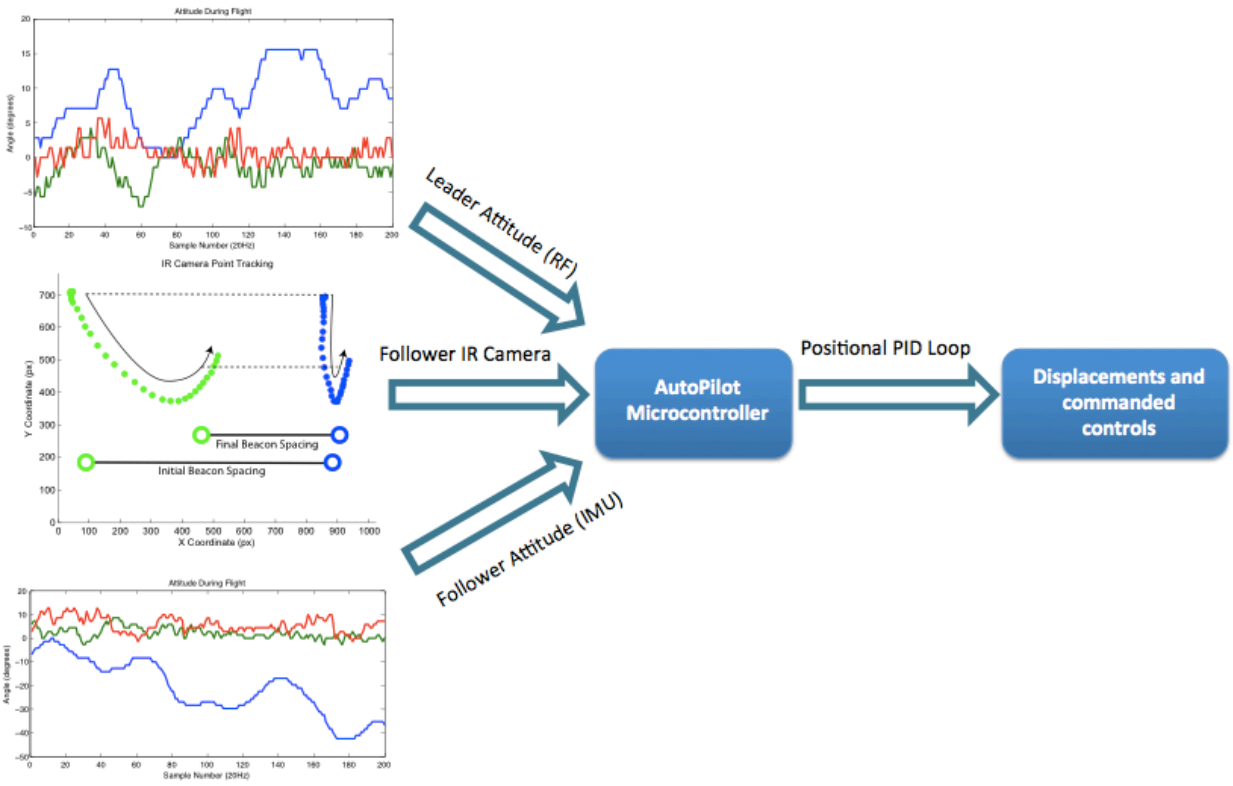


Figure 15: Sensor fusion of wireless data and IR-camera to determine 3D localization

for flight guidance (there is no input to the microcontrollers). However, the vision system on the follower quadrotor is essential to determining the spatial orientation and placement of the quadrotors while in flight. The sensor used on the follower units is a PixArt IR camera. This camera communicates with the sensor microcontroller on an I^2C (inter-integrated circuit) bus and is able to detect and track the locations of up to four IR points with a resolution of 1024x768 and a refresh rate of approximately 500Hz. To ensure that the lead quadrotor does not leave the FOV when the quadrotor pitches, the IR camera is attached to a micro-servo. Using the follower's attitude information the servo is actuated to offset the pitch and maintain the IR camera at a horizontal position. The sensor microcontroller is connected to both the IMU as well as the flight microcontroller using serial communication at a baud rate of 115,200. This provides the sensor microcontroller with the IMU (attitude) data of the follower node as well as the leader (which is transmitted over the XBee Pro wireless connection and sent through the flight microcontroller). This data is then combined with the IR camera data to provide the relative position data. This is done by first determining the camera projection based off of the yaw, pitch, and roll IMU information from each of the units and the differences in attitude to obtain the correct view of the IR beacon points which are then projecting onto the world frame (represented by the the x-axis straight ahead of the follower and parallel to the ground, the y-axis straight out to the right of the follower and parallel to the ground, and the z-axis straight up from the follower and perpendicular to the ground). After completing this projection, the x value provides pitch control, the y value provides roll control, the z value provides thrust control, and the difference in the yaw values between leader and follower provides yaw control to follow the lead unit.

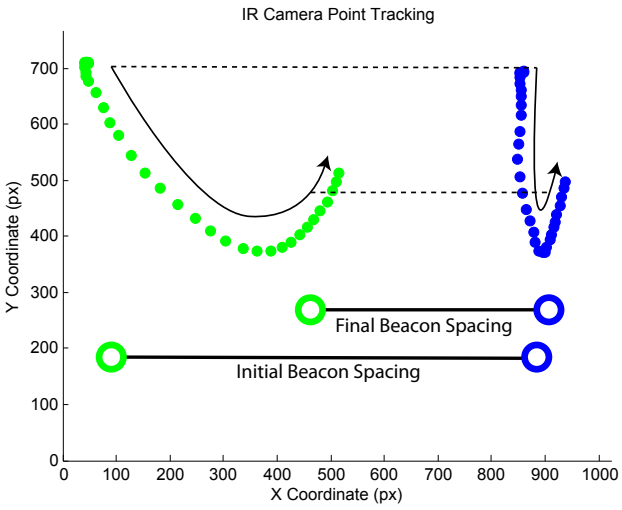


Figure 16: Infrared beacons perceived by the near-infrared camera

5.9 Environmental Sensor Data

The environmental sensor data is also combined to form more complete and effective information for the mission task. First, the ultrasonic rangefinder data is collected and aggregated to form a rough image of the area surrounding the units. When plotted in the GUI (Figure 13), this provides system users approximations to the dimensions of rooms and areas of interest. In addition, velocity data and/or GPS data (when possible) is combined with this range data to form a clearer SLAM estimate of the mission topology. This allows users and rescuers faster search times and improved situation response.

Other sensor payloads installed on the units provide a wide range of useful data to system users. For example when combined with the sonar sensor information and the SLAM data, results from a thermopile array return approximate locations of heat signatures in an area (possibly indicating people or other heat sources). When this information is combined with an atmospheric sensor (such as an airborne-particle sensor), these results can then be paired with sources of smoke that might indicate a fire. In this way additional sensor data is overlaid with existing data to form more relevant results to system users and improve mission success.

5.10 Localization Accuracy

The 3D localization created resulted in a high accuracy, low computation system. The localization information, determined in large part by the near-infrared camera on the follower, is limited to an area behind the leader and within the beam-angle of the infrared beacons and the field of view of the near-infrared camera. Figure 17 shows the measured accuracy of the system at discrete sampling points. Even at 2 meters away, the accuracy remains 5-10 centimeters despite the orientation of the leader or follower. Towards the edges of the field of view and the beam angle of the infrared beacons the accuracy diminishes as expected, and accuracy decreases as distance increases from the leader. Furthermore, the plot only shows distances up to 2 meters, but the system can measure location up to about 8 meters distance.

This allows a follower quadrotor to respond appropriately to displacement disturbances. For example, Figure 18 shows the follower's response to a forward displacement from the leader. Here the follower is commanded to follow at 120 cm (the black line), and it tries to converge on this distance. The line at the bottom (blue) represents the commanded angle (in this case pitch) corresponding to the displacement.

5.11 Website Impact

In order to better serve the community of UAV developers, all software as well as information about the hardware used has been made open-source to the general public. This information can be found at the url $http://www.airhacks.org$. This website has already had a large impact on projects from a variety of different countries and universities. Specifically, as of May 2011 this website has had 1,335 distinct visitors from 77 countries around the world. The team has also received several comments of thanks for helping other projects succeed in their goals. A map showing which areas of the world have/are visiting the website the most can be seen in Figure 19.

6 Lessons Learned

In the hope of providing guidance and support to future students during their Senior Design projects, this section contains lessons learned throughout the process of completing this system. These include purchase order delays, using a code repository, implementing checksums in data communication, and planning for unforeseen issues.

6.1 Order Delays

Due to the RAVEN system's extensive reliance on physical hardware, numerous purchase orders were placed to obtain equipment and parts to construct the quadrotor platforms. While this worked well for many of the parts, some of the crucial components (namely the motor and motor controllers) turned out to be backordered for almost a month during the early stages of the project. This prevented important development from occurring and required the schedule to be adjusted to take into account the delays. Although the project was able to proceed as scheduled when the parts arrived, valuable time could have been applied toward the project had an additional step been taken in the selection of project components. During the selection process, backup vendors and similar components could have been utilized. This would allow components to be ordered in a timely manner and prevent project delays.

6.2 SVN Code Repository

In order to complete the RAVEN system, several thousand lines of code on multiple platforms and multiple languages had to be systematically and concurrently developed. To ensure that code was maintained throughout the process, an SVN repository (a software revision control system) was utilized. This provided several advantages. First, the SVN repository provided a common backup location for the duration of the project. This prevented costly errors from occurring due to computer and media failures. Second, it provided a common method to keep the code up-to-date

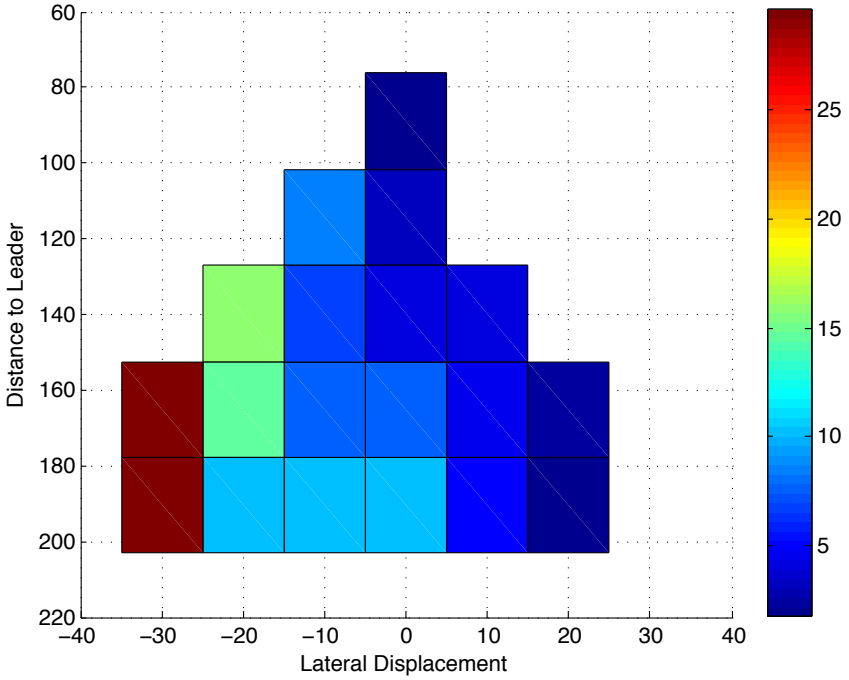


Figure 17: Accuracy of the 3D localization scheme. All measurements are in centimeters

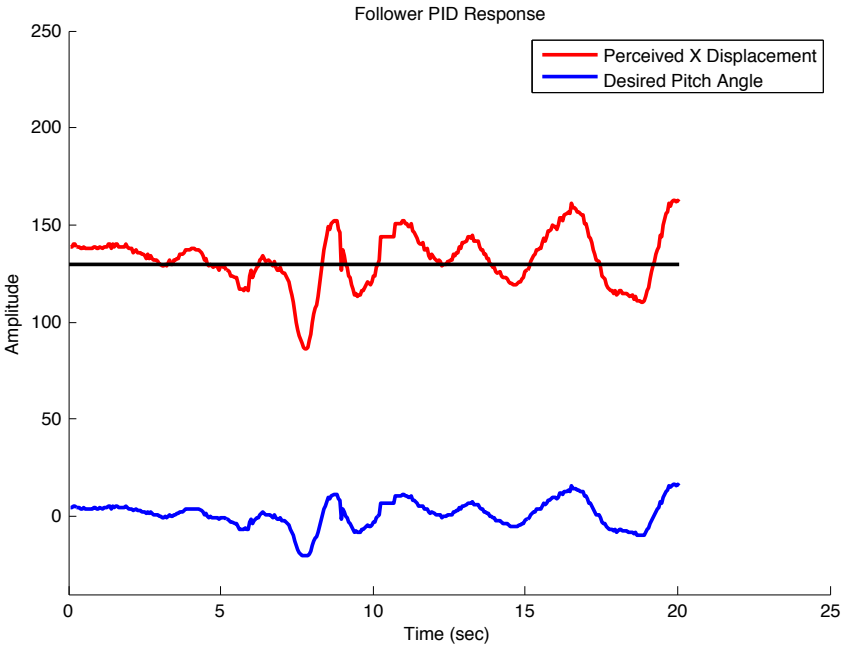


Figure 18: Response of follower to displacement (red converging on black) with corresponding desired angle to compensate for distance (blue).

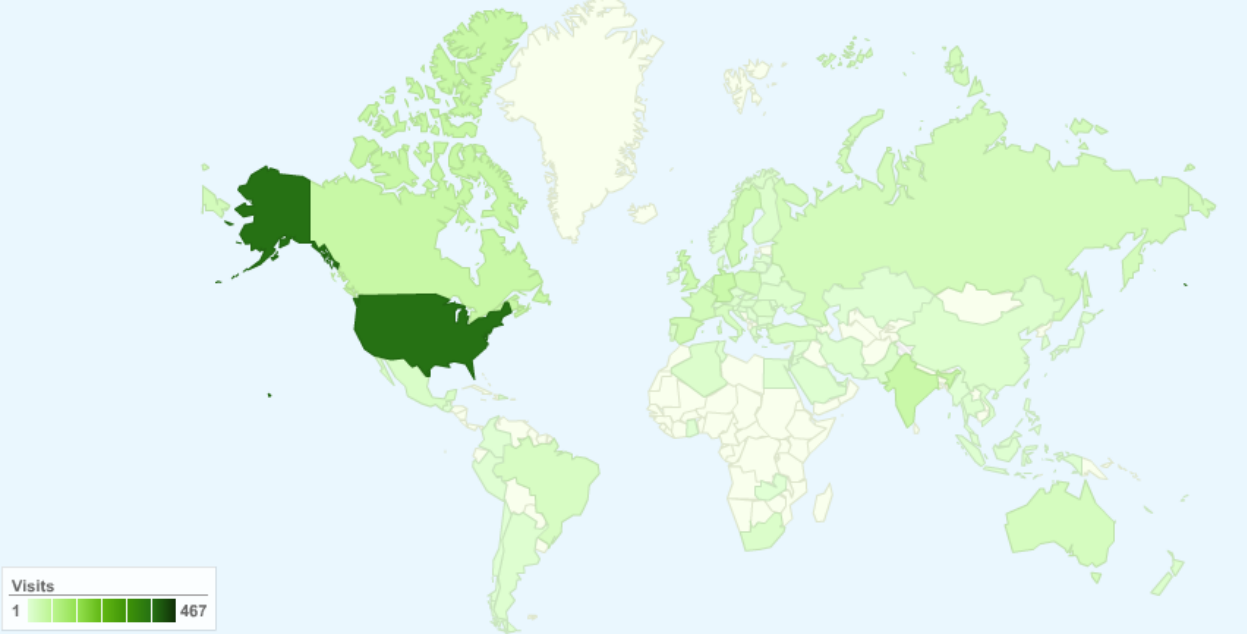


Figure 19: Frequency of visits to www.airhacks.org vs geographical location. Plot obtained from Google Analytics.

on each of the group member's computers. This allowed multiple parts of the code to be modified and maintained without running into inconsistencies. Last, the repository allows for revisions of the code to be checked out. This allowed for code versions to be reverted for debugging if modules stopped working correctly.

6.3 Communication Checksum Implementation

An important part of the R.A.V.E.N. system is platform communication and sensor data acquisition. This leads to many microcontrollers and devices communicating over wire and wireless connections. A problem that arose in the beginning of the project was data corruption during transmission. This was due to non-robust coding practices that did not incorporate a checksum, which adds an component to the data packet to check for transmission errors. After implementing the checksum in the IMU and wireless data communication, random errors were removed and the system responded appropriately.

6.4 Planning for Unforeseen Issues

Since the quadrotors used in the R.A.V.E.N. system were built in-house, unforeseen hardware issues arose that were outside our area of expertise. Specifically, the frame of the quadrotors combined with the movement of the motors/props created high frequency oscillations that resonated in the frame material and caused errors in the gyro measurements provided by the inertial measurement unit. This resulted in the flight microcontroller (which maintained stability) attempting to correct for false signals and lead to moments of system instability during flight. Although physical isolation of the IMU worked in correcting this severe issue, had there not been extra time built

into the schedule to account for unknown errors during the project this issue could have resulted in problems completing the project.

7 Equipment, Fabrication, and Software Needs

This project used equipment and software readily within Penn Engineering.

8 Conclusions and Recommendations

After completing this project, the following conclusions were reached:

1. We have designed a leader-follower system with relative localization. Quadrotor platforms with the system installed are able to determine their relative locations in three-dimensional space using only the inertial measurement data and infrared beacon locations. This allows for the correct command responses for tracking and following to be achieved and sent to the flight control system.

2. Localization is a difficult issue and key challenge to today's robotic systems. Our system takes a step toward solving this problem using low-power, inexpensive components implemented on a highly mobile device. Although the system is not able to completely localize (a human operator is still required to control the lead unit), the system provides a current implementation with cooperative sensor fusion in software.

3. In addition to the leader-follower system, a highly-capable quadrotor platform was designed. The RAVEN quadrotors are able to use readily-available lithium-polymer battery systems and carry payloads in excess of 800 grams. The available connections on the microcontrollers also allow for easy implementation of a wide range of plug-and-play environmental sensors, sending desired data back to the user base-station.

4. The entire system is open-source in design. This means that both the hardware and software information is made available on our website ($http://www.airhacks.org$). The platform information section of the website provides guidance for component selection as well as vendors, and the code repository allows visitors to examine our implementation of the leader-follower system and download to test and make changes.

Future work that could be done with this project includes expanding the capabilities of the platform as well as utilizing the platform in new applications. This could involve improving the platform itself by creating the next-generation quadrotor with upgraded components and improved microcontrollers or using the current platform in situations other than SAR or with other sensors/equipment installed.

9 Nomenclature

AHRS	Attitude Heading Reference System
Attitude	Orientation relative to a fixed frame
BLDC	Brushless Direct Current (DC)
EMF	Electromotor Force
EKF	Extended Kalman Filter
ESC	Electronic Speed Controller (Motor Controller)
FOV	Field of view
GPS	Global Positioning System
GUI	Graphical user interface
IR	Infrared
LIDAR	Light detection and ranging
IMU	Inertial measurement unit
MAV	Micro-Aerial Vehicle
PCB	Printed Circuit Board
Pitch	Rotation about the y axis, in the lateral direction
PWM	Pulse Width Modulation
RC	Remote Control
RF	Radio Frequency
Roll	Rotation about the x axis, in the direction of motion
RAV	Robotic Air Vehicle
RSSI	Received Signal Strength Indicator
SAR	Search and Rescue
SLAM	Simultaneous location and mapping
Quadrotor	Four-rotor vertical take-off and landing vehicle
UAV	Unmanned Aerial Vehicle
VTOL	Vertical Take-Off and Landing
Yaw	Rotation about the z axis, normal to the plane of locomotion

10 References

[1] Voyles, Richard and Choset, Howie. Editorial: Search and Rescue Robots. Journal of Field Robotics 25(1), 12 (2008). Wiley Periodicals, Inc.

[2] MIT Department of Mechanical Engineering. Autonomous Navigation, Position Control, and Landing of a Quadrotor Using GPS and Vision. Design of Eletromechanical Robotic Systems, Dec 2009.

[3] University of Pennsylvania General Robotics, Automation, Sensing & Perception (GRASP) Laboratory, Multi-Autonomous Ground-robotic International Challenge (MAGIC) Team, 2010.

[4] Markus Achetelik, Abraham Bachrach, Ruijie He, Samuel Prentice, and Nicholas Roy. Autonomous Navigation and Exploration of a Quadrotor Helicopter in GPS-denied Indoor Environments, Association for Unmanned Vehicle Systems International (AUVSI) Symposium 2009.

[5] Daniel Mellinger, Nathan Michael, and Vijay Kumar. Trajectory Generation and Control for Precise Aggressive Maneuvers with Quadrotors. Int. Symposium on Experimental Robotics, Dec 2010.

[6] Renzo De Nardi, Owen Holland, John Woods, and Adrian Clark. SwarMAV: A Swarm of Miniature Aerial Vehicles. 21^{st} Bristol UAV Systems Conference, April 2006.

[7] Jonathan Fink, Tom Collins, Vijay Kumar, Yasamin Mostofi, John Baras, and Brian Sadler. A Simulation Environment for Modeling and Development of Algorithms for Ensembles for Mobile Microsystems, 2009.

[8] Lorenz Meier, Friedrich Fraundorfer, and Marc Pollefreys. Onboard Object Recognition on the PixHawk Micro Air Vehicle. Fourth International Conference on Cognitive Systems, 2010.

11 Bibliography

DraganFly Innovations Inc. Draganfly government and military. http://www.draganfly.com/our-customers/government.php.

William Etter, Paul Martin, and Rahul Mangharam. Cooperative Flight Guidance of Autonomous Unmanned Aerial Vehicles. University of Pennsylvania. The Second International Workshop on Networks of Cooperating Objects 2011.

Dongbing Gu. Leader-Follower Flocking: Algorithms and Experiments. IEEE Transactions on Control Systems Technology, Vol. 17, No. 5. Sept 2009.

G. Hoffmann, D. Gorur Rajnarayan, S. L. Waslander, D. Dostalv, J. Soon Jang, and C. J. Tomlin. The Stanford Testbed of Autonomous Rotorcraft for Multi-Agent Control (STARMAC). Digital Avionics Systems Conference, 2, 2004.

Gabriel M. Hoffmann and Steven L. Waslander. Quadrotor Helicopter Trajectory Tracking Control, University of California - Berkeley, Aug 2008.

C. Howard. Automated Life Jacket Detection Enhances Search and Rescue Operations. MilitaryAerospace.com, 2011.

Aldo Jaimes, Srinath Kota, and Jose Gomez. An approach to surveillance an area using swarm of fixed wing and quad-rotor unmanned aerial vehicles UAV(s), Autonomous Control Engineering Lab, University of Texas at San Jose, 2008.

J. Leonard and H. Durrant-Whyte. Simultaneous map building and localization for an autonmous mobile robot. IEEE/RSJ International Workshop on Intelligent Robots and Systems, 3(5), Nov 1991.

Xiaohai Li, Zhijun Cai and Jizhong Xiao. Biologically Inspired FLocking of Swarms with Dynamic Topology in Uniform Environments.

James F. Roberts, Timothy S. Stirling, Jean-Christophe Zufferey, and Dario Floreano. Quadrotor Using Minimal Sensing For Autonomous Indoor Flight, Ecole Polytechnique Federale de Lausanne, Switzerland, 2007.

Ken Sugawara and Toshinori Watanabe. A Study on Biologically Inspired Flocking Robots. Proceedings of the 2003 IEEE International Conference on Robotics, Intelligent Systems and Signal Processing, pp. 324-330, Oct 2003.

Zongyao Wang, Dongbing Gu, and Huosheng Hu. Leader-follower Flocking Experiments Using Estimated Flocking Center. Proceedings of the 2009 IEEE Internation Conference on Mechatronics and Automation, Aug 9 2009.

Zongyao Wang and Dongbing Gu. A Local Sensor Based Leader-follower Flocking System. 2008 IEEE International Conference on Robotics and Automation, May 19 2008.

Neil R. Watson, Nigel W. John, William J. Crowther. Simulation of Unmanned Air Vehicle Flocking. Proceedings of the Theory and Practice of Computer Graphics, 2003.

12 Financial Information

This section contains financial information for the Remote Autonomous Vehicle Explorer Network project.

12.1 Budget Rational

To achieve the desired results for this project, a budget of approximately $1,300 was required. This budget allowed for the necessary components of the quadrotor platforms to be obtained as well as components specific to the RAVEN system. Since the quadrotors were custom-built and designed in-house, several different vendors (both in the U.S. and overseas) were utilized to obtain the off-the-shelf materials. The list of materials required for the R.A.V.E.N. system is found below in Table 5.
In addition to the main budget, other expenses were present. These can be divided into support and spare material expenses. Support material expenses were useful in completing the project however were not components of the main system designed. These included the lithium-polymer battery charger, the video display on the base station for user feedback, as well as the on-board high-definition camera for recording the system in use. Spare material expenses were needed to maintain the quadrotors and included extra motors, props, and electronic speed controllers.

12.2 Itemized Budget

An itemized budget listed in approximate order of priority follows.

Table 5: Itemized budget

Item	Vendor	Quantity	Unit Price	Subtotal
Inertial Measurement Unit	CHRobotics	2	$199	$398
Pixart IR Camera	Amazon	1	$30	$30
mbed Microcontroller	Sparkfun	1	$59.95	$59.95
ArduPilot mega	Sparkfun	2	$59.95	$119.90
Quadrotor Frame	CNC Helicopter	2	$124	$248
XBee - Wire Antenna	Sparkfun	2	$43	$86
Xbee - External Antenna	Sparkfun	1	$44.95	$44.95
Electronic Speed Controller	Hobbyking	8	$7.80	$62.40
Brushless Motor	Hobbyking	8	$14.95	$119.60
LiPo Battery	Hobby Partz	2	$16.95	$33.90
RC Control Unit	Hobby Partz	1	$44.95	$44.95
XBee Adapter	Sparkfun	1	$24.95	$24.95
XBee Adapter	LadyAda	2	$10	$20
Props	Draganflyer	2	$9.99	$19.98
XBee Antenna	Sparkfun	1	$7.95	$7.95
			Total	$1,320.53

This project is funded by ESE Senior Design, the University of Pennsylvania mLab (Embedded Systems Laboratory), and a Vagelos Undergraduate Research Grant.

13 Ethics

The R.A.V.E.N. multi-quadrotor system was developed in order to aid in search and rescue scenarios, however this good-intentioned application did not render the final product void of all ethical implications. Specifically, the system can be analyzed in terms of a larger, international context and the ethics of the individual components used.

13.1 Within a Larger Context

The R.A.V.E.N. system raises such ethical questions as right to privacy and the potential for creating harmful military applications.

13.1.1 Right to Privacy

The baseline R.A.V.E.N. system was designed with an on-board camera to relay to the user a first-person view from the lead quadrotor. By the very nature of the system designed, this camera will reach locations not naturally obtained by the human eye. In effect, this could breach the contract of privacy by placing a camera where an individual could not reasonably expect one to be located. This could become increasingly troublesome with the addition of more advanced vision sensors such as thermal cameras. These would allow the user of the R.A.V.E.N. system to see heat signatures beyond line-of-sight, breaching the unspoken contract of privacy even further. This, however, is a reasonable concern with any UAV system–not just multi-unit UAV systems.

13.1.2 Military Applications

In dealing with UAV systems there often exists the possibility of adapting equipment to meet military needs. The R.A.V.E.N. system is no different in this regard–in fact, individuals interested in adopting the project as a military contract have already contacted the team. Working within the context of a military contract presents a number of ethical dilemmas. It is not difficult to imagine situations in which a multiple UAV system would yield significant military advantages–whether increasing the radius of effectiveness during a bombing raid or allowing allied soldiers to spot enemy soldiers and equipment more easily. In effect, the R.A.V.E.N. system could be used (directly or indirectly) towards the killing of enemy soldiers.

13.2 Individual Components Used

The components used in constructing the R.A.V.E.N. prototype do not present any ethical dilemmas in and of themselves, but should the system be mass-manufactured there would necessarily be concern over the disposal of the lead solder used in the electrical connections. Similarly, the batteries used in the R.A.V.E.N. system are high power lithium polymer batteries, and they should be handled and disposed of carefully.

13.3 Recommendations

Towards addressing these possible ethical concerns, several design constraints and guidelines may be followed. The proposed guidelines are as follow:

1. Restrict the publishing and distributing of images and video surveillance obtained by use of the system. This would mitigate any breach of privacy caused by the system.

2. Use of the system should be limited to emergency response personnel. This would disallow the general public from using the system as a hobby toy and thereby limiting access of the video and image surveillance data to individuals that would truly benefit from the information.

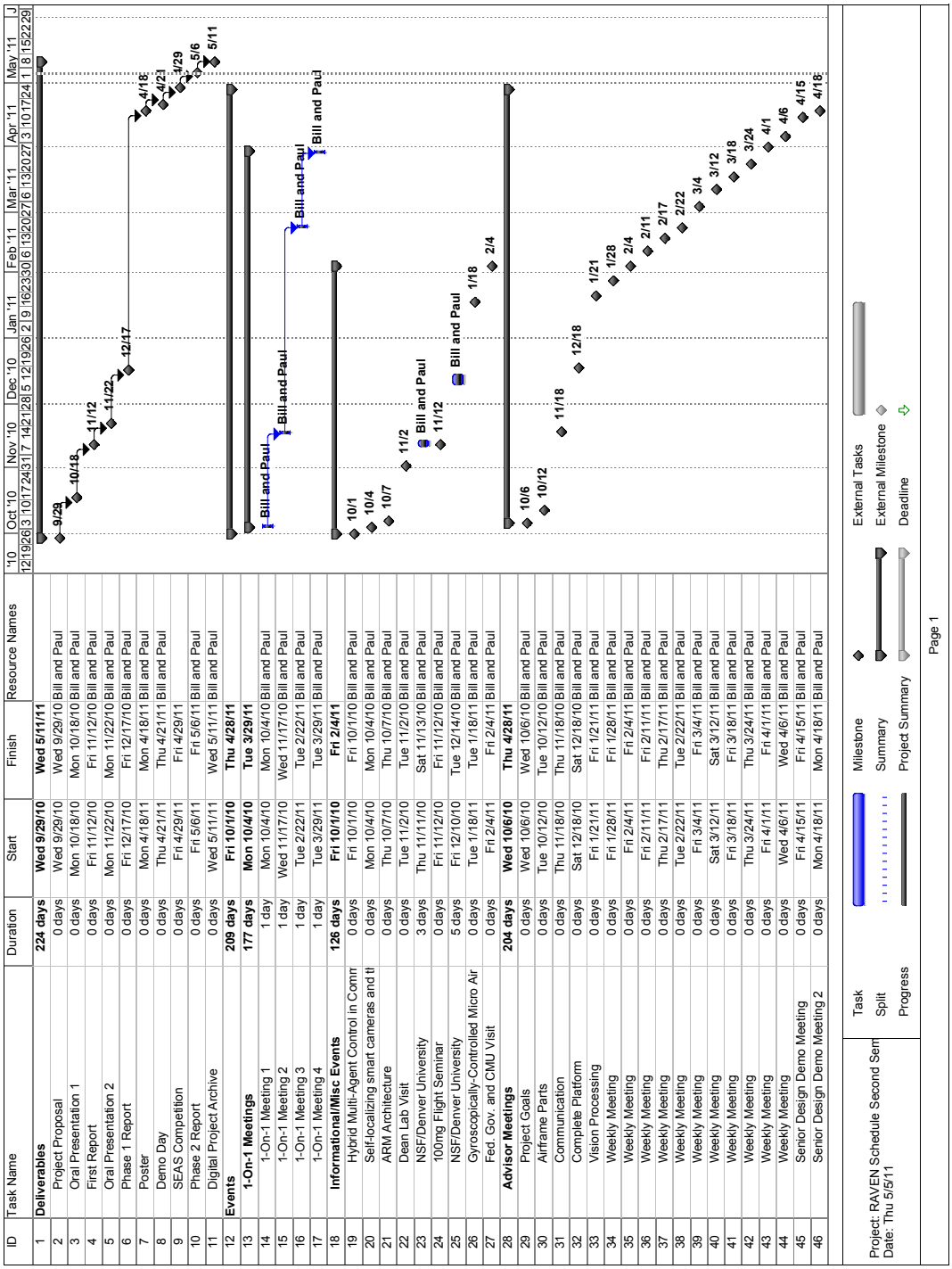
3. Care should be taken in building each unit of the system to minimize use of lead solder.

4. Training sessions should be instigated for all consumers of the system, outlining not only how to safely fly the lead quadrotor but also how to properly handle and dispose of the high-power lithium polymer batteries.

14 Software Documentation

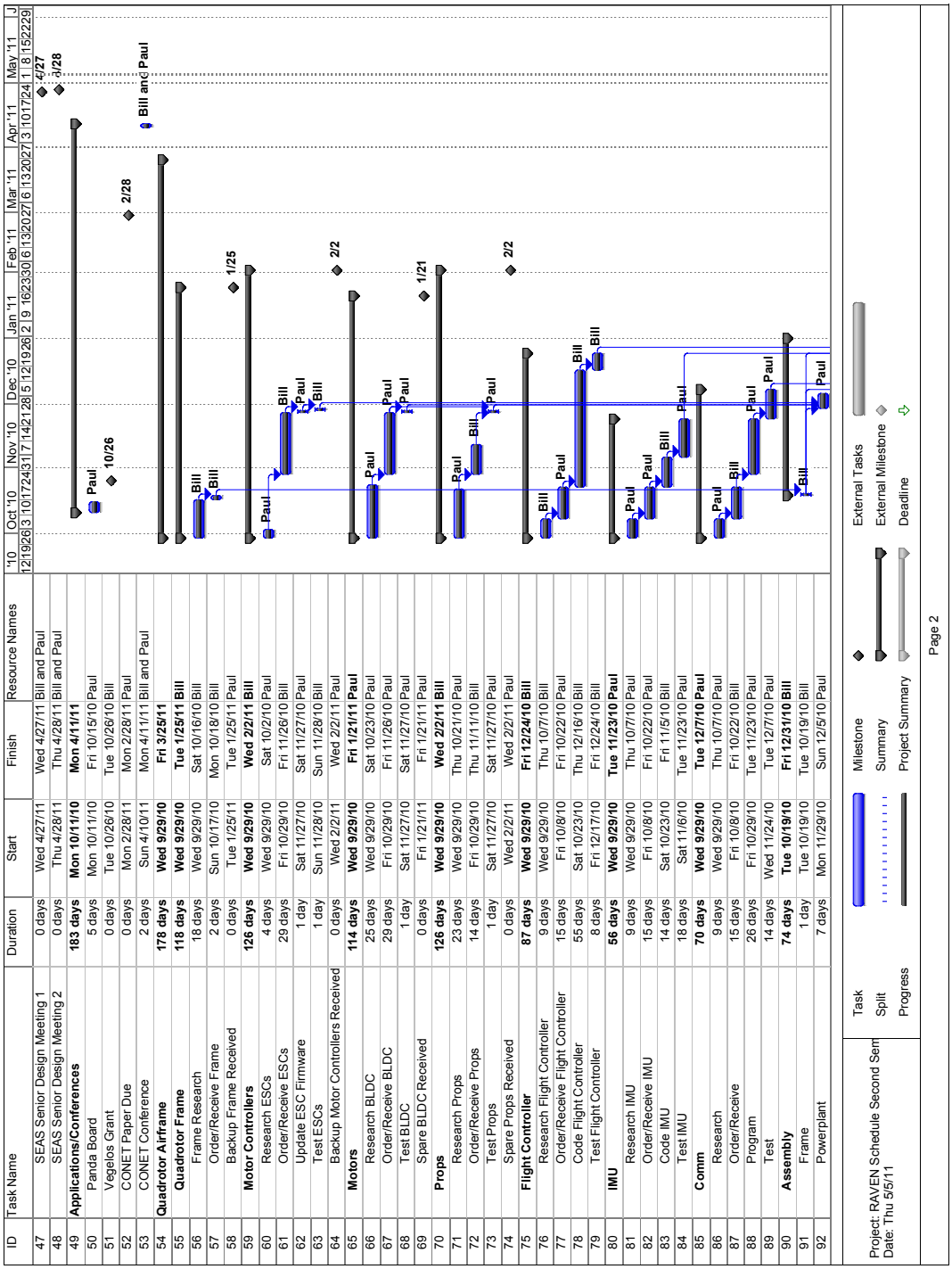
See Appendix 2 for a sample of Software Documentation.

A1 Expanded Project Schedule



ID	Task Name	Duration	Start	Finish	Resource Names
1	**Deliverables**	**224 days**	**Wed 9/29/10**	**Wed 5/11/11**	
2	Project Proposal	0 days	Wed 9/29/10	Wed 9/29/10	Bill and Paul
3	Oral Presentation 1	0 days	Mon 10/18/10	Mon 10/18/10	Bill and Paul
4	First Report	0 days	Fri 11/12/10	Fri 11/12/10	Bill and Paul
5	Oral Presentation 2	0 days	Mon 11/22/10	Mon 11/22/10	Bill and Paul
6	Phase 1 Report	0 days	Fri 12/17/10	Fri 12/17/10	Bill and Paul
7	Poster	0 days	Mon 4/18/11	Mon 4/18/11	Bill and Paul
8	Demo Day	0 days	Thu 4/21/11	Thu 4/21/11	Bill and Paul
9	SEAS Competition	0 days	Fri 4/29/11	Fri 4/29/11	
10	Phase 2 Report	0 days	Fri 5/6/11	Fri 5/6/11	Bill and Paul
11	Digital Project Archive	0 days	Wed 5/11/11	Wed 5/11/11	Bill and Paul
12	**Events**	**209 days**	**Fri 10/1/10**	**Thu 4/28/11**	
13	**1-On-1 Meetings**	**177 days**	**Mon 10/4/10**	**Tue 3/29/11**	
14	1-On-1 Meeting 1	1 day	Mon 10/4/10	Mon 10/4/10	Bill and Paul
15	1-On-1 Meeting 2	1 day	Wed 11/17/10	Wed 11/17/10	Bill and Paul
16	1-On-1 Meeting 3	1 day	Tue 2/22/11	Tue 2/22/11	Bill and Paul
17	1-On-1 Meeting 4	1 day	Tue 3/29/11	Tue 3/29/11	Bill and Paul
18	**Informational/Misc Events**	**126 days**	**Fri 10/1/10**	**Fri 2/4/11**	
19	Hybrid Multi-Agent Control in Comm	0 days	Fri 10/1/10	Fri 10/1/10	Bill and Paul
20	Self-localizing smart cameras and ti	0 days	Mon 10/4/10	Mon 10/4/10	Bill and Paul
21	ARM Architecture	0 days	Thu 10/7/10	Thu 10/7/10	Bill and Paul
22	Dean Lab Visit	0 days	Tue 11/2/10	Tue 11/2/10	Bill and Paul
23	NSF/Denver University	3 days	Thu 11/11/10	Sat 11/13/10	Bill and Paul
24	100mg Flight Seminar	0 days	Fri 11/12/10	Fri 11/12/10	Bill and Paul
25	NSF/Denver University	5 days	Fri 12/10/10	Tue 12/14/10	Bill and Paul
26	Gyroscopically-Controlled Micro Air	0 days	Tue 1/18/11	Tue 1/18/11	Bill and Paul
27	Fed. Gov. and CMU Visit	0 days	Fri 2/4/11	Fri 2/4/11	Bill and Paul
28	**Advisor Meetings**	**204 days**	**Wed 10/6/10**	**Thu 4/28/11**	
29	Project Goals	0 days	Wed 10/6/10	Wed 10/6/10	Bill and Paul
30	Airframe Parts	0 days	Tue 10/12/10	Tue 10/12/10	Bill and Paul
31	Communication	0 days	Thu 11/18/10	Thu 11/18/10	Bill and Paul
32	Complete Platform	0 days	Sat 12/18/10	Sat 12/18/10	Bill and Paul
33	Vision Processing	0 days	Fri 1/21/11	Fri 1/21/11	Bill and Paul
34	Weekly Meeting	0 days	Fri 1/28/11	Fri 1/28/11	Bill and Paul
35	Weekly Meeting	0 days	Fri 2/4/11	Fri 2/4/11	Bill and Paul
36	Weekly Meeting	0 days	Fri 2/11/11	Fri 2/11/11	Bill and Paul
37	Weekly Meeting	0 days	Thu 2/17/11	Thu 2/17/11	Bill and Paul
38	Weekly Meeting	0 days	Tue 2/22/11	Tue 2/22/11	Bill and Paul
39	Weekly Meeting	0 days	Fri 3/4/11	Fri 3/4/11	Bill and Paul
40	Weekly Meeting	0 days	Sat 3/12/11	Sat 3/12/11	Bill and Paul
41	Weekly Meeting	0 days	Fri 3/18/11	Fri 3/18/11	Bill and Paul
42	Weekly Meeting	0 days	Thu 3/24/11	Thu 3/24/11	Bill and Paul
43	Weekly Meeting	0 days	Fri 4/1/11	Fri 4/1/11	Bill and Paul
44	Weekly Meeting	0 days	Wed 4/6/11	Wed 4/6/11	Bill and Paul
45	Senior Design Demo Meeting	0 days	Fri 4/15/11	Fri 4/15/11	Bill and Paul
46	Senior Design Demo Meeting 2	0 days	Mon 4/18/11	Mon 4/18/11	Bill and Paul

Figure 20: Gantt chart showing expanded project schedule - page 1.



ID	Task Name	Duration	Start	Finish	Resource Names
47	SEAS Senior Design Meeting 1	0 days	Wed 4/27/11	Wed 4/27/11	Bill and Paul
48	SEAS Senior Design Meeting 2	0 days	Thu 4/28/11	Thu 4/28/11	Bill and Paul
49	**Applications/Conferences**	**183 days**	**Mon 10/11/10**	**Mon 4/11/11**	
50	Panda Board	5 days	Mon 10/11/10	Fri 10/15/10	Paul
51	Vegelos Grant	0 days	Tue 10/26/10	Tue 10/26/10	Bill
52	CONET Paper Due	0 days	Mon 2/28/11	Mon 2/28/11	Paul
53	CONET Conference	2 days	Sun 4/10/11	Mon 4/11/11	Bill and Paul
54	**Quadrotor Airframe**	**178 days**	**Wed 9/29/10**	**Fri 3/25/11**	
55	**Quadrotor Frame**	**118 days**	**Wed 9/29/10**	**Tue 1/25/11**	**Bill**
56	Frame Research	18 days	Wed 9/29/10	Sat 10/16/10	Bill
57	Order/Receive Frame	2 days	Sun 10/17/10	Mon 10/18/10	Bill
58	Backup Frame Received	0 days	Tue 1/25/11	Tue 1/25/11	Paul
59	**Motor Controllers**	**126 days**	**Wed 9/29/10**	**Wed 2/2/11**	**Bill**
60	Research ESCs	4 days	Wed 9/29/10	Sat 10/2/10	Paul
61	Order/Receive ESCs	29 days	Fri 10/29/10	Fri 11/26/10	Bill
62	Update ESC Firmware	1 day	Sat 11/27/10	Sat 11/27/10	Paul
63	Test ESCs	1 day	Sun 11/28/10	Sun 11/28/10	Bill
64	Backup Motor Controllers Received	0 days	Wed 2/2/11	Wed 2/2/11	Paul
65	**Motors**	**114 days**	**Wed 9/29/10**	**Fri 1/21/11**	**Paul**
66	Research BLDC	25 days	Wed 9/29/10	Sat 10/23/10	Paul
67	Order/Receive BLDC	29 days	Fri 10/29/10	Fri 11/26/10	Paul
68	Test BLDC	1 day	Sat 11/27/10	Sat 11/27/10	Paul
69	Spare BLDC Received	0 days	Fri 1/21/11	Fri 1/21/11	Paul
70	**Props**	**126 days**	**Wed 9/29/10**	**Wed 2/2/11**	**Bill**
71	Research Props	23 days	Wed 9/29/10	Thu 10/21/10	Paul
72	Order/Receive Props	14 days	Fri 10/29/10	Thu 11/11/10	Bill
73	Test Props	1 day	Sat 11/27/10	Sat 11/27/10	Paul
74	Spare Props Received	0 days	Wed 2/2/11	Wed 2/2/11	Paul
75	**Flight Controller**	**87 days**	**Wed 9/29/10**	**Fri 12/24/10**	**Bill**
76	Research Flight Controller	9 days	Wed 9/29/10	Thu 10/7/10	Bill
77	Order/Receive Flight Controller	15 days	Fri 10/8/10	Fri 10/22/10	Paul
78	Code Flight Controller	55 days	Sat 10/23/10	Thu 12/16/10	Bill
79	Test Flight Controller	8 days	Fri 12/17/10	Fri 12/24/10	Bill
80	**IMU**	**56 days**	**Wed 9/29/10**	**Tue 11/23/10**	**Paul**
81	Research IMU	9 days	Wed 9/29/10	Thu 10/7/10	Paul
82	Order/Receive IMU	15 days	Fri 10/8/10	Fri 10/22/10	Paul
83	Code IMU	14 days	Sat 10/23/10	Fri 11/5/10	Bill
84	Test IMU	18 days	Sat 11/6/10	Tue 11/23/10	Paul
85	**Comm**	**70 days**	**Wed 9/29/10**	**Tue 12/7/10**	**Paul**
86	Research	9 days	Wed 9/29/10	Thu 10/7/10	Paul
87	Order/Receive	15 days	Fri 10/8/10	Fri 10/22/10	Bill
88	Program	26 days	Fri 10/29/10	Tue 11/23/10	Paul
89	Test	14 days	Wed 11/24/10	Tue 12/7/10	Paul
90	**Assembly**	**74 days**	**Tue 10/19/10**	**Fri 12/31/10**	**Bill**
91	Frame	1 day	Tue 10/19/10	Tue 10/19/10	Bill
92	Powerplant	7 days	Mon 11/29/10	Sun 12/5/10	Paul

Figure 21: Gantt chart showing expanded project schedule - page 2.

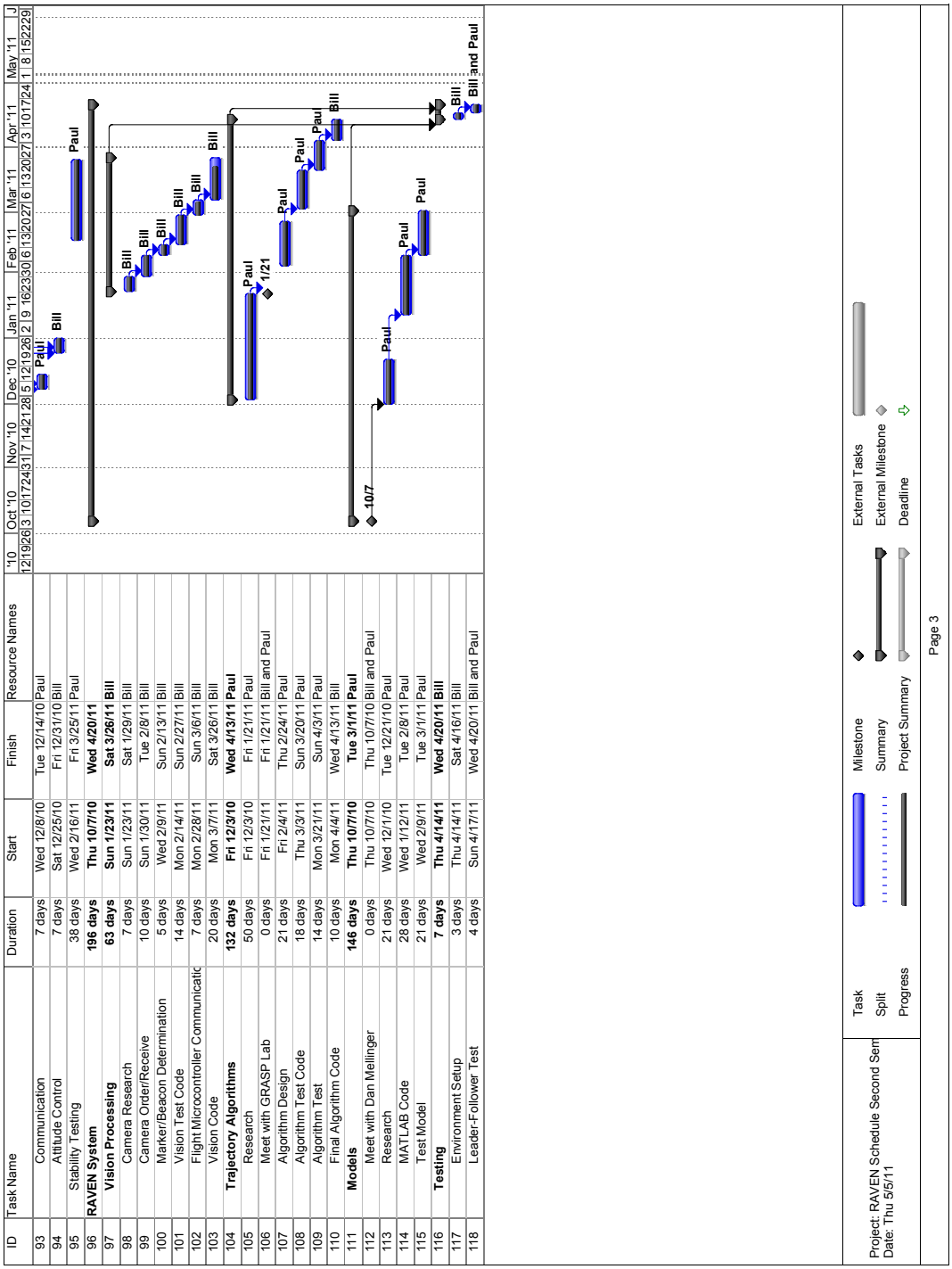


Figure 22: Gantt chart showing expanded project schedule - page 3.

A2 Sample Software Documentation

The code presented below is for the ArduPilot Mega microcontroller. This is the main microcontroller used for stabilization of the quadrotor platform, and the code below is required to request a packet containing IMU data from the IMU and to parse the incoming data.

ArduPilot Mega IMU Code Version 3

```
/*****************************************************************
 R.A.V.E.N. Quadrotor - November 2010
 www.AirHacks.com
 Copyright (c) 2010.  All rights reserved.

 Version: 1.6
 March 27, 2011

 Authors:
       William Etter (UPenn EE 'll)
       Paul Martin (UPenn EE 'll)

 This program is free software: you can redistribute it and/or modify
 it under the terms of the GNU General Public License as published by
 the Free Software Foundation, either version 3 of the License, or
 (at your option) any later version.

 This program is distributed in the hope that it will be useful,
 but WITHOUT ANY WARRANTY; without even the implied warranty of
 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 GNU General Public License for more details.

 You should have received a copy of the GNU General Public License
 along with this program. If not, see <http://www.gnu.org/licenses/>.
*****************************************************************/

// TO DO
//  - Get correct hex value for ZERO_RATE_GYROS and put into Calibrate function
//  - Do something if IMU didn't initialize successfully
//  - Check if more/less delay is required during startup


/*****************************************************************
//        IMU SERIAL SETUP
*****************************************************************/
// Baud Rate = 115,200
#define IMUSerialBaud  115200
#define IMUSerialAvailable  Serial2.available
#define IMUSerialWrite  Serial2.write
#define IMUSerialRead  Serial2.read
#define IMUSerialFlush  Serial2.flush
#define IMUSerialBegin  Serial2.begin

/*****************************************************************
//        PACKET STRUCTURE
*****************************************************************/
// Function 's' 'n' 'p' PT N d1 ... dN     CHK
// Byte      1   2   3  4  5 6  ... N+5  N+6 N+7
// RX Packet Description
// 1 - 3 Each received packet must begin with the three-byte (character) sequence
// "snp" to signal the beginning of a new packet.
// 4 PT specifies the packet type.
// 5 N specifies the number of data bytes to expect.
// 6 - (N+5) d1 through dN contain the N data bytes in the packet.
// (N+6) - (N+7) CHK is a two-byte checksum.

/*****************************************************************
//        CHR-6dm AHRS Value Conversion Factors
*****************************************************************/
#define YAW_FACTOR (0.0109863F*135.0F/100.0F)         //  /LSB
#define PITCH_FACTOR (0.0109863F*90.0F/72.0F)       //  /LSB
#define ROLL_FACTOR (0.0109863F*90.0F/72.0F)        //  /LSB
#define YAW_RATE_FACTOR 0.0137329F    //  /s/LSB
#define PITCH_RATE_FACTOR 0.0137329F  //  /s/LSB
#define ROLL_RATE_FACTOR 0.0137329F   //  /s/LSB
#define MAGX_FACTOR 0.061035F         //  mGauss/LSB
#define MAGY_FACTOR 0.061035F         //  mGauss/LSB
#define MAGZ_FACTOR 0.061035F         //  mGauss/LSB
#define GYROX_FACTOR 0.01812F         //  /s/LSB
#define GYROY_FACTOR 0.01812F         //  /s/LSB
#define GYROZ_FACTOR 0.01812F         //  /s/LSB
#define ACCELX_FACTOR 0.106812F       //  mg/LSB
#define ACCELY_FACTOR 0.106812F       //  mg/LSB
#define ACCELZ_FACTOR 0.106812F       //  mg/LSB

/*****************************************************************
//        IMU RX PACKET DEFINITIONS (-> IMU)
```

```
*****************************************************************************/
#define SET_ACTIVE_CHANNELS 0x80    // Specifies which channel data should be transmitted over the UART.
#define SET_SILENT_MODE 0x81        // Enables "Silent Mode." In Silent Mode, the AHRS only reports data when
                                    // a GET_DATA packet is received.
#define SET_BROADCAST_MODE 0x82     // Enables "Broadcast Mode." In Broadcast Mode, the AHRS automatically
                                    // transmits sensor data every Ts milliseconds, where Ts is encoded in
                                    // the data section of the SET_BROADCAST_MODE packet.
#define SET_GYRO_BIAS 0x83          // Manually sets the x-axis rate gyro bias term. The bias term can be
                                    // automatically set for all gyro axes by sending a ZERO_RATE_GYROS packet.
#define SET_ACCEL_BIAS 0x84         // Manually sets the x-axis accelerometer bias term.
#define SET_ACCEL_REF_VECTOR 0x85   // Manually sets the accelerometer reference vector used by the EKF to determine
                                    // "which way is down"
#define AUTO_SET_ACCEL_REF 0x86     // Causes the AHRS to set the current accelerometer measurements as the reference
                                    // vector (sets pitch and roll angles to zero for the given orientation).
#define ZERO_RATE_GYROS 0x87        // Starts internal self-calibration of all three rate gyro axes. By default, rate
                                    // gyros are zeroed on AHRS startup, but gyro startup calibration can be disabled
                                    // (or re-enabled) by sending a SET_START_CAL packet.
#define SELF_TEST 0x88              // Instructs the AHRS to perform a self-test of all sensor channels. A STATUS_REPORT
                                    // packet is transmitted after the self-test is complete.
#define SET_START_CAL 0x89          // Enables or disables automatic startup calibration of rate gyro biases.
#define SET_PROCESS_COVARIANCE 0x8A // Sets the 3x3 matrix representing the covariance of the process noise used in the
                                    // prediction step of the EKF.
#define SET_MAG_COVARIANCE 0X8B     // Sets the 3x3 matrix representing the covariance of the measurement noise used in
                                    // the magnetometer update step of the EKF
#define SET_ACCEL_COVARIANCE 0x8C   // Sets the 3x3 matrix representing the covariance of the measurement noise used in
                                    // the accelerometer update step of the EKF
#define SET_EKF_CONFIG 0x8D         // Sets the EKF_CONFIG register. Can be used to enable/disable the EKF, or to enable/disable
                                    // accelerometer and magnetometer updates.
#define SET_GYRO_ALIGNMENT 0x8E     // Sets the 3x3 matrix used to correct rate gyro crossaxis misalignment.
#define SET_ACCEL_ALIGNMENT 0x8F    // Sets the 3x3 matrix used to correct accelerometer cross-axis misalignment
#define SET_MAG_REF_VECTOR 0x90     // Sets the reference vector representing the expected output of the magnetometer when yaw,
                                    // pitch, and roll angles are zero.
#define AUTO_SET_MAG_REF 0x91       // Sets the current magnetometer output as the reference vector.
#define SET_MAG_CAL 0x92            // Sets the 3x3 magnetic field distortion correction matrix to compensate for soft-iron
                                    // distortions, axis misalignment, and sensor scale inconsistencies.
#define SET_MAG_BIAS 0x93           // Sets the magnetic field measurement bias to compensate for hard iron distortions
#define SET_GYRO_SCALE 0x94         // Sets the scale factors used to convert raw ADC data to rates for angle estimation on the EKF.
#define EKF_RESET 0x95              // Sets all terms in the current state covariance matrix to zero and re-initializes angle estimates.
#define RESET_TO_FACTORY 0x96       // Resets all AHRS setting to factory default values
#define WRITE_TO_FLASH 0xA0         // Writes current AHRS configuration to on-board flash, so that the configuration persists
                                    // when the power is cycled.
#define GET_DATA 0x01               // In Listen Mode, causes the AHRS to transmit data from all active channels in a SENSOR_DATA packet.
#define GET_ACTIVE_CHANNELS 0x02    // Reports which channels are "active" in an ACTIVE_CHANNELS_REPORT packet. Active channels are
                                    // sensor channels that are measured and transmitted in response to a GET_DATA packet, or periodically
                                    // in Broadcast Mode.
#define GET_BROADCAST_MODE 0x03     // Returns the BROADCAST_MODE_REPORT packet, which specifies whether the AHRS is in Broadcast Mode or Silent Mode.
#define GET_ACCEL_BIAS 0x04         // Return the bias values for all three accel axes in a ACCEL_BIAS_REPORT packet.
#define GET_ACCEL_REF_VECTOR 0x05   // Returns the accelerometer reference vector in an ACCEL_REF_VECTOR_REPORT packet
#define GET_GYRO_BIAS 0x06          // Returns the bias values for all three rate gyros in a GYRO_BIAS_REPORT packet.
#define GET_GYRO_SCALE 0x07         // Returns the rate gyro scale factors used internally to convert raw rate gyro data to actual rates in a
                                    // GYRO_SCALE_REPORT packet.
#define GET_START_CAL 0x08          // Reports whether the AHRS is configured to calibrate rate gyro biases automatically on startup in a
                                    // START_CAL_REPORT packet.
#define GET_EKF_CONFIG 0x09         // Returns the one byte EKF configuration register in an EKF_CONFIG_REPORT packet.
#define GET_ACCEL_COVARIANCE 0X0A   // Returns the variance of the accelerometer measurements used by the EKF update step
#define GET_MAG_COVARIANCE 0x0B     // Returns the variance of the magnetometer measurements used by the EKF update step
#define GET_PROCESS_COVARIANCE 0x0C // Returns the variance of the EKF prediction step
#define GET_STATE_COVARIANCE 0x0D   // Returns the 3x3 matrix representing the covariance of the current EKF state estimates in a
                                    // STATE_COVARIANCE_REPORT packet.
#define GET_GYRO_ALIGNMENT 0x0E     // Returns the 3x3 matrix used to correct rate gyro cross-axis misalignment
#define GET_ACCEL_ALIGNMENT 0x0F    // Returns the 3x3 matrix used to correct accelerometer cross-axis misalignment
#define GET_MAG_REF_VECTOR 0x10     // Returns the magnetometer reference vector in a MAG_REF_VECTOR_REPORT packet
#define GET_MAG_CAL Returns 0x11    // the 3x3 magnetometer correction matrix in an MAG_CAL_REPORT packet
#define GET_MAG_BIAS 0x12           // Returns the magnetometer bias correction used by the EKF. The bias is reported in a
                                    // MAG_BIAS_REPORT packet.

/*****************************************************************************
//        IMU TX PACKET DEFINITIONS (IMU ->)
*****************************************************************************/
#define COMMAND_COMPLETE 0xB0       // Transmitted by the AHRS upon successful completion of a command that does not require
                                    // data to be returned.
#define COMMAND_COMPLETE_SIZE 8     // 8 Bytes in COMMAND_COMPLETE Packet
#define COMMAND_FAILED 0xB1         // Transmitted by the AHRS when a command received over the UART could not be executed.
#define BAD_CHECKSUM 0xB2           // Transmitted by the AHRS when a received packet checksum does not match the sum of the
                                    // other bytes in the packet.
#define BAD_DATA_LENGTH 0xB3        // Transmitted by the AHRS when a received packet contained more or less data than
                                    // expected for a given packet type.
#define UNRECOGNIZED_PACKET 0xB4    // Transmitted if the AHRS receives an unrecognized packet.
#define BUFFER_OVERFLOW 0xB5        // Transmitted by the AHRS when the internal receive buffer overflows before a full packet is received.
#define STATUS_REPORT 0xB6          // Transmitted at the end of a self-test procedure, triggered by a SELF_TEST command.
#define STATUS_REPORT_SIZE 8        // 8 Bytes in STATUS_REPORT Packet
#define SENSOR_DATA 0xB7            // Sent in response to a GET_DATA packet, or sent automatically in Broadcast Mode.
#define GYRO_BIAS_REPORT 0xB8       // Sent in response to the GET_GYRO_BIAS command.
#define GYRO_SCALE_REPORT 0xB9      // Sent in response to a GET_GYRO_SCALE command.
#define START_CAL_REPORT 0xBA       // Sent in response to a GET_START_CAL command.
#define ACCEL_BIAS_REPORT 0xBB      // Sent in response to the GET_ACCEL_BIAS command.
#define ACCEL_REF_VECTOR_REPORT 0xBC // Sent in response to a GET_ACCEL_REF_VECTOR command, or sent after an AUTO_SET_ACCEL_REF command.
#define ACCEL_REF_VECTOR_REPORT_SIZE 13 // 13 Bytes in ACCEL_REF_VECTOR_REPORT Packet
#define ACTIVE_CHANNEL_REPORT 0xBD   // Sent in response to a GET_ACTIVE_CHANNELS command.
#define ACCEL_COVARIANCE_REPORT 0xBE // Sent in response to a GET_ACCEL_COVARIANCE command
#define MAG_COVARIANCE_REPORT 0xBF   // Sent in response to a GET_MAG_COVARIANCE command.
#define PROCESS_COVARIANCE_REPORT 0xC0 //Sent in response to a GET_PROCESS_COVARIANCE command
```

```
#define STATE_COVARIANCE_REPORT 0xC1 // Sent in response to a GET_STATE_COVARIANCE command
#define EKF_CONFIG_REPORT 0xC2       // Sent in response to a GET_EKF_CONFIG command.
#define GYRO_ALIGNMENT_REPORT 0xC3   // Sent in response to a GET_GYRO_ALIGNMENT command.
#define ACCEL_ALIGNMENT_REPORT 0xC4  // Sent in response to a GET_ACCEL_ALIGNMENT command.
#define MAG_REF_VECTOR_REPORT 0xC5   // Sent in response to a GET_MAG_REF_VECTOR command
#define MAG_CAL_REPORT 0xC6          // Sent in response to a GET_MAG_CAL command
#define MAG_BIAS_REPORT 0xC7         // Sent in response to a GET_MAG_BIAS command
#define BROADCAST_MODE_REPORT 0XC8   // Sent in response to a GET_BROADCAST_MODE command.

/*************************************************************************
Function: IMU_init()
Purpose:  Run self-test
          Set values to output
Returns:  boolean IMUinitialization = if IMU initialized successfully
*************************************************************************/
boolean IMU_init(void){
  // Variables
  boolean IMUinitialization = true;
  int sum = 0;

  // Open Serial Channel, 115,200 Baud Rate
  IMUSerialBegin(IMUSerialBaud);

  // ---------- Run Self-Test -----------------
  // clear incoming buffer
  IMUSerialFlush();
  sum = 0;
  // Packet header
  IMUSerialWrite('s');
  sum+='s';
  IMUSerialWrite('n');
  sum+='n';
  IMUSerialWrite('p');
  sum+='p';
  // Packet type
  IMUSerialWrite(SELF_TEST);
  sum+=SELF_TEST;
  // Number of data bytes
  IMUSerialWrite((byte)0);
  sum+=(byte)0;
  // Checksum = Sum of all preceding bytes in packet
  IMUSerialWrite((byte)((sum >> 8) & 0x0FF));
  IMUSerialWrite((byte)((sum) & 0x0FF));

  delay(200);

  // ---------- Set IMU Values Channels to Output-----------------
  // clear incoming buffer
  IMUSerialFlush();
  sum = 0;
  // Packet header
  IMUSerialWrite('s');
  sum+='s';
  IMUSerialWrite('n');
  sum+='n';
  IMUSerialWrite('p');
  sum+='p';
  // Packet type
  IMUSerialWrite(SET_ACTIVE_CHANNELS);
  sum+=SET_ACTIVE_CHANNELS;
  // Number of data bytes
  IMUSerialWrite((byte)2);
  sum+=(byte)2;
  IMUSerialWrite(0xFC); // Activate Estimates and Rates for Yaw, Pitch, and Roll
  //IMUSerialWrite(0xE0); // Activate Estimates for Yaw, Pitch, and Roll
  sum+=0xFC;
  IMUSerialWrite((byte)0); // Don't Activate Rate Gyro data in X, Y, and Z
  sum+=0x00;
  // Checksum = Sum of all preceding bytes in packet
  IMUSerialWrite((byte)((sum >> 8) & 0x0FF));
  IMUSerialWrite((byte)((sum) & 0x0FF));

  //boolean silent = IMU_silent();

  IMUSerialFlush();
  return IMUinitialization;
}

/*************************************************************************
Function: IMU_calibrate()
Purpose:  Calibrates IMU
            - Zero Rate Gyros
            - Zero Accelerometer Measurements
Returns:  boolean noerror = if IMU calibrated successfully
*************************************************************************/
boolean IMU_calibrate(void){
  boolean noerror = true;
  // ---------- Zero Rate Gyros -----------------
  // clear incoming buffer
  IMUSerialFlush();
  int sum = 0;
  // Packet header
```

```
  IMUSerialWrite('s');
  sum+='s';
  IMUSerialWrite('n');
  sum+='n';
  IMUSerialWrite('p');
  sum+='p';
  // Packet type
  IMUSerialWrite(ZERO_RATE_GYROS);
  sum+=ZERO_RATE_GYROS;
  // Number of data bytes
  IMUSerialWrite((byte)0);
  sum+=(byte)0;
  // Checksum = Sum of all preceding bytes in packet
  IMUSerialWrite((byte)((sum >> 8) & 0x0FF));
  IMUSerialWrite((byte)((sum) & 0x0FF));

  delay(200);

  // ---------- Zero Accelerometers -----------------
  // clear incoming buffer
  IMUSerialFlush();
  sum = 0;
  // Packet header
  IMUSerialWrite('s');
  sum+='s';
  IMUSerialWrite('n');
  sum+='n';
  IMUSerialWrite('p');
  sum+='p';
  // Packet type
  IMUSerialWrite(AUTO_SET_ACCEL_REF);
  sum+=AUTO_SET_ACCEL_REF;
  // Number of data bytes
  IMUSerialWrite((byte)0);
  sum+=(byte)0;
  // Checksum = Sum of all preceding bytes in packet
  IMUSerialWrite((byte)((sum >> 8) & 0x0FF));
  IMUSerialWrite((byte)((sum) & 0x0FF));

  IMUSerialFlush();
  return noerror;
}

/*****************************************************************
Function: IMU_data_request()
Purpose:  Sends data request to CHRobotics IMU
*****************************************************************/
void IMU_data_request(void){
  // clear incoming buffer
  IMUSerialFlush();
  int sum = 0;
  // Packet header
  IMUSerialWrite('s');
  sum+='s';
  IMUSerialWrite('n');
  sum+='n';
  IMUSerialWrite('p');
  sum+='p';
  // Packet type
  IMUSerialWrite(GET_DATA);
  sum+=GET_DATA;
  // Number of data bytes
  IMUSerialWrite((byte)0);
  sum+=(byte)0;
  // Checksum = Sum of all preceding bytes in packet
  IMUSerialWrite((byte)((sum >> 8) & 0x0FF));
  IMUSerialWrite((byte)((sum) & 0x0FF));
}

/*****************************************************************
Function: IMU_data_read()
Purpose:  Sends data request to CHRobotics IMU
*****************************************************************/
void IMU_data_read(void){

  // Wait for serial data
  while(IMUSerialAvailable()<20){
  }

  newIMUdata = true;

  // Read Data Packet
  byte data[12];
  int num;
  byte header;
  header = IMUSerialRead(); // byte 1 ('s')
  header = IMUSerialRead(); // byte 2 ('n')
  header = IMUSerialRead(); // byte 3 ('p')
  header = IMUSerialRead(); // byte 4 (Sensor Data)
  header = IMUSerialRead(); // byte 5 (Number of data bytes)
  header = IMUSerialRead(); // byte 6 (active channels)
  header = IMUSerialRead(); // byte 7 (active channels)
```

```
  // DATA BYTES
  data[0]=IMUSerialRead(); // Yaw data byte 1
  data[1]=IMUSerialRead(); // Yaw data byte 2
  num = (int)data[0]<<8;
  num |= data[1];
  yaw = (num*YAW_FACTOR)+yaw_offset;

  data[2]=IMUSerialRead(); // Pitch data byte 1
  data[3]=IMUSerialRead(); // Pitch data byte 2
  num = (int)data[2]<<8;
  num |= data[3];
  pitch = -1.0*(num*PITCH_FACTOR)+pitch_offset;
  //Serial.println(pitch);

  data[4]=IMUSerialRead(); // Roll data byte 1
  data[5]=IMUSerialRead(); // Roll data byte 2
  num = (int)data[4]<<8;
  num |= data[5];
  roll = (num*ROLL_FACTOR)+roll_offset;

  data[6]=IMUSerialRead(); // Yaw Rate data byte 1
  data[7]=IMUSerialRead(); // Yaw Rate data byte 2
  num = (int)data[6]<<8;
  num |= data[7];
  yawrate = (num*YAW_RATE_FACTOR)+yawrate_offset;

  data[8]=IMUSerialRead(); // Pitch Rate data byte 1
  data[9]=IMUSerialRead(); // Pitch Rate data byte 2
  num = (int)data[8]<<8;
  num |= data[9];
  pitchrate = -1*(num*PITCH_RATE_FACTOR)+pitchrate_offset;

  data[10]=IMUSerialRead(); // Roll Rate data byte 1
  data[11]=IMUSerialRead(); // Roll Rate data byte 2
  num = (int)data[10]<<8;
  num |= data[11];
  rollrate = (num*ROLL_RATE_FACTOR)+rollrate_offset;

  header = IMUSerialRead(); // Checksum
  header = IMUSerialRead(); // Checksum

  // clear incoming buffer
  IMUSerialFlush();
}

/*************************************************************************
Function: IMU_data()
Purpose:  Gets data from CHRobotics IMU
          - places result into IMU data variables
*************************************************************************/
void IMU_data(void){
  uint16_t check = 0;
  uint16_t checksum = 0;
  byte header;
  while(IMUSerialAvailable()>20){
    if(IMUSerialRead() == 's'){
      check += 115;
      if(IMUSerialRead() =='n'){
        check += 110;
        if(IMUSerialRead() == 'p'){
          check += 112;
          packetfound = true;
        }
      }
    }
  }
  if(packetfound && IMUSerialAvailable()>17){
    // Found complete Packet Header
    // Read Data Packet
    byte data[12];
    int num;
    header = IMUSerialRead(); // byte 4 (Sensor Data)
    check += header;
    header = IMUSerialRead(); // byte 5 (Number of data bytes)
    check += header;
    header = IMUSerialRead(); // byte 6 (active channels)
    check += header;
    header = IMUSerialRead(); // byte 7 (active channels)
    check += header;
    // DATA BYTES
    data[0]=IMUSerialRead(); // Yaw data byte 1
    check += data[0];
    data[1]=IMUSerialRead(); // Yaw data byte 2
    check += data[1];
    num = (int)data[0]<<8;
    num |= data[1];
    yaw = (num*YAW_FACTOR)+yaw_offset;

    data[2]=IMUSerialRead(); // Pitch data byte 1
    check += data[2];
    data[3]=IMUSerialRead(); // Pitch data byte 2
    check += data[3];
```

```
    num = (int)data[2]<<8;
    num |= data[3];
    pitch = -1.0*(num*PITCH_FACTOR)+pitch_offset;

    data[4]=IMUSerialRead(); // Roll data byte 1
    check += data[4];
    data[5]=IMUSerialRead(); // Roll data byte 2
    check += data[5];
    num = (int)data[4]<<8;
    num |= data[5];
    roll = (num*ROLL_FACTOR)+roll_offset;

    data[6]=IMUSerialRead(); // Yaw Rate data byte 1
    check += data[6];
    data[7]=IMUSerialRead(); // Yaw Rate data byte 2
    check += data[7];
    num = (int)data[6]<<8;
    num |= data[7];
    yawrate = (num*YAW_RATE_FACTOR)+yawrate_offset;

    data[8]=IMUSerialRead(); // Pitch Rate data byte 1
    check += data[8];
    data[9]=IMUSerialRead(); // Pitch Rate data byte 2
    check += data[9];
    num = (int)data[8]<<8;
    num |= data[9];
    pitchrate = -1.0*(num*PITCH_RATE_FACTOR)+pitchrate_offset;

    data[10]=IMUSerialRead(); // Roll Rate data byte 1
    check += data[10];
    data[11]=IMUSerialRead(); // Roll Rate data byte 2
    check += data[11];
    num = (int)data[10]<<8;
    num |= data[11];
    rollrate = (num*ROLL_RATE_FACTOR)+rollrate_offset;

    // Read final checksum
    checksum = (int)(IMUSerialRead()<<8); // Read final checksum
    checksum |= IMUSerialRead(); // Read final checksum
    if(check == checksum){
      newIMUdata = true;
    }
    packetfound = false;
  }
}

/**************************************************************************
Function: IMU_restart()
Purpose:  Restarts the CHRobotics IMU calculations
          - (if passes too closely to singularity)
Returns:  boolean noerror = if IMU restarted successfully
**************************************************************************/
boolean IMU_restart(void){
  boolean noerror = true;
  // ---------- IMU Restart (EKF Reset) -----------------
  // clear incoming buffer
  IMUSerialFlush();
  int sum = 0;
  // Packet header
  IMUSerialWrite('s');
  sum+='s';
  IMUSerialWrite('n');
  sum+='n';
  IMUSerialWrite('p');
  sum+='p';
  // Packet type
  IMUSerialWrite(0x95);
  sum+=0x95;
  // Number of data bytes
  IMUSerialWrite((byte)0);
  sum+=(byte)0;
  // Checksum = Sum of all preceding bytes in packet
  IMUSerialWrite((byte)((sum >> 8) & 0x0FF));
  IMUSerialWrite((byte)((sum) & 0x0FF));

  //  Read Command Complete Packet
  byte commandcomp[COMMAND_COMPLETE_SIZE];
  int i = 0;
  // Wait for serial data
  while(IMUSerialAvailable()<COMMAND_COMPLETE_SIZE);
  // Fill array with packet data
  while(IMUSerialAvailable()){
     commandcomp[i] = IMUSerialRead();
     i++;
  }
  // Check Packet Contents
  if(commandcomp[3] != COMMAND_COMPLETE){
    // Wrong packet type received
    noerror = false;
  }
  //SerPriln("IMU Reset");
  return noerror;
```

```
}

/*************************************************************************
Function: IMU_write()
Purpose:  Writes previous commands to flash
Returns:  boolean noerror = if IMU wrote to flash successfully
*************************************************************************/
boolean IMU_write(void){
  boolean noerror = true;
  // clear incoming buffer
  IMUSerialFlush();
  int sum = 0;
  // Packet header
  IMUSerialWrite('s');
  sum+='s';
  IMUSerialWrite('n');
  sum+='n';
  IMUSerialWrite('p');
  sum+='p';
  // Packet type
  IMUSerialWrite(WRITE_TO_FLASH);
  sum+=WRITE_TO_FLASH;
  // Number of data bytes
  IMUSerialWrite((byte)0);
  sum+=(byte)0;
  // Checksum = Sum of all preceding bytes in packet
  IMUSerialWrite((byte)((sum >> 8) & 0x0FF));
  IMUSerialWrite((byte)((sum) & 0x0FF));

  // Read Command Complete Packet
  byte commandcomp[COMMAND_COMPLETE_SIZE];
  int i = 0;
  // Wait for serial data
  while(IMUSerialAvailable()<COMMAND_COMPLETE_SIZE);
  // Fill array with packet data
  while(IMUSerialAvailable()){
     commandcomp[i] = IMUSerialRead();
     i++;
  }
  // Check Packet Contents
  if(commandcomp[3] != COMMAND_COMPLETE){
    // Wrong packet type received
    noerror = false;
  }
  return noerror;
}

/*************************************************************************
Function: IMU_silent()
Purpose:  Set IMU to Silent Mode
Returns:  boolean noerror = if IMU was set to silent mode successfully
*************************************************************************/
boolean IMU_silent(void){
  boolean noerror = true;
  // clear incoming buffer
  IMUSerialFlush();
  int sum = 0;
  // Packet header
  IMUSerialWrite('s');
  sum+='s';
  IMUSerialWrite('n');
  sum+='n';
  IMUSerialWrite('p');
  sum+='p';
  // Packet type
  IMUSerialWrite(SET_SILENT_MODE);
  sum+=SET_SILENT_MODE;
  // Number of data bytes
  IMUSerialWrite((byte)0);
  sum+=(byte)0;
  // Checksum = Sum of all preceding bytes in packet
  IMUSerialWrite((byte)((sum >> 8) & 0x0FF));
  IMUSerialWrite((byte)((sum) & 0x0FF));

  // Read Command Complete Packet
  int i = 0;
  byte commandcomp[COMMAND_COMPLETE_SIZE];
  // Wait for serial data
  while(IMUSerialAvailable()<COMMAND_COMPLETE_SIZE);
  // Fill array with packet data
  while(IMUSerialAvailable()){
     commandcomp[i] = IMUSerialRead();
     i++;
  }
  // Check Packet Contents
  if(commandcomp[3] != COMMAND_COMPLETE){
    // Wrong packet type received
    noerror = false;
  }
  return noerror;
}
```

```
/*********************************************************************
Function: IMU_broadcast()
Purpose:  Set IMU to Broadcast Mode
Returns:  boolean noerror = if IMU was set to broadcast mode successfully
*********************************************************************/
boolean IMU_broadcast(void){
  boolean noerror = true;
  // clear incoming buffer
  IMUSerialFlush();
  int sum = 0;
  // Packet header
  IMUSerialWrite('s');
  sum+='s';
  IMUSerialWrite('n');
  sum+='n';
  IMUSerialWrite('p');
  sum+='p';
  // Packet type
  IMUSerialWrite(0x82);
  sum+=0x82;
  // Number of data bytes
  IMUSerialWrite(0x01);
  sum+=0x01;
  // Data byte - Set Broadcast to 300Hz
  IMUSerialWrite(0xFF);
  sum+=0xFF;
  // Checksum = Sum of all preceding bytes in packet
  IMUSerialWrite((byte)((sum >> 8) & 0x0FF));
  IMUSerialWrite((byte)((sum) & 0x0FF));

  // Read Command Complete Packet
  int i = 0;
  byte commandcomp[COMMAND_COMPLETE_SIZE];
  // Wait for serial data
  while(IMUSerialAvailable()<COMMAND_COMPLETE_SIZE);
  // Fill array with packet data
  while(IMUSerialAvailable()){
     commandcomp[i] = IMUSerialRead();
     i++;
  }
  // Check Packet Contents
  if(commandcomp[3] != COMMAND_COMPLETE){
    // Wrong packet type received
    noerror = false;
  }
  return noerror;
}

/*********************************************************************
Function: IMU_GyroFilter()
Purpose:  IMU Low Pass Filter on Gyro Data
*********************************************************************/
void IMU_RateFilter(void) {
  // Filter Gyro Values
  yawrate = (1.0-ratefilter_alpha)*yawrate_old + yawrate*ratefilter_alpha;
  pitchrate = (1.0-ratefilter_alpha)*pitchrate_old + pitchrate*ratefilter_alpha;
  rollrate = (1.0-ratefilter_alpha)*rollrate_old + rollrate*ratefilter_alpha;
  // Set current values to previous values
  yawrate_old = yawrate;
  pitchrate_old = pitchrate;
  rollrate_old = rollrate;
}
```

ArduPilot Mega IMU Code Version 2

```
/*****************************************************************
 R.A.V.E.N. Quadrotor - November 2010
 www.AirHacks.com
 Copyright (c) 2010.  All rights reserved.

 Version: 1.2
 December 15, 2010

 Authors:
       William Etter (UPenn EE 'll)
       Paul Martin (UPenn EE 'll)

 This program is free software: you can redistribute it and/or modify
 it under the terms of the GNU General Public License as published by
 the Free Software Foundation, either version 3 of the License, or
 (at your option) any later version.

 This program is distributed in the hope that it will be useful,
 but WITHOUT ANY WARRANTY; without even the implied warranty of
 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 GNU General Public License for more details.

 You should have received a copy of the GNU General Public License
 along with this program. If not, see <http://www.gnu.org/licenses/>.
```

```
*****************************************************************/

// TO DO
//  - Get correct hex value for ZERO_RATE_GYROS and put into Calibrate function
//  - Do something if IMU didn't initialize successfully
//  - Check if more/less delay is required during startup


/*****************************************************************
//        IMU SERIAL SETUP
*****************************************************************/
// Baud Rate = 115,200
#define IMUSerialBaud  115200
#define IMUSerialAvailable  Serial2.available
#define IMUSerialWrite  Serial2.write
#define IMUSerialRead  Serial2.read
#define IMUSerialFlush  Serial2.flush
#define IMUSerialBegin  Serial2.begin

/*****************************************************************
//        PACKET STRUCTURE
*****************************************************************/
// Function 's' 'n' 'p' PT N d1 ... dN     CHK
// Byte      1   2   3  4  5 6  ... N+5  N+6 N+7
// RX Packet Description
// 1 - 3 Each received packet must begin with the three-byte (character) sequence
// "snp" to signal the beginning of a new packet.
// 4 PT specifies the packet type.
// 5 N specifies the number of data bytes to expect.
// 6 - (N+5) d1 through dN contain the N data bytes in the packet.
// (N+6) - (N+7) CHK is a two-byte checksum.

/*****************************************************************
//        CHR-6dm AHRS Value Conversion Factors
*****************************************************************/
#define YAW_FACTOR 0.0109863F         //  /LSB
#define PITCH_FACTOR 0.0109863F       //  /LSB
#define ROLL_FACTOR 0.0109863F        //  /LSB
#define YAW_RATE_FACTOR 0.0137329F    //  /s/LSB
#define PITCH_RATE_FACTOR 0.0137329F  //  /s/LSB
#define ROLL_RATE_FACTOR 0.0137329F   //  /s/LSB
#define MAGX_FACTOR 0.061035F         //  mGauss/LSB
#define MAGY_FACTOR 0.061035F         //  mGauss/LSB
#define MAGZ_FACTOR 0.061035F         //  mGauss/LSB
#define GYROX_FACTOR 0.01812F         //  /s/LSB
#define GYROY_FACTOR 0.01812F         //  /s/LSB
#define GYROZ_FACTOR 0.01812F         //  /s/LSB
#define ACCELX_FACTOR 0.106812F       //  mg/LSB
#define ACCELY_FACTOR 0.106812F       //  mg/LSB
#define ACCELZ_FACTOR 0.106812F       //  mg/LSB

/*****************************************************************
//        IMU RX PACKET DEFINITIONS (-> IMU)
*****************************************************************/
#define SET_ACTIVE_CHANNELS 0x80    // Specifies which channel data should be transmitted over the UART.
#define SET_SILENT_MODE 0x81        // Enables "Silent Mode." In Silent Mode, the AHRS only reports data when
                                    // a GET_DATA packet is received.
#define SET_BROADCAST_MODE 0x82     // Enables "Broadcast Mode." In Broadcast Mode, the AHRS automatically
                                    // transmits sensor data every Ts milliseconds, where Ts is encoded in
                                    // the data section of the SET_BROADCAST_MODE packet.
#define SET_GYRO_BIAS 0x83          // Manually sets the x-axis rate gyro bias term. The bias term can be
                                    // automatically set for all gyro axes by sending a ZERO_RATE_GYROS packet.
#define SET_ACCEL_BIAS 0x84         // Manually sets the x-axis accelerometer bias term.
#define SET_ACCEL_REF_VECTOR 0x85   // Manually sets the accelerometer reference vector used by the EKF to determine
                                    // "which way is down"
#define AUTO_SET_ACCEL_REF 0x86     // Causes the AHRS to set the current accelerometer measurements as the reference
                                    // vector (sets pitch and roll angles to zero for the given orientation).
#define ZERO_RATE_GYROS 0x87        // Starts internal self-calibration of all three rate gyro axes. By default, rate
                                    // gyros are zeroed on AHRS startup, but gyro startup calibration can be disabled
                                    // (or re-enabled) by sending a SET_START_CAL packet.
#define SELF_TEST 0x88              // Instructs the AHRS to perform a self-test of all sensor channels. A STATUS_REPORT
                                    // packet is transmitted after the self-test is complete.
#define SET_START_CAL 0x89          // Enables or disables automatic startup calibration of rate gyro biases.
#define SET_PROCESS_COVARIANCE 0x8A // Sets the 3x3 matrix representing the covariance of the process noise used in the
                                    // prediction step of the EKF.
#define SET_MAG_COVARIANCE 0X8B     // Sets the 3x3 matrix representing the covariance of the measurement noise used in
                                    // the magnetometer update step of the EKF
#define SET_ACCEL_COVARIANCE 0x8C   // Sets the 3x3 matrix representing the covariance of the measurement noise used in
                                    // the accelerometer update step of the EKF
#define SET_EKF_CONFIG 0x8D         // Sets the EKF_CONFIG register. Can be used to enable/disable the EKF, or to enable/disable
                                    // accelerometer and magnetometer updates.
#define SET_GYRO_ALIGNMENT 0x8E     // Sets the 3x3 matrix used to correct rate gyro crossaxis misalignment.
#define SET_ACCEL_ALIGNMENT 0x8F    // Sets the 3x3 matrix used to correct accelerometer cross-axis misalignment
#define SET_MAG_REF_VECTOR 0x90     // Sets the reference vector representing the expected output of the magnetometer when yaw,
                                    // pitch, and roll angles are zero.
#define AUTO_SET_MAG_REF 0x91       // Sets the current magnetometer output as the reference vector.
#define SET_MAG_CAL 0x92            // Sets the 3x3 magnetic field distortion correction matrix to compensate for soft-iron
                                    // distortions, axis misalignment, and sensor scale inconsistencies.
#define SET_MAG_BIAS 0x93           // Sets the magnetic field measurement bias to compensate for hard iron distortions
#define SET_GYRO_SCALE 0x94         // Sets the scale factors used to convert raw ADC data to rates for angle estimation on the EKF.
#define EKF_RESET 0x95              // Sets all terms in the current state covariance matrix to zero and re-initializes angle estimates.
#define RESET_TO_FACTORY 0x96       // Resets all AHRS setting to factory default values
```

```
#define WRITE_TO_FLASH 0xA0         // Writes current AHRS configuration to on-board flash, so that the configuration persists
                                    // when the power is cycled.
#define GET_DATA 0x01               // In Listen Mode, causes the AHRS to transmit data from all active channels in a SENSOR_DATA...
#define GET_ACTIVE_CHANNELS 0x02    // Reports which channels are "active" in an ACTIVE_CHANNELS_REPORT packet...
#define GET_BROADCAST_MODE 0x03     // Returns the BROADCAST_MODE_REPORT packet, which specifies whether...
#define GET_ACCEL_BIAS 0x04         // Return the bias values for all three accel axes in a ACCEL_BIAS_REPORT packet.
#define GET_ACCEL_REF_VECTOR 0x05   // Returns the accelerometer reference vector in an ACCEL_REF_VECTOR_REPORT...
#define GET_GYRO_BIAS 0x06          // Returns the bias values for all three rate gyros in a GYRO_BIAS_REPORT packet.
#define GET_GYRO_SCALE 0x07         // Returns the rate gyro scale factors used internally to convert raw rate gyro data to actual...
                                    // GYRO_SCALE_REPORT packet.
#define GET_START_CAL 0x08          // Reports whether the AHRS is configured to calibrate rate gyro biases automatically on...
                                    // START_CAL_REPORT packet.
#define GET_EKF_CONFIG 0x09         // Returns the one byte EKF configuration register in an EKF_CONFIG_REPORT packet.
#define GET_ACCEL_COVARIANCE 0X0A   // Returns the variance of the accelerometer measurements used by the EKF...
#define GET_MAG_COVARIANCE 0x0B     // Returns the variance of the magnetometer measurements used by the EKF...
#define GET_PROCESS_COVARIANCE 0x0C // Returns the variance of the EKF prediction step
#define GET_STATE_COVARIANCE 0x0D   // Returns the 3x3 matrix representing the covariance of the current EKF...
                                    // STATE_COVARIANCE_REPORT packet.
#define GET_GYRO_ALIGNMENT 0x0E     // Returns the 3x3 matrix used to correct rate gyro cross-axis misalignment
#define GET_ACCEL_ALIGNMENT 0x0F    // Returns the 3x3 matrix used to correct accelerometer cross-axis misalignment
#define GET_MAG_REF_VECTOR 0x10     // Returns the magnetometer reference vector in a MAG_REF_VECTOR_REPORT...
#define GET_MAG_CAL Returns 0x11    // the 3x3 magnetometer correction matrix in an MAG_CAL_REPORT packet
#define GET_MAG_BIAS 0x12           // Returns the magnetometer bias correction used by the EKF. The bias is reported in a
                                    // MAG_BIAS_REPORT packet.

/*************************************************************************
//        IMU TX PACKET DEFINITIONS (IMU ->)
*************************************************************************/
#define COMMAND_COMPLETE 0xB0       // Transmitted by the AHRS upon successful completion of a command...
                                    // data to be returned.
#define COMMAND_COMPLETE_SIZE 8     // 8 Bytes in COMMAND_COMPLETE Packet
#define COMMAND_FAILED 0xB1         // Transmitted by the AHRS when a command received over the UART could...
#define BAD_CHECKSUM 0xB2           // Transmitted by the AHRS when a received packet checksum does no...
                                    // other bytes in the packet.
#define BAD_DATA_LENGTH 0xB3        // Transmitted by the AHRS when a received packet contained more or...
                                    // expected for a given packet type.
#define UNRECOGNIZED_PACKET 0xB4    // Transmitted if the AHRS receives an unrecognized packet.
#define BUFFER_OVERFLOW 0xB5        // Transmitted by the AHRS when the internal receive buffer overflows before...
#define STATUS_REPORT 0xB6          // Transmitted at the end of a self-test procedure, triggered by a SELF_TEST...
#define STATUS_REPORT_SIZE 8        // 8 Bytes in STATUS_REPORT Packet
#define SENSOR_DATA 0xB7            // Sent in response to a GET_DATA packet, or sent automatically in Broadcast Mode.
#define GYRO_BIAS_REPORT 0xB8       // Sent in response to the GET_GYRO_BIAS command.
#define GYRO_SCALE_REPORT 0xB9      // Sent in response to a GET_GYRO_SCALE command.
#define START_CAL_REPORT 0xBA       // Sent in response to a GET_START_CAL command.
#define ACCEL_BIAS_REPORT 0xBB      // Sent in response to the GET_ACCEL_BIAS command.
#define ACCEL_REF_VECTOR_REPORT 0xBC // Sent in response to a GET_ACCEL_REF_VECTOR command, or ...
#define ACCEL_REF_VECTOR_REPORT_SIZE 13 // 13 Bytes in ACCEL_REF_VECTOR_REPORT Packet
#define ACTIVE_CHANNEL_REPORT 0xBD   // Sent in response to a GET_ACTIVE_CHANNELS command.
#define ACCEL_COVARIANCE_REPORT 0xBE // Sent in response to a GET_ACCEL_COVARIANCE command
#define MAG_COVARIANCE_REPORT 0xBF   // Sent in response to a GET_MAG_COVARIANCE command.
#define PROCESS_COVARIANCE_REPORT 0xC0 //Sent in response to a GET_PROCESS_COVARIANCE command
#define STATE_COVARIANCE_REPORT 0xC1 // Sent in response to a GET_STATE_COVARIANCE command
#define EKF_CONFIG_REPORT 0xC2       // Sent in response to a GET_EKF_CONFIG command.
#define GYRO_ALIGNMENT_REPORT 0xC3   // Sent in response to a GET_GYRO_ALIGNMENT command.
#define ACCEL_ALIGNMENT_REPORT 0xC4  // Sent in response to a GET_ACCEL_ALIGNMENT command.
#define MAG_REF_VECTOR_REPORT 0xC5   // Sent in response to a GET_MAG_REF_VECTOR command
#define MAG_CAL_REPORT 0xC6          // Sent in response to a GET_MAG_CAL command
#define MAG_BIAS_REPORT 0xC7         // Sent in response to a GET_MAG_BIAS command
#define BROADCAST_MODE_REPORT 0XC8   // Sent in response to a GET_BROADCAST_MODE command.

/*************************************************************************
Function: IMU_init()
Purpose:  Run self-test
          Set values to output
Returns:  boolean IMUinitialization = if IMU initialized successfully
*************************************************************************/
boolean IMU_init(void){
  // Variables
  boolean IMUinitialization = true;
  int sum = 0;

  // Open Serial Channel, 115,200 Baud Rate
  IMUSerialBegin(IMUSerialBaud);

  // ---------- Run Self-Test -----------------
  // clear incoming buffer
  IMUSerialFlush();
  sum = 0;
  // Packet header
  IMUSerialWrite('s');
  sum+='s';
  IMUSerialWrite('n');
  sum+='n';
  IMUSerialWrite('p');
  sum+='p';
  // Packet type
  IMUSerialWrite(SELF_TEST);
  sum+=SELF_TEST;
  // Number of data bytes
  IMUSerialWrite((byte)0);
  sum+=(byte)0;
  // Checksum = Sum of all preceding bytes in packet
```

```
  IMUSerialWrite((byte)((sum >> 8) & 0x0FF));
  IMUSerialWrite((byte)((sum) & 0x0FF));

  // Read Status Report Packet
  byte statreport[STATUS_REPORT_SIZE];
  int i = 0;
  // Wait for serial data
  while(IMUSerialAvailable()<STATUS_REPORT_SIZE);
  // Fill array with packet data
  while(IMUSerialAvailable()){
     statreport[i] = IMUSerialRead();
     //Serial.println(input[i],HEX);
     i++;
  }
  // Check Packet Contents
  if(statreport[3] != STATUS_REPORT){
    // Wrong packet type received
    // DO SOMETHING HERE
    IMUinitialization = false;
  }
  else if(statreport[5]!=0x00){
    // Error on channel
    // DO SOMETHING HERE
    IMUinitialization = false;
  }

  delay(200);

  // ---------- Set IMU Values Channels to Output-----------------
  // clear incoming buffer
  IMUSerialFlush();
  sum = 0;
  // Packet header
  IMUSerialWrite('s');
  sum+='s';
  IMUSerialWrite('n');
  sum+='n';
  IMUSerialWrite('p');
  sum+='p';
  // Packet type
  IMUSerialWrite(SET_ACTIVE_CHANNELS);
  sum+=SET_ACTIVE_CHANNELS;
  // Number of data bytes
  IMUSerialWrite((byte)2);
  sum+=(byte)2;
  IMUSerialWrite(0xFC); // Activate estimates and rates for Yaw, Pitch, and Roll
  sum+=0xFC;
  IMUSerialWrite(0x0E); // Activate accelerometer data in X, Y, and Z
  sum+=0x0E;
  // Checksum = Sum of all preceding bytes in packet
  IMUSerialWrite((byte)((sum >> 8) & 0x0FF));
  IMUSerialWrite((byte)((sum) & 0x0FF));

  // Read Command Complete Packet
  i = 0;
  byte commandcomp[COMMAND_COMPLETE_SIZE];
  // Wait for serial data
  while(IMUSerialAvailable()<COMMAND_COMPLETE_SIZE);
  // Fill array with packet data
  while(IMUSerialAvailable()){
     commandcomp[i] = IMUSerialRead();
     // DO SOMETHING HERE
     i++;
  }
  // Check Packet Contents
  if(commandcomp[3] != COMMAND_COMPLETE){
    // Wrong packet type received
    //  DO SOMETHING HERE
    IMUinitialization = false;
  }

  return IMUinitialization;
}

/*********************************************************************
Function: IMU_calibrate()
Purpose:  Calibrates IMU
            - Zero Rate Gyros
            - Zero Accelerometer Measurements
Returns:  boolean noerror = if IMU calibrated successfully
*********************************************************************/
boolean IMU_calibrate(void){
  boolean noerror = true;
  // ---------- Zero Rate Gyros -----------------
  // clear incoming buffer
  IMUSerialFlush();
  int sum = 0;
  // Packet header
  IMUSerialWrite('s');
  sum+='s';
  IMUSerialWrite('n');
  sum+='n';
```

```
  IMUSerialWrite('p');
  sum+='p';
  // Packet type

  IMUSerialWrite(0x8B);
  sum+=0x8B;

  ////////////////////////////////////////////////////////
  // NEEDS TO BE FIXED WITH CORRECT VALUE
  //IMUSerialWrite(ZERO_RATE_GYROS);
  //sum+=ZERO_RATE_GYROS;
  ////////////////////////////////////////////////////////

  // Number of data bytes
  IMUSerialWrite((byte)0);
  sum+=(byte)0;
  // Checksum = Sum of all preceding bytes in packet
  IMUSerialWrite((byte)((sum >> 8) & 0x0FF));
  IMUSerialWrite((byte)((sum) & 0x0FF));

  // Read Command Complete Packet
  byte commandcomp[COMMAND_COMPLETE_SIZE];
  int i = 0;
  // Wait for serial data
  while(IMUSerialAvailable()<COMMAND_COMPLETE_SIZE);
  // Fill array with packet data
  while(IMUSerialAvailable()){
     commandcomp[i] = IMUSerialRead();
     //Serial.println(commandcomp[i],HEX);
     i++;
  }
  // Check Packet Contents
  if(commandcomp[3] != COMMAND_COMPLETE){
    // Wrong packet type received
    //SerPriln("IMU Calibration Failed");
    noerror = false;
  }

  // ---------- Zero Accelerometers -----------------
  // clear incoming buffer
  IMUSerialFlush();
  sum = 0;
  // Packet header
  IMUSerialWrite('s');
  sum+='s';
  IMUSerialWrite('n');
  sum+='n';
  IMUSerialWrite('p');
  sum+='p';
  // Packet type
  IMUSerialWrite(AUTO_SET_ACCEL_REF);
  sum+=AUTO_SET_ACCEL_REF;
  // Number of data bytes
  IMUSerialWrite((byte)0);
  sum+=(byte)0;
  // Checksum = Sum of all preceding bytes in packet
  IMUSerialWrite((byte)((sum >> 8) & 0x0FF));
  IMUSerialWrite((byte)((sum) & 0x0FF));

  // Read Command Complete Packet
  byte accelref[ACCEL_REF_VECTOR_REPORT_SIZE];
  i = 0;
  // Wait for serial data
  while(IMUSerialAvailable()<ACCEL_REF_VECTOR_REPORT_SIZE);
  // Fill array with packet data
  while(IMUSerialAvailable()){
     accelref[i] = IMUSerialRead();
     i++;
  }
  // Check Packet Contents
  if(accelref[3] != ACCEL_REF_VECTOR_REPORT){
    // Wrong packet type received
    //  DO SOMETHING HERE
    noerror = false;
  }

  return noerror;
}

/*************************************************************************
Function: IMU_data()
Purpose:  Gets data from CHRobotics IMU
          - places result into IMU data variables
*************************************************************************/
void IMU_data(void){
  // clear incoming buffer
  IMUSerialFlush();
  int sum = 0;
  // Packet header
  IMUSerialWrite('s');
  sum+='s';
  IMUSerialWrite('n');
```

```
sum+='n';
IMUSerialWrite('p');
sum+='p';
// Packet type
IMUSerialWrite(GET_DATA);
sum+=GET_DATA;
// Number of data bytes
IMUSerialWrite((byte)0);
sum+=(byte)0;
// Checksum = Sum of all preceding bytes in packet
IMUSerialWrite((byte)((sum >> 8) & 0x0FF));
IMUSerialWrite((byte)((sum) & 0x0FF));

// Read Command Complete Packet
byte input[7];
byte data[27];
int i = 0;

// Wait for serial data
while(IMUSerialAvailable()<27);

// Read Data Packet
int num;
byte header = IMUSerialRead(); // byte 1 ('s')
header = IMUSerialRead(); // byte 2 ('n')
header = IMUSerialRead(); // byte 3 ('p')
header = IMUSerialRead(); // byte 4 (Sensor Data)
header = IMUSerialRead(); // byte 5 (Number of data bytes)
header = IMUSerialRead(); // byte 6 (active channels)
header = IMUSerialRead(); // byte 7 (active channels)
// DATA BYTES
data[0]=IMUSerialRead(); // Yaw data byte 1
data[1]=IMUSerialRead(); // Yaw data byte 2
num = (int)data[0]<<8;
num |= data[1];
yaw = (num*YAW_FACTOR);

data[2]=IMUSerialRead(); // Pitch data byte 1
data[3]=IMUSerialRead(); // Pitch data byte 2
num = (int)data[2]<<8;
num |= data[3];
//Serial.println(num);
pitch = (num*PITCH_FACTOR);

data[4]=IMUSerialRead(); // Roll data byte 1
data[5]=IMUSerialRead(); // Roll data byte 2
num = (int)data[4]<<8;
num |= data[5];
//Serial.println(num);
roll = (num*ROLL_FACTOR);

data[6]=IMUSerialRead(); // Yaw Rate data byte 1
data[7]=IMUSerialRead(); // Yaw Rate data byte 2
num = (int)data[6]<<8;
num |= data[7];
//Serial.println(num);
yawrate = (num*YAW_RATE_FACTOR);

data[8]=IMUSerialRead(); // Pitch Rate data byte 1
data[9]=IMUSerialRead(); // Pitch Rate data byte 2
num = (int)data[8]<<8;
num |= data[9];
//Serial.println(num);
pitchrate = (num*PITCH_RATE_FACTOR);

data[10]=IMUSerialRead(); // Roll Rate data byte 1
data[11]=IMUSerialRead(); // Roll Rate data byte 2
num = (int)data[10]<<8;
num |= data[11];
//Serial.println(num);
rollrate = (num*ROLL_RATE_FACTOR);

data[12]=IMUSerialRead(); // Accel Z data byte 1
data[13]=IMUSerialRead(); // Accel Z data byte 2
num = (int)data[12]<<8;
num |= data[13];
//Serial.println(num);
accelz = (num*ACCELZ_FACTOR);

data[14]=IMUSerialRead(); // Accel Y data byte 1
data[15]=IMUSerialRead(); // Accel Y data byte 2
num = (int)data[14]<<8;
num |= data[15];
//Serial.println(num);
accely = (num*ACCELY_FACTOR);

data[16]=IMUSerialRead(); // Accel X data byte 1
data[17]=IMUSerialRead(); // Accel X data byte 2
num = (int)data[16]<<8;
num |= data[17];
//Serial.println(num);
accelz = (num*ACCELZ_FACTOR);
```

```
}

/*********************************************************************
Function: IMU_restart()
Purpose:  Restarts the CHRobotics IMU calculations
          - (if passes too closely to singularity)
Returns:  boolean noerror = if IMU restarted successfully
*********************************************************************/
boolean IMU_restart(void){
  boolean noerror = true;
  // ---------- IMU Restart (EKF Reset) -----------------
  // clear incoming buffer
  IMUSerialFlush();
  int sum = 0;
  // Packet header
  IMUSerialWrite('s');
  sum+='s';
  IMUSerialWrite('n');
  sum+='n';
  IMUSerialWrite('p');
  sum+='p';
  // Packet type
  IMUSerialWrite(0x95);
  sum+=0x95;
  // Number of data bytes
  IMUSerialWrite((byte)0);
  sum+=(byte)0;
  // Checksum = Sum of all preceding bytes in packet
  IMUSerialWrite((byte)((sum >> 8) & 0x0FF));
  IMUSerialWrite((byte)((sum) & 0x0FF));

  //  Read Command Complete Packet
  byte commandcomp[COMMAND_COMPLETE_SIZE];
  int i = 0;
  // Wait for serial data
  while(IMUSerialAvailable()<COMMAND_COMPLETE_SIZE);
  // Fill array with packet data
  while(IMUSerialAvailable()){
     commandcomp[i] = IMUSerialRead();
     //Serial.println(commandcomp[i],HEX);
     i++;
  }
  // Check Packet Contents
  if(commandcomp[3] != COMMAND_COMPLETE){
    // Wrong packet type received
    //SerPriln("IMU Failed to Restart");
    noerror = false;
  }
  //SerPriln("IMU Reset");
  return noerror;
}

/*********************************************************************
Function: IMU_write()
Purpose:  Writes previous commands to flash
Returns:  boolean noerror = if IMU wrote to flash successfully
*********************************************************************/
boolean IMU_write(void){
  boolean noerror = true;
  // clear incoming buffer
  IMUSerialFlush();
  int sum = 0;
  // Packet header
  IMUSerialWrite('s');
  sum+='s';
  IMUSerialWrite('n');
  sum+='n';
  IMUSerialWrite('p');
  sum+='p';
  // Packet type
  IMUSerialWrite(WRITE_TO_FLASH);
  sum+=WRITE_TO_FLASH;
  // Number of data bytes
  IMUSerialWrite((byte)0);
  sum+=(byte)0;
  // Checksum = Sum of all preceding bytes in packet
  IMUSerialWrite((byte)((sum >> 8) & 0x0FF));
  IMUSerialWrite((byte)((sum) & 0x0FF));

  // Read Command Complete Packet
  byte commandcomp[COMMAND_COMPLETE_SIZE];
  int i = 0;
  // Wait for serial data
  while(IMUSerialAvailable()<COMMAND_COMPLETE_SIZE);
  // Fill array with packet data
  while(IMUSerialAvailable()){
     commandcomp[i] = IMUSerialRead();
     //Serial.println(commandcomp[i],HEX);
     i++;
  }
  // Check Packet Contents
  if(commandcomp[3] != COMMAND_COMPLETE){
```

```
    // Wrong packet type received
    //SerPriln("IMU Write to Flash Failed");
    noerror = false;
  }
  return noerror;
}

/*************************************************************************
Function: IMU_silent()
Purpose:  Set IMU to Silent Mode
Returns:  boolean noerror = if IMU was set to silent mode successfully
*************************************************************************/
boolean IMU_silent(void){
  boolean noerror = true;
  // clear incoming buffer
  IMUSerialFlush();
  int sum = 0;
  // Packet header
  IMUSerialWrite('s');
  sum+='s';
  IMUSerialWrite('n');
  sum+='n';
  IMUSerialWrite('p');
  sum+='p';
  // Packet type
  IMUSerialWrite(SET_SILENT_MODE);
  sum+=SET_SILENT_MODE;
  // Number of data bytes
  IMUSerialWrite((byte)0);
  sum+=(byte)0;
  // Checksum = Sum of all preceding bytes in packet
  IMUSerialWrite((byte)((sum >> 8) & 0x0FF));
  IMUSerialWrite((byte)((sum) & 0x0FF));

  // Read Command Complete Packet
  int i = 0;
  byte commandcomp[COMMAND_COMPLETE_SIZE];
  // Wait for serial data
  while(IMUSerialAvailable()<COMMAND_COMPLETE_SIZE);
  // Fill array with packet data
  while(IMUSerialAvailable()){
     commandcomp[i] = IMUSerialRead();
     // Serial.println(commandcomp[i],HEX);
     i++;
  }
  // Check Packet Contents
  if(commandcomp[3] != COMMAND_COMPLETE){
    // Wrong packet type received
    //SerPriln("Silent Mode Failed");
    noerror = false;
  }
  return noerror;
}
```

ArduPilot Mega IMU Code Version 1

```
/*****************************************************************
 R.A.V.E.N. Quadrotor - November 2010
 www.AirHacks.com
 Copyright (c) 2010.  All rights reserved.

 Version: 1.0
 December 8, 2010

 Authors:
       William Etter (UPenn EE 'll)
       Paul Martin (UPenn EE 'll)

 This program is free software: you can redistribute it and/or modify
 it under the terms of the GNU General Public License as published by
 the Free Software Foundation, either version 3 of the License, or
 (at your option) any later version.

 This program is distributed in the hope that it will be useful,
 but WITHOUT ANY WARRANTY; without even the implied warranty of
 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 GNU General Public License for more details.

 You should have received a copy of the GNU General Public License
 along with this program. If not, see <http://www.gnu.org/licenses/>.
*****************************************************************/



/*************************************************************************
//        IMU SERIAL SETUP
*************************************************************************/
// Baud Rate = 115,200
#define IMUSerialBaud  115200
```

```
#define IMUSerialAvailable  Serial2.available
#define IMUSerialWrite  Serial2.write
#define IMUSerialRead  Serial2.read
#define IMUSerialFlush  Serial2.flush
#define IMUSerialBegin  Serial2.begin



/*****************************************************************************
//        CHR-6dm AHRS Value Conversion Factors
*****************************************************************************/
#define YAW_FACTOR 0.0109863F         //  /LSB
#define PITCH_FACTOR 0.0109863F       //  /LSB
#define ROLL_FACTOR 0.0109863F        //  /LSB
#define YAW_RATE_FACTOR 0.0137329F    //  /s/LSB
#define PITCH_RATE_FACTOR 0.0137329F  //  /s/LSB
#define ROLL_RATE_FACTOR 0.0137329F   //  /s/LSB
#define MAGX_FACTOR 0.061035F         //  mGauss/LSB
#define MAGY_FACTOR 0.061035F         //  mGauss/LSB
#define MAGZ_FACTOR 0.061035F         //  mGauss/LSB
#define GYROX_FACTOR 0.01812F         //  /s/LSB
#define GYROY_FACTOR 0.01812F         //  /s/LSB
#define GYROZ_FACTOR 0.01812F         //  /s/LSB
#define ACCELX_FACTOR 0.106812F       //  mg/LSB
#define ACCELY_FACTOR 0.106812F       //  mg/LSB
#define ACCELZ_FACTOR 0.106812F       //  mg/LSB

/*****************************************************************************
//        IMU RX PACKET DEFINITIONS (-> IMU)
*****************************************************************************/
#define SET_ACTIVE_CHANNELS 0x80
#define SET_SILENT_MODE 0x81
#define SET_BROADCAST_MODE 0x82
#define SET_GYRO_BIAS 0x83
#define SET_ACCEL_BIAS 0x84
#define SET_ACCEL_REF_VECTOR 0x85
#define AUTO_SET_ACCEL_REF 0x86
#define ZERO_RATE_GYROS 0x87
#define SELF_TEST 0x88
#define SET_START_CAL 0x89
#define SET_PROCESS_COVARIANCE 0x8A
#define SET_MAG_COVARIANCE 0X8B
#define SET_ACCEL_COVARIANCE 0x8C
#define SET_EKF_CONFIG 0x8D
#define SET_GYRO_ALIGNMENT 0x8E
#define SET_ACCEL_ALIGNMENT 0x8F
#define SET_MAG_REF_VECTOR 0x90
#define AUTO_SET_MAG_REF 0x91
#define SET_MAG_CAL 0x92
#define SET_MAG_BIAS 0x93
#define SET_GYRO_SCALE 0x94
#define EKF_RESET 0x95
#define RESET_TO_FACTORY 0x96
#define WRITE_TO_FLASH 0xA0
#define GET_DATA 0x01
#define GET_ACTIVE_CHANNELS 0x02
#define GET_BROADCAST_MODE 0x03
#define GET_ACCEL_BIAS 0x04
#define GET_ACCEL_REF_VECTOR 0x05
#define GET_GYRO_BIAS 0x06
#define GET_GYRO_SCALE 0x07
#define GET_START_CAL 0x08
#define GET_EKF_CONFIG 0x09
#define GET_ACCEL_COVARIANCE 0X0A
#define GET_MAG_COVARIANCE 0x0B
#define GET_PROCESS_COVARIANCE 0x0C
#define GET_STATE_COVARIANCE 0x0D
#define GET_GYRO_ALIGNMENT 0x0E
#define GET_ACCEL_ALIGNMENT 0x0F
#define GET_MAG_REF_VECTOR 0x10
#define GET_MAG_CAL Returns 0x11
#define GET_MAG_BIAS 0x12

/*****************************************************************************
//        IMU TX PACKET DEFINITIONS (IMU ->)
*****************************************************************************/
#define COMMAND_COMPLETE 0xB0
#define COMMAND_COMPLETE_SIZE 8
#define COMMAND_FAILED 0xB1
#define BAD_CHECKSUM 0xB2
#define BAD_DATA_LENGTH 0xB3
#define UNRECOGNIZED_PACKET 0xB4
#define BUFFER_OVERFLOW 0xB5
#define STATUS_REPORT 0xB6
#define STATUS_REPORT_SIZE 8
#define SENSOR_DATA 0xB7
#define GYRO_BIAS_REPORT 0xB8
#define GYRO_SCALE_REPORT 0xB9
#define START_CAL_REPORT 0xBA
#define ACCEL_BIAS_REPORT 0xBB
#define ACCEL_REF_VECTOR_REPORT 0xBC
#define ACCEL_REF_VECTOR_REPORT_SIZE 13
```

```
#define ACTIVE_CHANNEL_REPORT 0xBD
#define ACCEL_COVARIANCE_REPORT 0xBE
#define MAG_COVARIANCE_REPORT 0xBF
#define PROCESS_COVARIANCE_REPORT 0xC0
#define STATE_COVARIANCE_REPORT 0xC1
#define EKF_CONFIG_REPORT 0xC2
#define GYRO_ALIGNMENT_REPORT 0xC3
#define ACCEL_ALIGNMENT_REPORT 0xC4
#define MAG_REF_VECTOR_REPORT 0xC5
#define MAG_CAL_REPORT 0xC6
#define MAG_BIAS_REPORT 0xC7
#define BROADCAST_MODE_REPORT 0XC8

/************************************************************************
Function: IMU_init()
Purpose:  Run self-test
          Set values to output
Returns:  boolean IMUinitialization = if IMU initialized successfully
************************************************************************/
boolean IMU_init(void){
  // Variables
  boolean IMUinitialization = true;
  int sum = 0;

  // Open Serial Channel, 115,200 Baud Rate
  IMUSerialBegin(IMUSerialBaud);


  IMUSerialFlush();
  sum = 0;
  // Packet header
  IMUSerialWrite('s');
  sum+='s';
  IMUSerialWrite('n');
  sum+='n';
  IMUSerialWrite('p');
  sum+='p';
  // Packet type
  IMUSerialWrite(SELF_TEST);
  sum+=SELF_TEST;
  // Number of data bytes
  IMUSerialWrite((byte)0);
  sum+=(byte)0;
  // Checksum = Sum of all preceding bytes in packet
  IMUSerialWrite((byte)((sum >> 8) & 0x0FF));
  IMUSerialWrite((byte)((sum) & 0x0FF));

  // Read Status Report Packet
  byte statreport[STATUS_REPORT_SIZE];
  int i = 0;
  // Wait for serial data
  while(IMUSerialAvailable()<STATUS_REPORT_SIZE);
  // Fill array with packet data
  while(IMUSerialAvailable()){
     statreport[i] = IMUSerialRead();
     //Serial.println(input[i],HEX);
     i++;
  }
  // Check Packet Contents
  if(statreport[3] != STATUS_REPORT){
    // Wrong packet type received
    // DO SOMETHING HERE
    IMUinitialization = false;
  }
  else if(statreport[5]!=0x00){
    // Error on channel
    // DO SOMETHING HERE
    IMUinitialization = false;
  }


  // Read Command Complete Packet
  i = 0;
  byte commandcomp[COMMAND_COMPLETE_SIZE];
  // Wait for serial data
  while(IMUSerialAvailable()<COMMAND_COMPLETE_SIZE);
  // Fill array with packet data
  while(IMUSerialAvailable()){
     commandcomp[i] = IMUSerialRead();
     // DO SOMETHING HERE
     i++;
  }
  // Check Packet Contents
  if(commandcomp[3] != COMMAND_COMPLETE){
    // Wrong packet type received
    //  DO SOMETHING HERE
    IMUinitialization = false;
  }

  return IMUinitialization;
}
```

```
/**********************************************************************
Function: IMU_data()
**********************************************************************/
void IMU_data(void){
  // clear incoming buffer
  IMUSerialFlush();
  int sum = 0;
  // Packet header
  IMUSerialWrite('s');
  sum+='s';
  IMUSerialWrite('n');
  sum+='n';
  IMUSerialWrite('p');
  sum+='p';
  // Packet type
  IMUSerialWrite(GET_DATA);
  sum+=GET_DATA;
  // Number of data bytes
  IMUSerialWrite((byte)0);
  sum+=(byte)0;
  // Checksum = Sum of all preceding bytes in packet
  IMUSerialWrite((byte)((sum >> 8) & 0x0FF));
  IMUSerialWrite((byte)((sum) & 0x0FF));

  // Read Command Complete Packet
  byte input[7];
  byte data[27];
  int i = 0;

  // Wait for serial data
  while(IMUSerialAvailable()<27);

  // Read Data Packet
  int num;
  byte header = IMUSerialRead(); // byte 1 ('s')

  // DATA BYTES
  data[0]=IMUSerialRead(); // Yaw data byte 1
  data[1]=IMUSerialRead(); // Yaw data byte 2
  num = (int)data[0]<<8;
  num |= data[1];
  yaw = (num*YAW_FACTOR);

  data[2]=IMUSerialRead(); // Pitch data byte 1
  data[3]=IMUSerialRead(); // Pitch data byte 2
  num = (int)data[2]<<8;
  num |= data[3];
  pitch = (num*PITCH_FACTOR);

  data[4]=IMUSerialRead(); // Roll data byte 1
  data[5]=IMUSerialRead(); // Roll data byte 2
  num = (int)data[4]<<8;
  num |= data[5];
  roll = (num*ROLL_FACTOR);

  data[6]=IMUSerialRead(); // Yaw Rate data byte 1
  data[7]=IMUSerialRead(); // Yaw Rate data byte 2
  num = (int)data[6]<<8;
  num |= data[7];
  yawrate = (num*YAW_RATE_FACTOR);

  data[8]=IMUSerialRead(); // Pitch Rate data byte 1
  data[9]=IMUSerialRead(); // Pitch Rate data byte 2
  num = (int)data[8]<<8;
  num |= data[9];
  pitchrate = (num*PITCH_RATE_FACTOR);

  data[10]=IMUSerialRead(); // Roll Rate data byte 1
  data[11]=IMUSerialRead(); // Roll Rate data byte 2
  num = (int)data[10]<<8;
  num |= data[11];
  rollrate = (num*ROLL_RATE_FACTOR);

  data[12]=IMUSerialRead(); // Accel Z data byte 1
  data[13]=IMUSerialRead(); // Accel Z data byte 2
  num = (int)data[12]<<8;
  num |= data[13];
  accelz = (num*ACCELZ_FACTOR);

  data[14]=IMUSerialRead(); // Accel Y data byte 1
  data[15]=IMUSerialRead(); // Accel Y data byte 2
  num = (int)data[14]<<8;
  num |= data[15];
  accely = (num*ACCELY_FACTOR);

  data[16]=IMUSerialRead(); // Accel X data byte 1
  data[17]=IMUSerialRead(); // Accel X data byte 2
  num = (int)data[16]<<8;
  num |= data[17];
  accelz = (num*ACCELZ_FACTOR);
}
```

```
/*************************************************************************
Function: IMU_write()
Purpose:  Writes previous commands to flash
Returns:  boolean noerror = if IMU wrote to flash successfully
*************************************************************************/
boolean IMU_write(void){
  boolean noerror = true;
  // clear incoming buffer
  IMUSerialFlush();
  int sum = 0;
  // Packet header
  IMUSerialWrite('s');
  sum+='s';
  IMUSerialWrite('n');
  sum+='n';
  IMUSerialWrite('p');
  sum+='p';
  // Packet type
  IMUSerialWrite(WRITE_TO_FLASH);
  sum+=WRITE_TO_FLASH;
  // Number of data bytes
  IMUSerialWrite((byte)0);
  sum+=(byte)0;
  // Checksum = Sum of all preceding bytes in packet
  IMUSerialWrite((byte)((sum >> 8) & 0x0FF));
  IMUSerialWrite((byte)((sum) & 0x0FF));

  // Read Command Complete Packet
  byte commandcomp[COMMAND_COMPLETE_SIZE];
  int i = 0;
  // Wait for serial data
  while(IMUSerialAvailable()<COMMAND_COMPLETE_SIZE);
  // Fill array with packet data
  while(IMUSerialAvailable()){
     commandcomp[i] = IMUSerialRead();
     //Serial.println(commandcomp[i],HEX);
     i++;
  }
  // Check Packet Contents
  if(commandcomp[3] != COMMAND_COMPLETE){
    // Wrong packet type received
    //SerPriln("IMU Write to Flash Failed");
    noerror = false;
  }
  return noerror;
}

/*************************************************************************
Function: IMU_silent()
*************************************************************************/
boolean IMU_silent(void){
  boolean noerror = true;
  // clear incoming buffer
  IMUSerialFlush();
  int sum = 0;
  // Packet header
  IMUSerialWrite('s');
  sum+='s';
  IMUSerialWrite('n');
  sum+='n';
  IMUSerialWrite('p');
  sum+='p';
  // Packet type
  IMUSerialWrite(SET_SILENT_MODE);
  sum+=SET_SILENT_MODE;
  // Number of data bytes
  IMUSerialWrite((byte)0);
  sum+=(byte)0;
  // Checksum = Sum of all preceding bytes in packet
  IMUSerialWrite((byte)((sum >> 8) & 0x0FF));
  IMUSerialWrite((byte)((sum) & 0x0FF));

  // Read Command Complete Packet
  int i = 0;
  byte commandcomp[COMMAND_COMPLETE_SIZE];
  // Wait for serial data
  while(IMUSerialAvailable()<COMMAND_COMPLETE_SIZE);
  // Fill array with packet data
  while(IMUSerialAvailable()){
     commandcomp[i] = IMUSerialRead();
     // Serial.println(commandcomp[i],HEX);
     i++;
  }
  // Check Packet Contents
  if(commandcomp[3] != COMMAND_COMPLETE){
    // Wrong packet type received
    //SerPriln("Silent Mode Failed");
    noerror = false;
  }
  return noerror;
```

```
}
```