XINS

A Senior Project

presented to

the Faculty of the Electrical Engineering Department

California Polytechnic State University, San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Science

by

Kyle Howen

March, 2010

TABLE OF CONTENTS

LIST OF TABLES AND FIGURES

Tables ***Page***

Figures

Acknowledgements

My education is an ongoing quest for understanding. I am grateful for all the help provided by my mentors, teachers, and professors. This senior project marks a point of progress in my quest for understanding. And at this point I would like to thank:

-My 8th grade shop teacher Mr. Milo Jenkins.
He was the first to bring out my interest in education, philosophy, and engineering.

-My junior college electronics teacher Mr. Paul Nelson.
His teaching inspired me to pursue a B.S. in Electrical Engineering.

-My junior college engineering teacher Dr. Larry Owens.
His attitude gave me appreciation for the engineering community.

-My friend Mr. John Brewer, and my friend Mr. Jeff Brewer.
They both have gone out of their way to share the fruits of engineering with me.

-My professor Dr. John Oliver.
His approach to undergraduate education inspires me to someday teach.

I hope to give back the community of students and engineers the help I received and more.

Sincerely,

Kyle V. Howen

Abstract

Intelligently automating the control of aircraft is a present challenge for the engineering community. While the automation of motor control in aircraft is similar to groundcraft, the intelligent navigation is significantly more complex. Data such as attitude, altitude, and airspeed are more important and critical to successfully automating an aircraft. The heightened sensitivity to navigation data increases the cost and complexity of aircraft navigation systems.

The XINS seeks to provide a navigation system that is sufficient for remote control or automation of a small UAV. This will be accomplished with a general knowledge in electrical engineering concepts and with equipment available to hobbyists.

This paper covers the theory of design and the first steps to prototyping an aircraft navigation system. Though no testing on actual aircraft was available, simulations and ground tests (in an automobile) were performed. All testing supported the intended function of the XINS.

Given some size and power supply modification, the presented XINS prototype should be able to provide all required data to remotely or autonomously pilot a fixed wing aircraft. The equipment required is commercially available and the design techniques required fit within the capabilities of a Cal Poly EE undergraduate student.

1. Introduction & Background

As all good parents know, a student's summer break is best spent working. And what work is better than that which applies the knowledge learned from school? This is how the XINS was born. During the summer of 2010 a fellow Cal Poly student John Brewer and I set out to autopilot a remote controlled airplane. Immediately, we saw two distinct and challenging aspects of an autopilot: The gathering of flight dynamics, and the motor controlled response to these dynamics. We divided the tasks and set out to conquer. I took the responsibility of designing the navigation system to gather flight dynamics and named it XINS – The Xerous Inertial Navigation System.

The XINS was intended to be installed on a small hobbyist RC fixed wing aircraft. Therefore it gathers data in a fashion that assumes a fixed wing aircraft. Relatively small modifications could be made to suit the XINS providing navigation data for small ground vehicles, blimps, or helicopters. However this report strictly pertains to a fixed wing aircraft implementation and the XINS is tested accordingly

2. Requirements

The XINS should supply a UAV's autopilot with enough information to pilot an aircraft in real-time, and the XINS should comprise of equipment that is available to the public and affordable.

The specific data requirements imposed on this project are collectively referred to as the Flight Model of the XINS. The Flight Model is listed below:

- **Vehicular Attitude**: A collection of three rotations (Roll, Pitch, and Yaw) that fully describe the aircraft's orientation about its center of mass.
- **Forward Airspeed**: The airspeed used to model an aircraft's lift forces.
- **Altitude**: The altitude with respect to MSL (Mean Sea Level).
- **Geographic Location**: Coordinates composed of Latitude and Longitude.

And finally the project requirements for the XINS project are to:

- Design hardware that updates the Flight Model at a rate of 50Hz
- Transmit the Flight Model to a personal computer for analysis.

3. XINS Navigation Theory

Obtaining information about an aircraft's flight with electronic sensors is not always intuitive. This section on theory exists to introduce or refresh one's understanding of the relationship between the data measured onboard an aircraft and the Flight Model developed to navigate. Almost all of the theory presented here was learned or derived from the work of other people. Therefore I will explain where I gathered the theory presented, and why I chose it.

3.1 Coordinate Conventions & Maintaining Aircraft Attitude

I chose to base my navigation system around the coordinate systems presented from the paper "Direction Cosine Matrix IMU: Theory" by William Premerlani and Paul Bizard [1]. This paper provided a straightforward method for utilizing gyro data to update a DCM, and utilizing accelerometer and GPS data to maintain the DCM's validity. I will quickly cover what parts of the paper I used in designing the XINS. I strongly recommend anyone interested to read Premerlani and Bizard's paper. To locate this paper refer to the first listing in the Bibliography section.

3.1.1 Coordinate Systems: Earth & Body Reference Frames

The two coordinate systems used by the XINS for navigating the aircraft are the Earth Fixed Reference Frame and the Aircraft Body Reference Frame. These reference frames have right-handed coordinate systems and are each defined by three orthogonal axes. The axis conventions used here are the same as those presented by Premerlani and Bizard ([1], pg. 8-9).

Earth Fixed Reference Frame:

- **xe** – points North
- **ye** – points East
- **ze** – points down (in the direction of gravity)

Aircraft Body Reference Frame:

- **xb** – points out the aircraft's nose
- **yb** – points out the aircraft's starboard
- **zb** – points out the aircraft's floor

Figure 1 below is taken from Premerlani and Bizard's paper to illustrate these two reference frames.

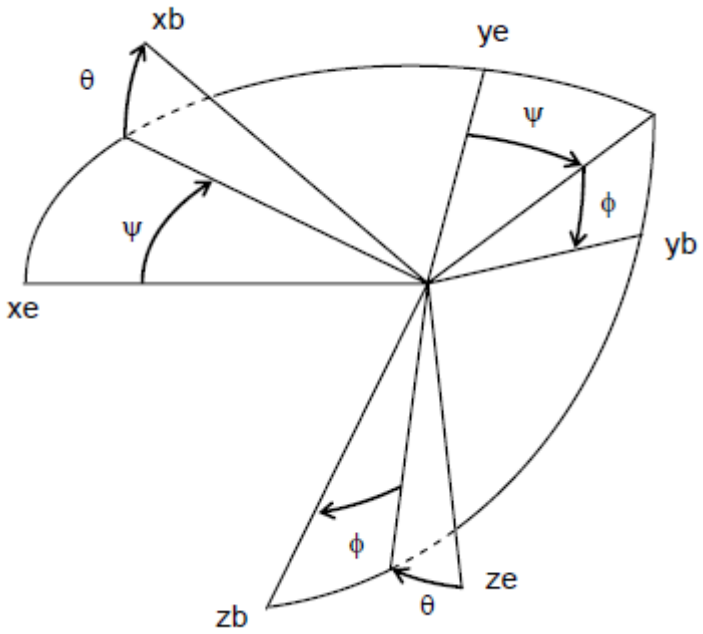


Figure 1: Earth and Body Reference frames denoted by orthogonal axis vectors.
Source: "Direction Cosine Matrix IMU: Theory" (Premerlani & Bizard 1)
Page 8, Figure2

3.1.2 Aircraft Attitude

In order to navigate an aircraft we must know its orientation about the aircraft's center of mass. This orientation is known as aircraft attitude and is comprised of three rotations:

- Roll (ϕ) – A 360° rotation about the aircraft body x-axis.
- Pitch (θ) – A 180° rotation about the aircraft body y-axis.
- Yaw (Ψ) – A 360° rotation about the aircraft body z-axis

These rotations are illustrated in Figure 1 above, and Figure 2 below.

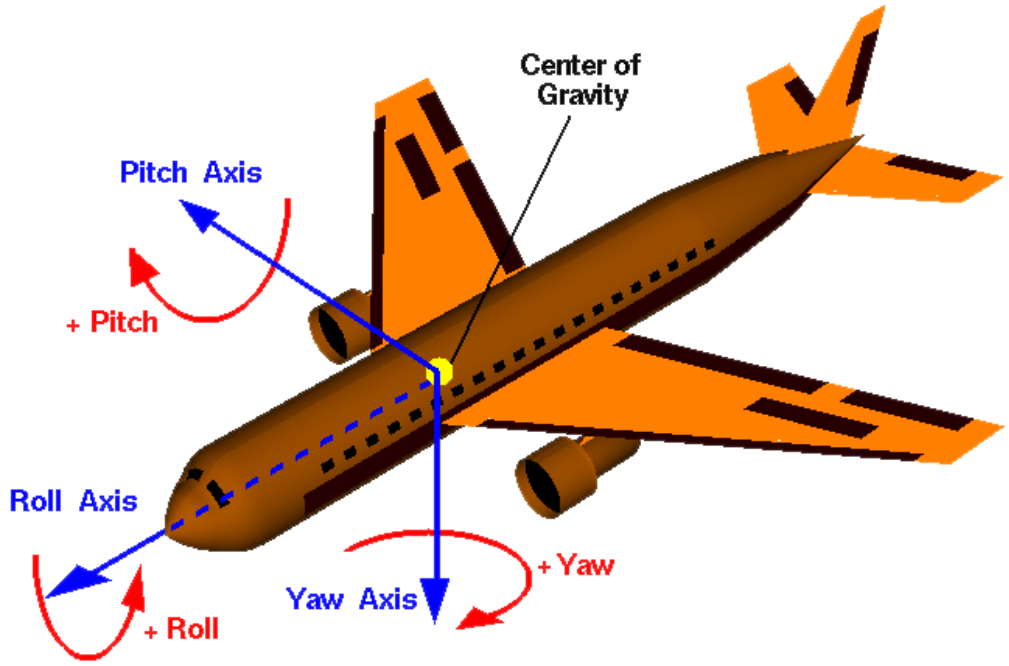


Figure 2: Attitude Rotations – Pitch, Roll, and Yaw.
Source: NASA (http://www.grc.nasa.gov/WWW/K-12/airplane/Images/rotations.gif)

3.1.3 Direction Cosine Matrix:

As Premerlani and Bizard point out, the rotation between the body and earth reference frames can be tracked with a Direction Cosine Matrix ([1], pg. 4). This matrix **R** can be defined by the attitude rotations as follows:

$$[\mathbf{R}] = \begin{bmatrix} \cos\theta\cos\psi & \sin\phi\sin\theta\cos\psi - \cos\phi\sin\psi & \cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi \\ \cos\theta\sin\psi & \sin\phi\sin\theta\sin\psi + \cos\phi\cos\psi & \cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi \\ -\sin\theta & \sin\phi\cos\theta & \cos\phi\cos\theta \end{bmatrix}$$

Equation 1

The DCM's columns are the body frame's axis unit vectors projected on the Earth frame axis vectors. Because of these projections a vector measured in the body frame can be transformed to the Earth frame by a simple matrix multiplication.

$$[\mathbf{V}\mathrm{e}] = [\mathbf{R}][\mathbf{V}\mathrm{b}] \qquad \textit{Equation 2}$$

3.1.4 Updating the DCM with Gyro Data

Using Gyro's to update the DCM turns out to be another simple matrix multiplication. Premerlani and Bizard start with the rate of change of a rotating vector from kinematics:

$$\frac{d\mathbf{r}(t)}{dt} = \boldsymbol{\omega}(t) \times \mathbf{r}(t) \qquad \textit{Equation 3}$$

where $\boldsymbol{\omega}(t)$ is the rotation rate vector obtained from Gyro's in the body reference frame ([1], pg. 13). They approximately integrate this nonlinear differential equation with respect to time by assuming a sufficiently small time interval ([1], pg. 14).

$$d\boldsymbol{\theta}(t) = \boldsymbol{\omega}(t)dt \qquad \textit{Equation 4} \qquad \text{Angular Displacement}$$

$$\mathbf{r}(t) = \mathrm{r}(0) + \int_0^{\mathrm{t}} d\boldsymbol{\theta}(t) \times \mathbf{r}(t) \qquad \textit{Equation 5} \qquad \text{Small vector rotations}$$

$$\mathbf{r_{earth}}(t + dt) = \mathbf{r_{earth}}(t) + \mathbf{r_{earth}}(t) x d\boldsymbol{\theta}(t) \qquad \textit{Equation 6} \qquad \text{Approximate integration}$$

Premerlani and Bizard arrive at Equation 6 by treating the Gyro measurements as negatives in the Earth Reference Frame ([1], pg. 15). The negative sign is combined with the cross product by swapping the order of the operands in the cross product. They take equation 6 and apply it to all three of the body frame's axis vectors in the earth reference frame ([1], pg. 15). The resulting rotation approximation in matrix form is the matrix **U** as seen in Equation 7.

$$[\mathbf{U}] = \begin{bmatrix} 1 & -d\boldsymbol{\theta}_{\mathbf{z}} & d\boldsymbol{\theta}_{\mathbf{y}} \\ d\boldsymbol{\theta}_{\mathbf{z}} & 1 & -d\boldsymbol{\theta}_{\mathbf{x}} \\ -d\boldsymbol{\theta}_{\mathbf{y}} & d\boldsymbol{\theta}_{\mathbf{x}} & 1 \end{bmatrix} \qquad \textit{Equation 7}$$

So finally Premerlani and Bizard's method to updating the DCM from Gyro data is:

$$\mathbf{[R}(t+dt)\mathbf{]} = \mathbf{[R}(t)\mathbf{]}\ \ \mathbf{[U}(t)\mathbf{]} \qquad \textit{Equation 8}$$

([1], pg. 15)

3.1.5 Deriving Attitude from the DCM

Premerlani and Bizard do not mention using Attitude angles to control aircraft. They instead offer methods to controlling aircraft during stable flight by monitoring different scalar values of the DCM. However I find the attitude angles more intuitive and much easier to deal with when confirming the correct operation of the XINS. Therefore my method for deriving roll, pitch, and yaw from the DCM is explained below.

From Equation 1 we can see that R_{31} = -sin(**Pitch).** Therefore

$$\textbf{Pitch} = \sin^{-1}(-R_{31}) \qquad \textit{Equation 9}$$

The arcsine in Equation 9 returns a ±90° result for pitch. A positive angle indicates our aircraft nose is pointed above the horizon, while a negative indicates the aircraft nose is pointed below the horizon.

Yaw can be determined from R_{12} and R_{11}.

$$\frac{R_{12}}{R_{11}} = \frac{\cos(\theta)\sin(\Psi)}{\cos(\theta)\cos(\Psi)} = \tan(\mathbf{Yaw}) \qquad \textit{Equation 10}$$

Therefore Yaw can be computed as shown in Equation 11

$$\mathbf{Yaw} = \tan^{-1}\left(\frac{R_{12}}{R_{11}}\right) \qquad \textit{Equation 11}$$

And finally roll can be found from using R_{33} the pitch result from Equation 9:

$$\mathbf{Roll} = \cos^{-1}\left(\frac{R_{33}}{\cos\left(\sin^{-1}(-R_{31})\right)}\right) \qquad \textit{Equation 12}$$

It should be noted that Equation 11 and 12 are particularly sensitive to pitch angles close to ±90°. At this point cosine(Pitch) → 0 and division by zero may occur. This is somewhat intuitive if we consider an airplane pointing straight up or down. At this attitude changes in roll and yaw are virtually impossible to differentiate.

Yaw and Roll's sensitivity to Pitch is particularly why the vector representation of rotations using a DCM is so important. My first approach to attitude was to use the current roll, pitch, and yaw rotations to anticipate how the gyro data should be integrated. This caused large errors to build up when pitch magnitude approached 90°. The DCM has no such sensitivity to pitch. Therefore by deriving attitude from the DCM we can smudge Pitch magnitudes of 90° with good faith that as the aircraft levels out the derived attitude will still retain its validity.

3.1.6 DCM Maintenance

Remember that Premerlani and Bizard's method to updating the DCM found in Equation 8 is an *approximation*. The unaccounted errors implicit with this operation cause the DCM's unit vectors to stray to magnitudes ≠ 1 and they slowly stray out of orthogonal alignment. Therefore Premerlani and Bizard provide a simple method to ortho-normalize the DCM ([1],

pg. 16,17). I used their recommended method in the XINS. And will repeat it here for reference purposes.

Let **X** be a vector equal to the first row of **R**. And let **Y** equal the second row of **R. X** represents the North vector in the Aircraft Body Reference Frame. **Y** represents the East vector in the Aircraft Body Reference Frame. North and East for our purposes should be orthogonal to each other. Therefore Premerlani and Bizard suggest letting the orthogonal *error* equal to the dot product of **X** and **Y** ([1], pg. 16).

$$error = \mathbf{X} \cdot \mathbf{Y} \qquad \textit{Equation 12}$$

And the orthogonal vectors are calculated as shown below in equation 13 ([1], pg. 17).

$$\begin{aligned} \mathbf{X}_{\mathbf{orthogonal}} &= \mathbf{X} - \frac{error}{2}\mathbf{Y} \\ \mathbf{Y}_{\mathbf{orthogonal}} &= \mathbf{Y} - \frac{error}{2}\mathbf{X} \\ \mathbf{Z}_{\mathbf{orthogonal}} &= \mathbf{X}_{\mathbf{orthogonal}} \times \mathbf{Y}_{\mathbf{orthogonal}} \end{aligned} \qquad \textit{Equation 13}$$

The 3 orthogonal vectors are simply normalized using a form of Taylor expansion in equation 14 and the resulting othro-normal vectors become the three rows of the ortho-normalized DCM ([1], pg. 17).

$$\mathbf{A}_{\mathbf{ortho-normal}} = \frac{\mathbf{1}}{\mathbf{2}}(\mathbf{3} - \mathbf{A}_{\mathbf{orthogonal}} \cdot \mathbf{A}_{\mathbf{orthogonal}})\mathbf{A}_{\mathbf{orthogonal}} \qquad \textit{Equation 14}$$

3.1.7 DCM Error Correction

Due to gyro drift and integration errors we must check our DCM against reference vectors to ensure our model of attitude is accurate. The two reference vectors recommended by Premerlani and Bizard are a gravity vector measured form accelerometers and a ground course angle obtained from a GPS ([1], pg. 17-18). Both of these reference vectors can have slow responses to changes in attitude. This makes them suitable for correcting long term attitude drift but we must still depend on gyros to supply a more immediate response to changes in attitude.

3.1.7a Yaw Correction:

The method I use to derive a yaw correction rotation is similar to Premerlani and Bizard's on page 22 of their paper. I encourage the reader to review their specific method; however I will present mine directly since I find my method to be more straightforward and easier to understand.

First we take the ground course angle provided by the GPS and our current Yaw angle and develop two dimensional unit vectors.

$$\begin{aligned}
&\textbf{GroundCourse} && = (\cos(\angle\text{GroundCourse}),\ \sin(\angle\text{GroundCourse}))\\
&\textbf{Yaw} && = (\cos(\angle\text{Yaw}),\ \sin(\angle\text{Yaw}))\\
&\angle\text{Correction}_{\text{YAW-EARTH}} && = \sin^{-1}(\textbf{Yaw} \times \textbf{GroundCourse})
\end{aligned}$$

$\angle\text{Correction}_{\text{YAW-EARTH}}$ can be treated as a rotation about the **ze**-axis from Figure 1. This way we can view this rotation in the Aircraft Body Reference Frame by multiplying $\angle\text{Correction}_{\text{YAW-EARTH}}$ by the third row of the rotation matrix **R**. This is equivalent to the following vector operation:

$$\mathbf{Correction}_{\mathbf{YAW-BODY}} = [\mathbf{R}]\begin{bmatrix} 0 \\ 0 \\ \angle\text{Correction}_{\text{YAW-EARTH}} \end{bmatrix} \qquad \textit{Equation 15}$$

The primary difference between my yaw correction and Premerlani and Bizard's is that I take the arcsine of the cross product that yields a correction rotation. This is to prevent error angles close to ±90° from appearing stronger than error angles close to 0° by the sine implicit in unit vector cross products. The truth is we want all yaw correction to be provided at a uniformly slow rate. This is due to the nature of GPS's ground course accuracy. As ground speed decreases, the accuracy of the ground course decreases (a plausible scenario of this would be vertical flight). At this point the ground course vector can be severely out of date. The error angle can be mistaken to be very large, and the sine of this unconfirmed error could cause a yaw correction gain calibrated for errors of 1-10° to blow out of proportion.

3.1.7b Roll and Pitch Correction:

The accelerometers yield apparent acceleration in the Aircraft Body Reference Frame. This acceleration is composed of gravitational acceleration, centrifugal accelerations, and accelerations due to the forces of thrust, drag, or wind. On average, accelerations due to thrust and wind are small and short-lived. Premerlani and Bizard recommend taking account for centrifugal accelerations only ([1], pg. 25). The XINS implementation follows this recommendation for the most part, but an approximate derivative of airspeed is available if accounting for thrust/drag acceleration is found necessary for a more pure gravitational measurement.

Premerlani and Bizard suggest modeling centrifugal acceleration as found in Equation 16 below where the velocity vector is simply the velocity along **xb** axis, otherwise known as airspeed ([1], pg. 25).

$$\mathbf{A}_{\mathbf{Centrifugal}} = \boldsymbol{\omega}_{\mathbf{gyro}} \times \mathbf{V} \qquad \textit{Equation 15}$$

Present gyro measurements and current airspeed obtained from the Flight Model make this acceleration easy to account for. We can find an approximate gravity vector shown in Equation 16.

$$\mathbf{A}_{\mathbf{Gravity}} = \mathbf{A}_{\mathbf{Acceleromter}} - \boldsymbol{\omega}_{\mathbf{gyro}} \times \mathbf{V} - \frac{\mathrm{d}\mathbf{Airspeed}}{\mathrm{dt}} \qquad \textit{Equation 15}$$

The gravity vector in the Aircraft Body Reference Frame can be compared with the 3rd row of our DCM. The third row is the Earth's Z-axis unit vector in the Aircraft Body Reference Frame; this is expected to point in the same direction as the gravity vector.

Again my implementation of roll and pitch correction differs from Premerlani and Bizard by normalizing the gravity vector and taking the arcsine of the correction rotation cross product to eliminate the correction rotation's dependence on the magnitude of the error angle. The roll and pitch correction rotations are calculated form Equations 16 and 17.

$$\mathbf{A}_{\mathbf{Gravity-Normal}} = \frac{\mathbf{A}_{\mathbf{Gravity}}}{\sqrt{\mathbf{A}_{\mathbf{Gravity}} \cdot \mathbf{A}_{\mathbf{Gravity}}}} \qquad \textit{Equation 16}$$

$$\mathbf{Correction}_{\mathbf{ROLLPITCH-BODY}} = \sin^{-1}\left(\begin{bmatrix} R_{31} \\ R_{32} \\ R_{33} \end{bmatrix} \times \mathbf{A}_{\mathbf{Gravity-Normal}}\right) \qquad \textit{Equation 17}$$

3.1.7c PI Feedback Controller

Premerlani and Bizard recommended applying a PI controller to the two correction rotations ([1], pg. 26-27). The XINS currently uses their straightforward implementation as shown below.

$$\textbf{Total Correction} = (W_{RP})\mathbf{Correction}_{\mathbf{ROLLPITCH-BODY}} + (W_Y)\mathbf{Correction}_{\mathbf{YAW-BODY}} \qquad \textit{Equation 18}$$

$$\boldsymbol{\omega}_P = (K_P)\textbf{Total Correction} \qquad \textit{Equation 19}$$

$$\boldsymbol{\omega}_I = \boldsymbol{\omega}_I + (K_i)(dt)\textbf{Total Correction} \qquad \textit{Equation 20}$$

$$\boldsymbol{\omega}_{correction} = \boldsymbol{\omega}_P + \boldsymbol{\omega}_I \qquad \textit{Equation 21}$$

And finally this correction is combined with the gyro data before Equations 4, 7, and 8 are applied to update the DCM as suggested by Premerlani and Bizard ([1], pg. 15).

3.2 Geographic Location

Fortunately, the Global Positioning System has taken most of the fun out of modeling geographic location. A GPS can provide latitude and longitude coordinates quickly and accurately enough for navigating a fixed wing aircraft. At this point the XINS does not convert latitude and longitude displacement to meters. However this would be necessary for autopilot path-finding which could become a future design goal.

3.3 Airspeed

Airspeed is the velocity of an aircraft along the aircraft body's **xb** axis. This velocity is used to calculate the lift force, and the lift force can play a large role in sensing stall conditions.

3.3.1 Fluid Velocity

Airspeed can be approximated from measuring fluid velocity in a low wind environment. As an aircraft travels through air, a higher pressure results from the inertia of the still air as it strikes the incident aircraft surfaces. Using a pitot tube attached to a differential pressure sensor we can determine fluid velocity. The equation used by the XINS is shown below. It is also illustrated on nasa.gov (http://www.grc.nasa.gov/WWW/K-12/airplane/pitot.html) [2].

$$\mathbf{V_{Fluid}} = \sqrt{\left|\frac{\mathbf{2(P_{differential})}}{\boldsymbol{\rho}}\right|} \qquad \textit{Equation 22}$$

$$\boldsymbol{\rho} = \textbf{fluid density} = \frac{\boldsymbol{\rho_0}\mathbf{P_{flight}}\mathbf{T_{MSL}}}{\mathbf{P_{MSL}}\mathbf{T_{flight}}} \qquad \textit{Equation 23}$$

Where

$\mathbf{P_{MSL}} = \mathbf{101.32kPa} = \textbf{Pressure at Mean Sea Level}$

$\mathbf{T_{MSL}} = 288.15\ Kelvin$

$\boldsymbol{\rho_0} = 1.225\frac{kg}{m^3}$

$\mathbf{P_{flight}}$ & $\mathbf{T_{flight}}$ **are the pressure and temperature in flight conditions in kPa and Kelvin**

3.3.2 GPS Airspeed

If a GPS signal is available then a ground speed can be obtained. Given a reliable pitch attitude we can approximate Airspeed by Equation 24 seen below.

$$\mathbf{V_{Airspeed}} = \frac{\mathrm{GPS_{GROUNDSPEED}}}{\mathbf{cos}(\theta)} \qquad \textit{Equation 24}$$

The assumption here is that the GPS ground course is in fact the attitude yaw vector. That is to say that there are a no significant crosswinds causing our ground course to diverge from the true attitude yaw rotation.

3.4 Altitude

Aircraft altitude can be used to maintain level and safe flight over a long term flight. Altitude with respect to mean sea level is related to absolute pressure. An approximate relationship between absolute air pressure and altitude above mean sea level can be found in Equation 25. Equation 25 is derived from the Ideal Gas Law, however the atmosphere does not behave as an ideal gas so this equation is only an approximation. A website with the derivation of Barometric Formula from the Ideal Gas Law can be found at (http://hyperphysics.phy-astr.gsu.edu/hbase/Kinetic/barfor.html) [3].

$$\mathbf{P_{Absolute}} = \mathbf{P_{MSL}}\mathbf{e}^{-(\mathbf{Altitude}/\mathbf{h_0})} \qquad \textit{Equation 25}$$

Where

$\mathbf{P_{Absolute}}$ and **Altitude** are the altitude of flight and the pressure at that altitude.
$\mathbf{P_{MSL}} = \mathbf{101.32kPa} = \mathbf{Pressure\ at\ Mean\ Sea\ Level}$
h_0 = scale height = (R T)/(M g_0)
R = 8.31432 J/Kmol K
T = Temperature
M = 0.0289644 kg/mole
g_0 = 9.80665 m/s^2

From Equation 25 we can derive **Altitude** in terms of measured absolute pressure **P** as shown below in Equation 26.

$$\mathbf{Altitude} = (-\mathbf{h_0})\mathbf{ln}(\mathbf{P_{Absolute}}/\mathbf{P_{MSL}}) \qquad \textit{Equation 26}$$

4. Design

The design challenge of this navigation system is to balance simplicity, affordability, and functionality. Although this challenge is similar to any design, the XINS is distinctly limited because it is a senior project. As a senior project the XINS must be affordable for a student and simple enough to complete in several months time. With this in mind let us review the subsystems of the XINS:

4.1 Central Processing Unit

A complicated navigation system requires a good deal of arithmetic. This means somewhere a microprocessor will take data provided by sensors and compute a Flight Model.

The first design choice is where the microprocessor is located. Sensor data could technically be transmitted to a Ground Station via wireless data link and all central processing could take place on a personal computer. This could work well with existing RC airplanes already operated from a remote control. However, the intention of the XINS is to eventually integrate it with an onboard autopilot to enable autonomous flight. Therefore an onboard navigation computer is necessary for fast and independent communication with an onboard autopilot.

Given the large number of sensors and interfacing a prebuilt microcontroller is necessary:

- A custom computer could be designed to perform XINS operations quickly and efficiently. However there was not enough time to purchase a microprocessor, ram, flash memory, SPI and UART modules, etc. and integrate a custom computer.
- A custom processor designed on an FPGA would likely consume a large amount of power and require too much space for a small UAV.
- With the wide variety and availability of small and efficient microcontrollers, using such a device is the only practical option remaining for the XINS.

The next question is which microcontroller the XINS should use. The following microcontroller requirements were compiled from the recursive engineering design method:

- 50kB Flash Program Space
- 3kB data RAM
- C-Language Compiler compatible with the math.h library.
- Floating Point Math, or a Digital Signal Processor capable of Vector and Matrix Math
- 10 MHz minimum clock speed
- 2 UART Modules

- 1 SPI Module
- 3 Internal ADCs

The Flash, RAM, and C-Language Compiler are necessary to bring development time down to several months. All microcontrollers can be programmed in their respective assembly languages, and assembly language programming can yield much higher performance. However the XINS software was expected to exceed the practicalities of assembly language programming. In hindsight, this expectation was incredibly accurate. Assembly programming would have proven a daunting task yielding no additional functionality.

Because a language such as C was chosen a relatively fast microcontroller was required. A processor clocked to over 10 MHz could perform the required arithmetic with floating point data. A microcontroller with DSP functions could possibly perform the same arithmetic with lower clock speeds. Both options were acceptable for the XINS's microcontroller.

The recursive design method mandated 2 UART modules, 1 SPI Module, and 3 internal ADCs for sensors. Most microcontrollers with Floating Point Math/DSP/10MHz Clock speeds have a hand full of built in communication modules and ADCs. Therefore this requirement was not constraining.

These requirements are by no means the minimum hardware requirements to build an aircraft navigation system. However these requirements were the most relevant to the XINS's time constraints.

Some microcontrollers that met the decided requirements are listed below:

-Microchip DSPIC6010 Family of MCUs – Development Board Cost : $250
-Atmel AT91SAM7S64 MCUs – Development Board Cost: $65.00
-Atmel TMS470A256 – Development Board Cost : $60.00

Obviously Atmel provides cheaper solutions than Microchip. However a Microchip dsPIC6010F was available for prototyping at the start of the XINS project, therefore it was selected as the XINS navigation system's microcontroller.

4.2 Geographic Coordinate Sensing

Geographic coordinates are most easily obtained through the latitude and longitude values provided by a GPS. Many GPS solutions are available for less than $100 USD and can provide latitude and longitude coordinates for our microcontroller through UARTs or SPI connections. Several GPS units were considered for use in the XINS:

20 Channel EM-408 SiRF III Receiver with Antenna/MMCX

- Price: $64.95
- Accuracy ~10m
- Input Current: 75mA
- Communication: 1Hz Update, 4800bps Binary or NMEA ASCII

20 Channel EM-406A SiRF III Receiver with Antenna

- Price: $59.95
- Accuracy ~10m
- Input Current: 70mA
- Communication: 1Hz Update, 4800bps Binary or NMEA ASCII

32 Channel LS20031 GPS 5Hz Receiver

- Price: $59.95
- Accuracy ~10m
- Input Current: 41mA
- Communication: 5Hz Update, 57600bps NMEA ASCII

The XINS uses the Locosys LS20031 GPS. This is primarily because of the 5Hz reporting rate and the high-speed data transfer of 57600bps. Recall that a design requirement is that the Flight Model is updated at 50Hz. This means the GPS data will have to be incorporated when available. A the Locosys 5Hz reporting rate helps shorten the gap between Flight Model geographic coordinate updates, and it does so with less power consumption than its competitors.

There are two drawbacks to this choice in GPS. The Locosys does not provide Binary data which can be parsed more efficiently than NMEA sentences. Also the Locosys requires 3.3VDC rails therefore requires regulation down from the dsPIC MCU's 5V supply rail. Regardless of these drawbacks, the unit is optimal for the XINS design.

4.3 Aircraft Attitude Sensing

In order to gather the data required to model aircraft attitude in section 3.1 both a GPS and an IMU is required. The IMU, or inertial measurement unit, supplies the gyro (angular rate sensor) measurements and accelerometer measurements in the three orthogonal aircraft body axes. This component can potentially be the most expensive. MEMS technology has made IMU's affordable however one can expect to spend several hundred dollars obtaining a decent MEMS IMU.

The requirements imposed on an IMU largely depend on the type of aircraft it is installed on. A higher quality IMU can provide more accurate gyro data and depend less on reference vectors for attitude. This may be desired if the aircraft is to perform complicated aerobatics. Also the scale by which acceleration and angular velocity is measured may limit the aerobatic performance of the IMU. For example, sharp turns could cause the apparent accelerations to saturate the accelerometer readings. For modest, level flight most MEMS tri-axis IMUs should be adequate to autopilot, stabilize, or navigate an aircraft.

That being said there were several IMU's available:

- **Atomic's: IMU 6 Degrees of Freedom - $124.95**
  -A stripped down IMU with a Gyro bandwidth and sensitivity of 88Hz,0.977°/s/t
- **Sparkfun's: IMU 6 Degrees of Freedom V4 – $449.95**
  -Sparkfun.com's own IMU with Gyro bandwidth and sensitivity 140Hz, 9.1mV/°/s
- **Analog Device's : ADIS16365 - ~$650.00**
  -A powerful IMU with Gyro bandwidth and sensitivity 350Hz, 0.05°/s/t

Sparkfun's: IMU 6 Degrees of Freedom V4 is probably the most cost effective choice. It has decent sensor resolution as well as the added bonus of magnetometer data that could help in modifying the XINS to suit rotorcraft and blimps. However the ADIS16365 unit was available at the start of the XINS project and has remarkable performance therefore it is the XINS's IMU.

4.4 Airspeed, Altitude, and Temperature Sensing

Determining airspeed through GPS and attitude data from section 3.3.2 can already be accomplished with the sensors discussed thus far.

Fluid velocity and altitude measurements require absolute and differential pressure sensors. There are no definite requirements placed on the pressure sensors. Because of this the popular and affordable Motorola MPXV5004DP and MPXV4115AP sensors were selected. They both cost less than $20 and can provide acceptable data.

Temperature measurement devices can range from thermisters circuits, pre-packaged and calibrated temperature sensors, or simple diode circuits. The diode circuit was used because of the availability, price, and adequate accuracy provided by this solution. The diode circuit is explained in section 5.1.4.

The pressure and temperature sensor outputs can be amplified to improve data resolution per ADC tick at a cost of data range. The dsPIC MCU chosen has 5V analog power rails. Therefore a rail-to-rail operational amplifier would be necessary to utilize as much of the available voltage range as possible. XINS only requires three amplifiers so cost was not an issue. Therefore the reputation and availability of the LMC6484 amplifier IC proved suitable for this project.

5. Implementation

Implementing the XINS theory was a step by step process. At the start of this project I had no previous experience with any microcontrollers or any of the hardware I had planned to interface. Assembly started by writing elementary programs for the dsPIC microcontroller on hand. Then each a sensor was individually evaluated, purchased, and integrated to the XINS to form what is now known as the Navigation Computer.

The Navigation Computer refers to the integrated microcontroller, sensors, and GPS that is installed onboard an aircraft to assemble a Flight Model. As the Navigation Computer developed in complexity a more advanced method of user communication was required. This lead to the development of the Ground Station software.

The Ground Station refers to the set of software that communicates with the XINS to display, track, and record Flight Model data. The Ground Station software is designed for a personal computer running an operating system equivalent to Windows XP. It consists of the XINS Graphical User Interface, any desired terminal communication software, and Google Earth.

I will report on the implementation of the whole XINS in the following order:

- **Navigation Computer - Hardware & Interfacing**
- **Navigation Computer - Software**
- **Ground Station Communication**

5.1 Navigation Computer – Hardware & Interfacing

The XINS Navigation Computer, or Nav Computer, is composed of a microcontroller, GPS, IMU, pressure sensors, and a temperature sensor. The Nav Computer component block diagram can be seen in Figure 3.

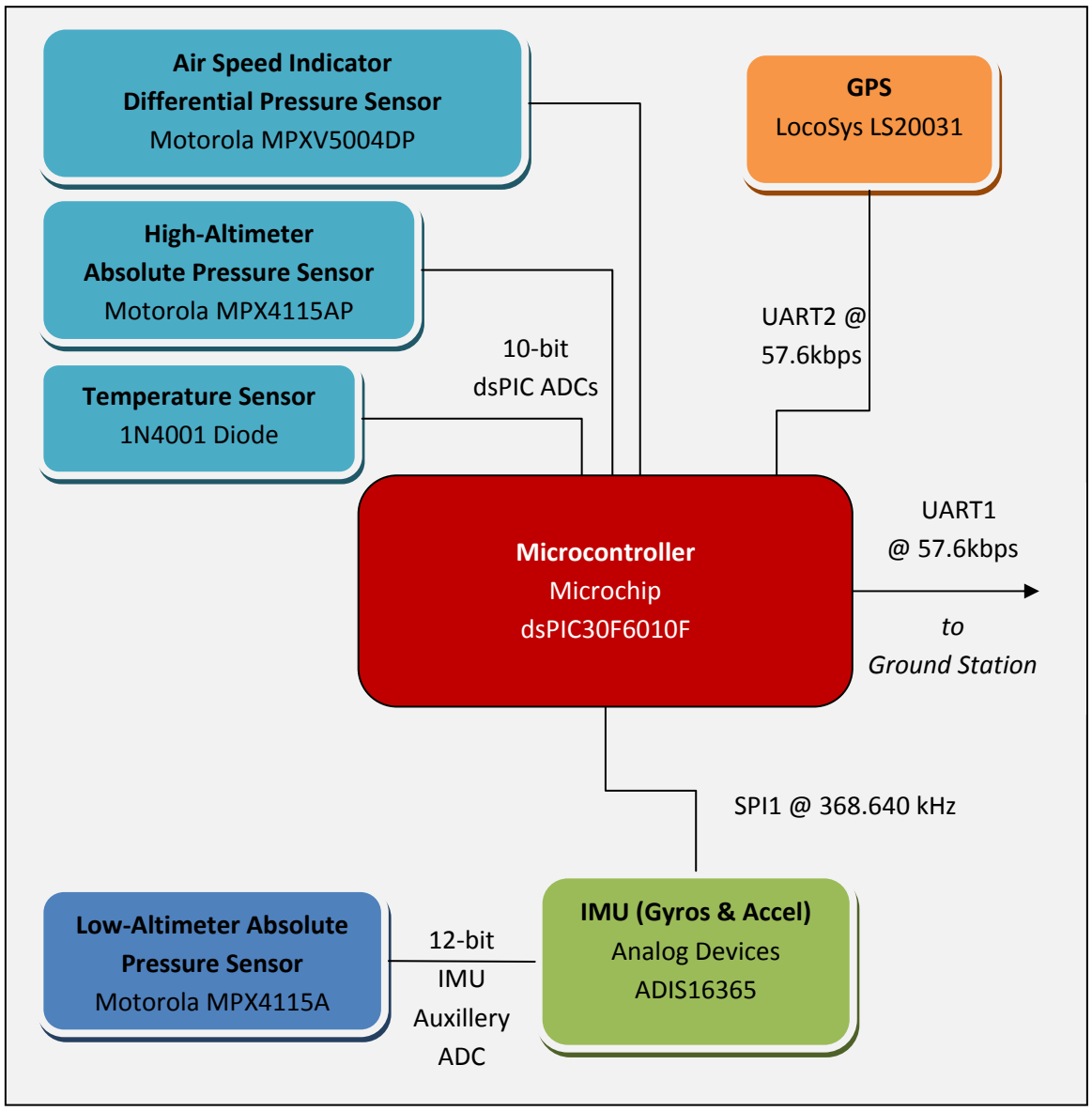


Figure 3: XINS Navigation Computer Block Diagram

A complete circuit diagram for the XINS Navigation Computer can be found in Appendix B pg. 54.

5.1.1 Microcontroller Implementation

The microcontroller chosen for the XINS was the Microchip dsPIC6010F. The Microchip dsPIC6010F used has 144kB of flash program space, 8kB of ram, a well developed C compiler, and a clock speed set to 29.4912MHz. This surplus in speed and memory allows the XINS to perform all of the required calculations in floating point arithmetic. The

dsPIC6010F also has the required two UART modules, one SPI module, and a 4 channel 10-bit high speed ADC. A fixed point DSP library is available however the XINS currently does not use it.

5.1.2 Global Positioning System Implementation

The Global Position System is required to provide time, latitude, longitude, ground speed, and a ground course heading. The LS20031 has the capability of reporting the following NMEA sentences: GGA, GLL, GSA, GSV, RMC, and VTG. The LS20031 is configured to only send the RMC sentence or Recommended Minimum Specific GNSS Data.

Figure 4 is a screen capture taken from the LS20030~3 Datasheet that illustrates the structure and contents of the NMEA RMC Sentence.

- **RMC---Recommended Minimum Specific GNSS Data**

Table 5.1-9 contains the values for the following example:

$GPRMC,053740.000,A,2503.6319,N,12136.0099,E,2.69,79.65,100106,,,A*53

Table 5.1-9 RMC Data Format

Name	Example	Units	Description
Message ID	$GPRMC		RMC protocol header
UTC Time	053740.000		hhmmss.sss
Status	A		A=data valid or V=data not valid
Latitude	2503.6319		ddmm.mmmm
N/S Indicator	N		N=north or S=south
Longitude	12136.0099		dddmm.mmmm
E/W Indicator	E		E=east or W=west
Speed over ground	2.69	knots	True
Course over ground	79.65	degrees	
Date	100106		ddmmyy
Magnetic variation		degrees	
Variation sense			E=east or W=west (Not shown)
Mode	A		A=autonomous, D=DGPS, E=DR
Checksum	*53		
<CR> <LF>			End of message termination

Figure 4: RMC Structure and Data Content

Source: Locosys 20031 Data Sheet, page 7

(http://www.locosystech.com/download/module/LS20030~3_datasheet_v1.1.pdf)

5.1.2b GPS Software Functions

There are two program functions that handle the GPS data:

- **`_U2RXInterrupt()`** *from ISR.c*
  This ISR is invoked upon each character received by UART2. It scans the receive buffer for a '$' indicating the beginning of the RMC NMEA sentence. Upon finding the '$' a SentenceLock is flagged and the NMEA sentence is added to a buffer string until a checksum can be completed successfully. After a checksum is completed the SentenceLock is deflagged and the ISR awaits the next '$'.

- **`GPS_Update()`***from UART.c*
  This function parses an available RMC sentence in the GPS buffer string. It extracts the date, time, ground speed, ground course, and latitude/longitude angles in decimal degrees for use in the Flight Model.

5.1.2c GPS Voltage Regulator

The 3.3V TTL logic is compatible with the dsPIC6010F UART Module. The TX and RX ports of the Locosys are connected to the dsPIC's UART2 RX and TX. However, the LS20031 requires 3.3V power rails. Therefore the ~5VDC supplied by the dsPIC's regulator must be stepped down to 3.3V. A National Semiconductor LM3940 Low Dropout Regulator for 5V to 3.3V conversion was used to accomplish this voltage step. The schematic for the LS20031, LM3940, and dsPIC6010F connections can be found in Figure 5.

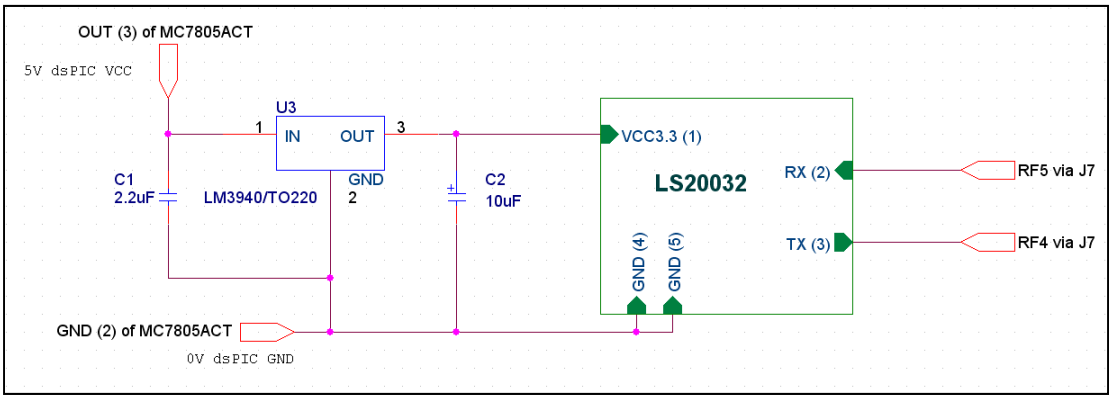


Figure 5: GPS Integration Schematic for LS20031 and dsPIC6010F w/ GPS Power Regulator.

The LS20031 was found experimentally to be a very noisy component that interfered with the XINS's ADC operation. This is why the power for the LM3940 is drawn directly from the dsPIC Development Board's 5V regulator and why a large capacitor is necessary at the LS20031 3.3V output.

5.1.3 Inertial Measurement Unit

The ADIS16365 IMU used has the following specifications:

- ±300°/s dynamic Gyro range
- 0.05°/s/LSB Angular Velocity Resolution
- ±17*g* dynamic Accelerometer range
- 0.33m*g*/LSB Acceleration Resolution
- SPI Communication

The ADIS16365 is connected to the dsPIC through the dsPIC's SPI1 module clocked at 368.640kHz. This connection is found in more detail on the schematic from Figure 6.

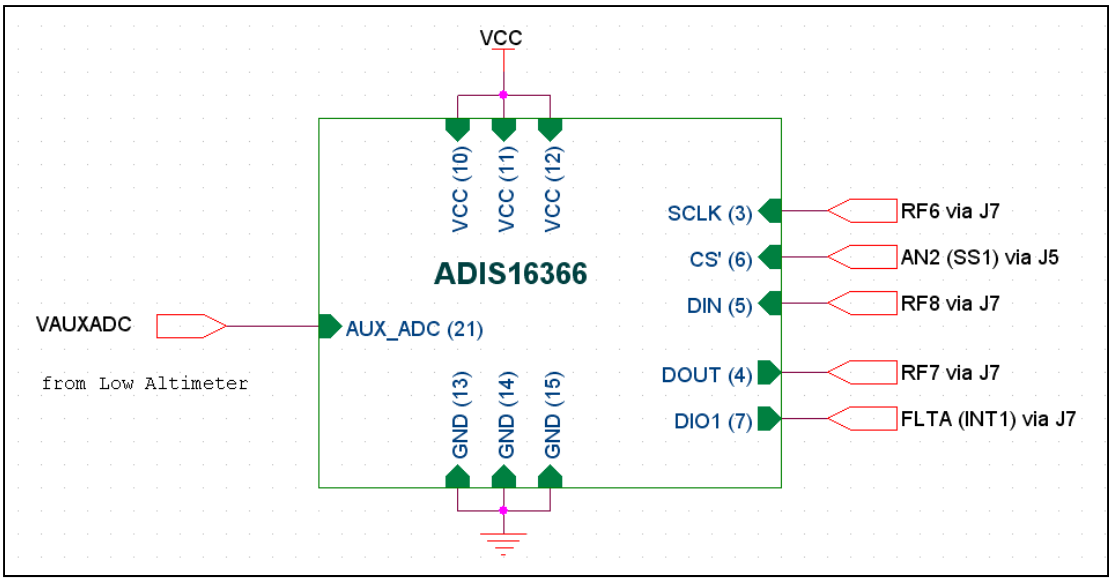


Figure 6: IMU Integration Schematic for ADIS16365 and dsPIC6010F

In order to physically connect the IMU to the dsPIC a 2-row 36 pin female header with 1mm pitch had to be obtained through Samtec. The female header was then soldered onto a SchmartBoard with 0.5mm pitch since no 1mm pitch was available. Wires were soldered to the SchmartBoard and finally a connection to the dsPIC could be established.

The dsPIC has an IMU structure which houses the variables used to communicate with and store data from the ADIS16365. Three major functions that are used to communicate with the IMU:

- **`SPI1_TXRX()`** *from IMU.c*
  This function is an uninterruptable transmit and receive function. It steers the SPI1 module to transmit IMU.TxData while simultaneously receiving IMU.RxData.

- **`IMU_Init()`** *from IMU.c*
  This function uses SPI1_TXRX() on power-up to set the ADIS16365's angular rate sensitivity to a maximum of 300°/s, and to set the internal average filter taps to 16.

- **`IMU_Update()`** *from IMU.c*
  This function uses SPI1_TXRX() to sequentially request and obtain all relevant IMU data and then stores the data in the IMU structure's integer variables. The data received in order is : SupplyOut, XGyro, YGyro, ZGyro, XAccl, YAccl, ZAccl, XTemp, YTemp, ZTemp, AuxADC.

5.1.4 Temperature Sensor

Equations 22 and 25 both require flight temperature to be measured. The results of these equations are not particularly sensitive to temperature therefore the sensor can be simple. A 1N4001 diode was used as a temperature sensor. Figure 7 shows the circuit diagram for the sensor's setup.

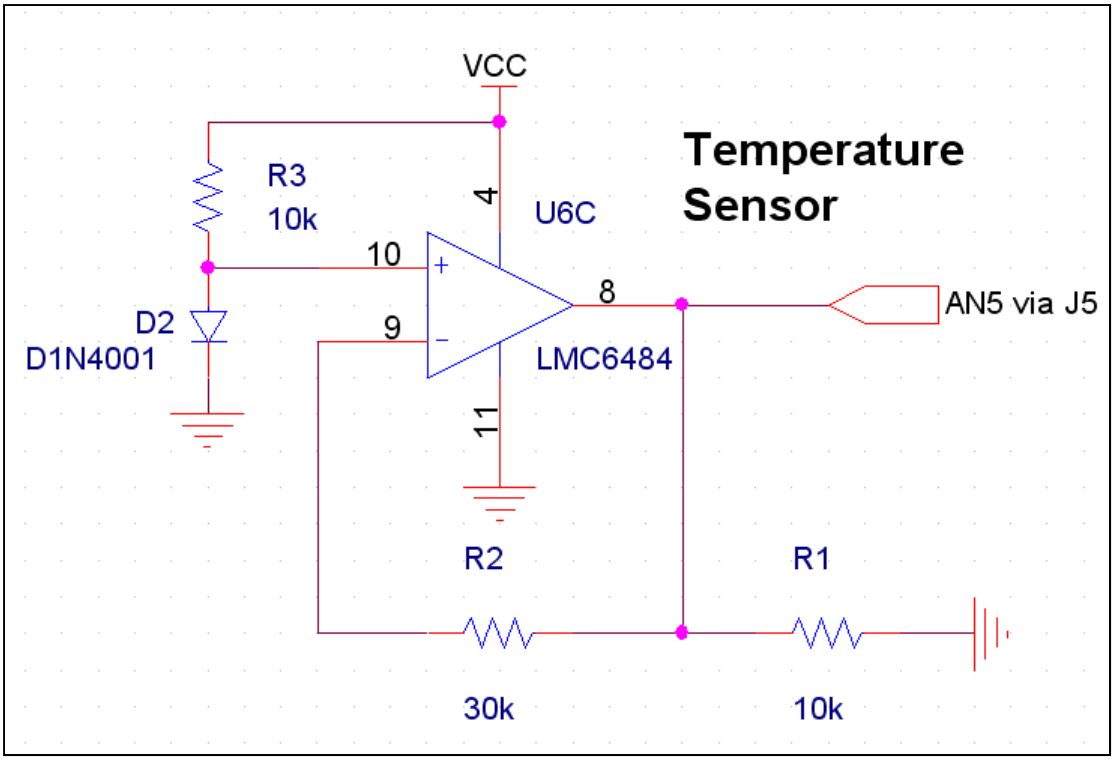


Figure 7: Temperature Sensor Integration Schematic for 1N4001, LMC6484, dsPIC6010F

The voltage across the diode was measured at 295 Kelvin. Equation 27 shows the model used for the diode voltage.

$$\mathbf{V_D} = \mathbf{V_{D@295K}} - (2.3\text{mV})(\mathbf{T} - 295\text{K}) \qquad \textit{Equation 27}$$

Equation 28 shows the temperature in terms of ticks of the internal dsPIC 10-bit ADC.

$$\mathbf{T = 295K} - \left(\frac{\mathbf{V_{SUPPLY}} \cdot (\mathbf{ADC_{TEMP}} - \mathbf{ADC_{TEMP@295K}})}{\mathbf{1024} \cdot \mathbf{2.3}\frac{\text{mV}}{^\circ\text{C}} \cdot \mathbf{A_{Temp}}} \right) \qquad \textit{Equation 28}$$

where we measured $\mathbf{ADC_{TEMP@295K}} = \mathbf{441}$

$\mathbf{V_{SUPPLY}}$ is measured by the ADIS16365

and from Figure 7 we see $\mathbf{A_{Temp}} \cong \left(\mathbf{1} + \frac{\mathbf{30k\Omega}}{\mathbf{10k\Omega}}\right) = \mathbf{4}$ *Equation 29*

This temperature calculation happens in the NAV_`Sensors(double)` function based on the IMU.Temp variable that stores the $\mathbf{ADC_{TEMP}}$ value from the 10-bit ADC and the NAV.IMUVCC which stores the $\mathbf{V_{SUPPLY}}$ in units of mV.

Figure 8 shows the transfer characteristic for the temperature sensor's op amp circuit.

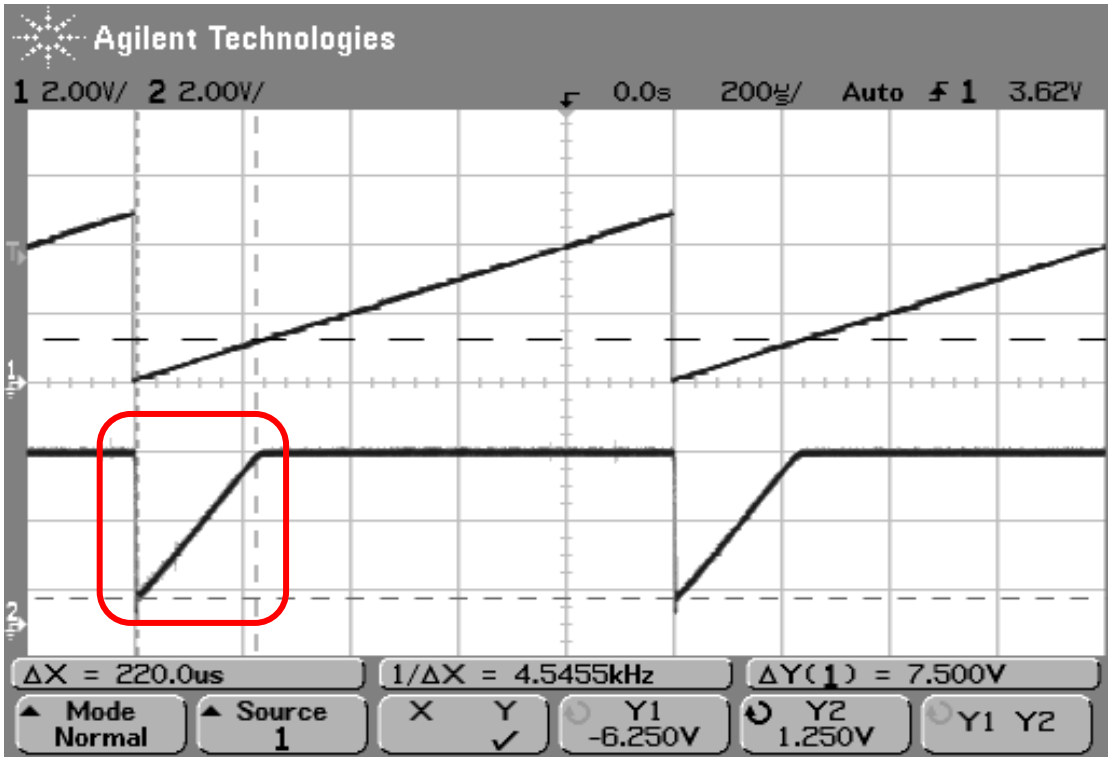


Figure 8: Temperature Sensor Amplifier Transfer Characteristic *(Bottom Trace). Vin (top trace) is a 1kHz triangle wave that spans from 0 to 5V.*

The slope of the highlighted section of the VTC is consistently very close to 4. Vout saturates when Vin is greater than 1.250V. A range of diode voltages from 0.3V – 1.2V is more than enough data to represent all safe operating temperatures for the XINS.

5.1.4a Temperature Sensor Resolution

Since the $\mathbf{ADC_{TEMP}}$ is an integer value we can obtain the sensor resolution by taking the partial derivative of Equation 28 with respect to ADC_{TEMP}.

$$\left|\frac{\partial \mathbf{T}}{\partial \mathbf{ADC_{TEMP}}}\right| = \frac{\mathbf{V_{SUPPLY}}}{\mathbf{1024} \cdot \mathbf{2.3}\,\frac{\mathrm{mV}}{^{\circ}\mathrm{C}} \cdot \mathbf{4}} \qquad \textit{Equation 30}$$

We can assume $\mathbf{V}_{supply}$ will be regulated at a level close to 4.9V. Therefore our temperature sensor resolution should be about 0.52°C/LSB.

5.1.4b Temperature Sensor Testing

A convenient wide range of temperatures was not available at the time to test the temperature sensor with. However Table 1 below shows promise for the sensor's accuracy over a range of room temperatures.

XINS Temp		External Thermometer		Error
(K)	(F)	(K)	(F)	(%)
296.65	74.3	296.7	74.5	0.04
297.92	76.586	297.9	76.5	0.02
294.73	70.844	294.8	70.9	0.01

Table 1: Temperature Sensor Test

Extensive testing and minor tuning may be needed in the future if the XINS is taken to a extreme temperature environment unlike California's central coast.

5.1.5 Airspeed Indicator - Differential Pressure Sensor

The differential pressure sensor is used to measure fluid velocity as the aircraft moves through air. The requirements placed on the differential pressure sensor are variable. A higher maximum differential pressure range increases fluid velocity range. In general a higher pressure range will reduce pressure resolution.

The recommended differential pressure sensor requirements would be a 5V ratiometric differential pressure sensor with a maximum differential pressure of around 3.5kPa. This allows for a fluid velocity of up to 150 knots to be measured. This should be enough range and resolution for the average hobbyist remote controlled flight.

The XINS uses a Motorola MPXV5004DP pressure sensor for its ASI. The MPXV5004DP delivers differential pressures from 0 to 3.92 kPa as a voltage that is ratiometric with respect to its supply voltage. The schematic for the MPXV5004DP can be found in Figure 9.

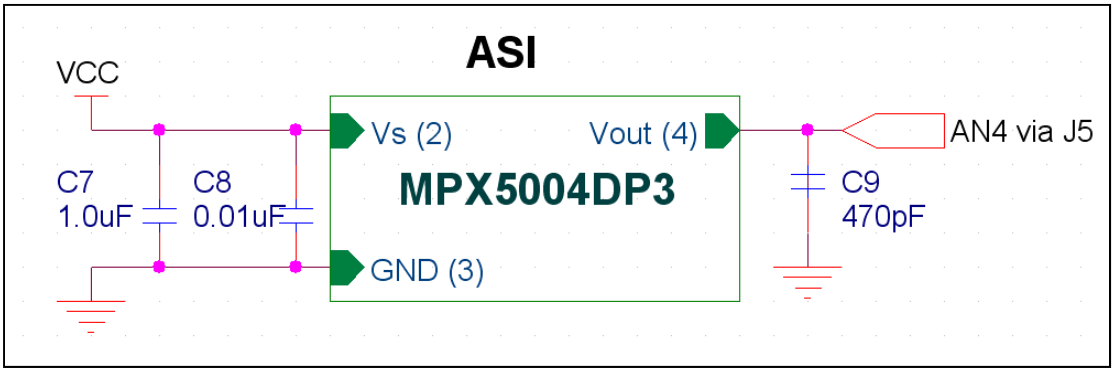


Figure 9: Differential Pressure Sensor Integration Schematic for MPXV5004DP and dsPIC6010F

The transfer function based off that given in page 3 of the MPXV5004DP datasheet is the following:

$$\mathbf{Vout} \cong \mathbf{Vs}((0.2 \cdot \mathbf{P}_{\mathrm{Differential}}) + 0.2) \qquad \textit{Equation 31}$$

Our 10-bit ADC measures the ratio of $\frac{\mathbf{Vout}}{\mathrm{Vs}}$ as $\mathrm{ADC}_{\mathrm{ASI}}$. Therefore the XINS model for the transfer function is the following:

$$\frac{\mathbf{ADC_{ASI}}}{\mathbf{1024}} \cong (0.2 \cdot \mathbf{P}_{\mathrm{Differential}}) + \mathrm{P}_{\mathrm{Offset}}) \qquad \textit{Equation 32}$$

P_{Offset} is calculated at start-up when the differential pressure is assumed to be approximately 0. This offset pressure is always very close to 0.2***Vs**. A value for $\mathbf{P}_{\mathrm{Differential}}$ is calculated in the NAV_`Sensors(double)` function based on Equation 32.

This value for differential pressure is then applied to Equation 22 to yield present fluid velocity.

5.1.5a Airspeed Indicator Testing

The XINS Nav Computer is currently equipped with a suitable pitot Tube for probing the differential pressure required in Equation 22. However the pitot Tube is too small to be mounted on an automobile, therefore there is no testing available for the ASI until the XINS and pitot Tube are mounted on a small UAV.

5.1.6 Altimeters - Absolute Pressure Sensors

The XINS is equipped with two altimeters, the High Altimeter and the Low Altimeter. They are each based upon the MPX4115A absolute pressure sensor measurements. This pressure sensor has a range from 15 to 115kPa which makes it suitable for approximating altitudes from several hundred meters below sea level to more than 4 kilometers above sea level. The circuit diagram for the two altimeters can be found in Figure 10.

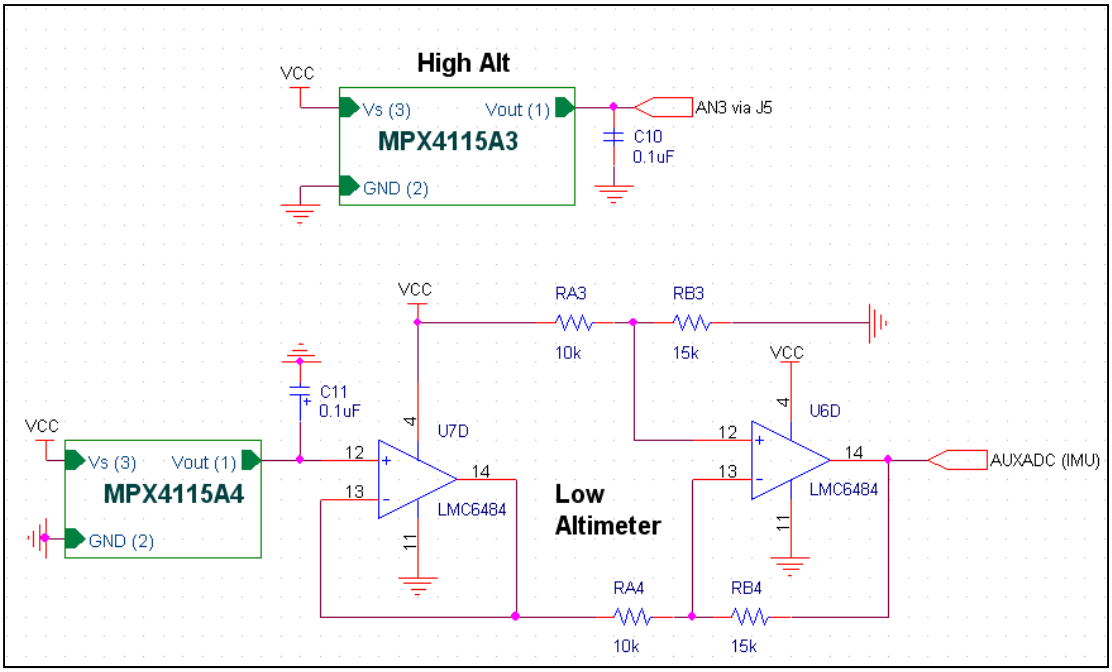


Figure 10: Absolute Pressure Sensor Integration Schematic for Two MPX4115A's, One LMC6484 Quad Rail-to-Rail Op-Amp, and the dsPIC6010F

5.1.6a High Altimeter

The High Altimeter is setup much like the Differential Pressure Sensor. Its output is ratiometric, unamplified, and tied directly to the dsPIC's 10-bit ADC. The $\mathbf{ADC_{P\text{-}HIGHALT}}$ is a measure of $\frac{\mathbf{V_{P-HIGHALT}}}{\mathbf{V_S}}$. The MPX4115A transfer function modeled from the transfer function provided on page 6 of the MPX4115A data sheet is found in Equation 33.

$$\frac{\mathbf{V_{P-HIGHALT}}}{\mathbf{V_S}} \cong \left(0.009 \cdot \mathbf{P}_{\text{abs}} - 0.095 \pm \mathbf{P}_{\text{error}}\right) \qquad \textit{Equation 33}$$

Where $\mathbf{P}_{\text{error}} = \pm 1.5\text{kPa} \cdot \text{Tempfactor}$
and TempFactor = 1 for all intended flight temperatures for the XINS.

The function NAV_`Sensors(double)` uses Equation 33 to solve for $\mathbf{P}_{\text{abs}}$ and applies this pressure to Equation 26 to approximate altitude with respect to mean sea level. This process is equivalent to Equation 34.

$$\mathbf{Altitude} = \mathbf{h_0} \ln\left(\frac{\frac{\mathbf{ADC_{P-HIGHALT}}}{1024} + 0.095}{\mathbf{P_{MSL}} \cdot 0.009}\right) \qquad \textit{Equation 34}$$

5.1.6b High Altimeter Resolution

To find the resolution of the High Altimeter we take the partial derivative of Equation 26 with respect to $\mathbf{ADC_{P\text{-}HIGHALT}}$.

$$\frac{\partial \mathbf{Altitude}}{\partial \mathbf{P_{Absolute}}} = \frac{-\mathbf{h_0}}{(\mathbf{P_{MSL}})} \cdot e^{\left(\frac{\mathbf{Altitude}}{\mathbf{h_0}}\right)}$$

$$\frac{\partial \mathbf{P_{Absolute}}}{\partial \mathbf{ADC_{P-HIGHALT}}} \cong \frac{1}{1024 * 0.009}$$

Assuming h_0, P_{MSL}, and V_s are constant and are 8635.02m, 101.325kPa, and 4.9V respectively we can then simplify the partial derivative and find:

$$\mathbf{High\ Altimeter\ Resolution} = \left|\frac{\mathrm{d}\mathbf{A}}{\mathrm{d}\mathbf{ADC}}\right| = \frac{\mathrm{d}\mathbf{A}}{\mathrm{d}\mathbf{P}}\frac{\mathrm{d}\mathbf{P}}{\mathrm{d}\mathbf{ADC}} = 9.24707 e^{\left(\frac{\mathbf{Altitude}}{\mathbf{8635.02m}}\right)} \frac{m}{LSB}$$

Equation 35

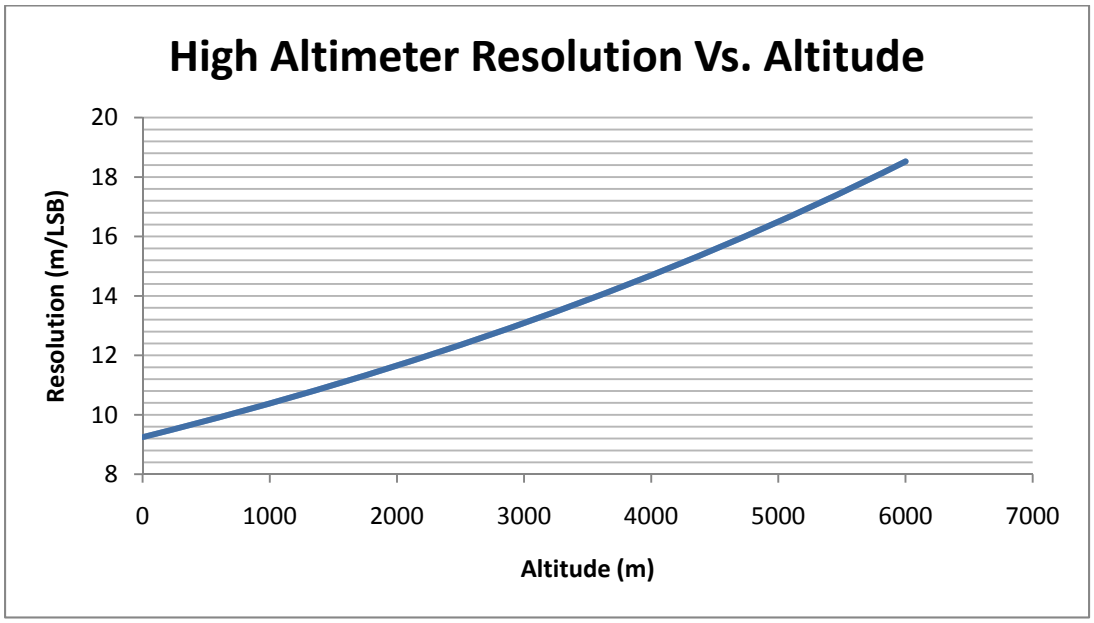


Figure 11: High Altimeter Resolution vs. Altitude

Figure 11 illustrates the change in altimeter resolution vs. altitude. The resolution stays between 10 and 20 meters per LSB for all intended altitudes. A key altitude to note is 2.6km. This is near where the more precise Low Altimeter begins to saturate and the High Altimeter takes over. This gives the XINS a best case resolution of 12.5m/LSB at any altitude above 2.6km.

5.1.6c Low Altimeter

Notice from the circuit diagram in Figure 10 that the Low Altimeter's MPX4115A is attached to a differential op-amp circuit. The idea of using a differential amplifier circuit with LMC6484 rail to rail quad op-amp was suggested by Mr. Jeff Brewer [4]. The ideal equation for the amplifier circuit is found in Equation 36.

$$\mathbf{V}_{\text{Difference}} \cong \frac{\text{R2}}{\text{R1}}(\text{V}_\text{s} - \text{V}_\text{L}) \qquad \textit{Equation 36}$$

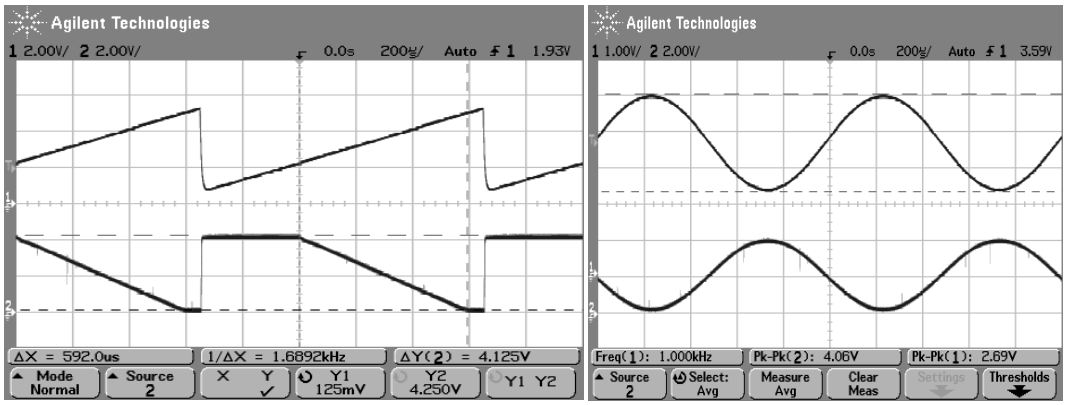


Figure 12: The Low Altimeter Differential Amplifier

***Left**: Voltage transfer characteristic (bottom trace) from a triangle wave input (top trace) that illustrates the saturation region of the amplifier. **Right:** Voltage input waveform (top trace) and voltage output waveform (bottom trace) that supports a differential gain of -1.50.*

Figure 12 supports the assumption that the differential amplifier circuit is behaving according to Equation 36.

5.1.6d Low Altimeter Transfer Function

If we apply the transfer function found in Equation 33 to V_L in Equation 36 we can obtain a transfer function for the entire Low Altimeter analog circuitry.

$$\mathbf{V}_{\text{AUXADC}} = \frac{\text{R2}}{\text{R1}}(\text{V}_\text{s})(1.095 - 0.009[\text{P}_{\text{abs-low}} \pm \text{P}_{\text{Error}}]) \qquad \textit{Equation 37}$$

The Analog Devices IMU has 12-bit ADCs that have non-ratiometric output values. This 2-bit increase in precision helps improve the Low Altimeter's pressure resolution. However the ADC max voltage allowable is 3.3V. This maximum limitation is partly why the differential op-amp circuit was necessary. The voltage from a MPX4115A at mean sea level given a 5V supply is approximately 4.09V. This means the IMU's AUXADC would already be saturated at sea level. Since the XINS was constructed along the California coast the high precision altimeter was desired to read low altitudes.

The amplifier gain R2/R1 was chosen to be 1.5 so that the node voltages of the amplifier circuit would all stay some distance from the 0V and 5V rails. This ensured that the LMC6484 was not distorting the ideal model of the differential amplifier setup. Any amplifier gain greater than 1 will improve the Low Altimeter's resolution since the AUXADC has a fixed resolution of 0.81mV/LSB and the SupplyOut ADC has a fixed resolution of 2.41mV/LSB. This improvement in resolution comes as a loss in pressure/altitude range.

A LM4040C41 4.09V reference voltage was tested in the differential amplifier circuit to increase the altitude range by reducing the minimum output voltage of the amplifier circuit, thereby utilizing more of the AUXADC's lower range. There were two problems with using this 4.09V reference. The first was that the noise from the 4.09V reference and the supply voltage were not correlated, therefore adding to the total output noise of the amplifier. The second was that the 4.09V reference spoiled the ratiometric properties of the sensor's output. Since V_L is ratiometric with respect to Vs, the Vs term in Equation 36 is conveniently factored out from the difference term. And since Vs is already measured by the IMU the ratio V_{AUXADC}/V_S can be easily calculated providing a robust calculation of an amplified pressure signal.

5.1.6e Low Altimeter Resolution

The function NAV_`Sensors(double)` uses Equation 37 to solve for $\mathbf{P}_{\mathrm{abs-low}}$ and applies this pressure to Equation 26 to approximate altitude with respect to mean sea level. This process is equivalent to Equation 34. Giving a similar analysis to the Low Altimeter as shown in the High Altimeter Resolution section we find that the Low Altimeter's resolution is improved:

Assuming h_0, P_{MSL}, and V_s are constant and are 8635.02m, 101.325kPa, and 4.9V respectively we can then simplify the Low Altitude Resolution to

Low Altimeter Resolution

$$= \frac{\mathbf{h_0 \cdot 0.81}\frac{\mathrm{mV}}{\mathrm{LSB}}}{\mathbf{P_{MSL} \cdot V_S \cdot A_{Diff}}} \cdot \boldsymbol{e}^{\left(\frac{\mathbf{Altitude}}{\mathbf{h_0}}\right)}\left(\frac{m}{LSB}\right) \qquad \textit{Equation 38}$$

$$\approx 1.0435\boldsymbol{e}^{\left(\frac{\mathbf{Altitude}}{\mathbf{8635.02m}}\right)}\left(\frac{m}{LSB}\right)$$

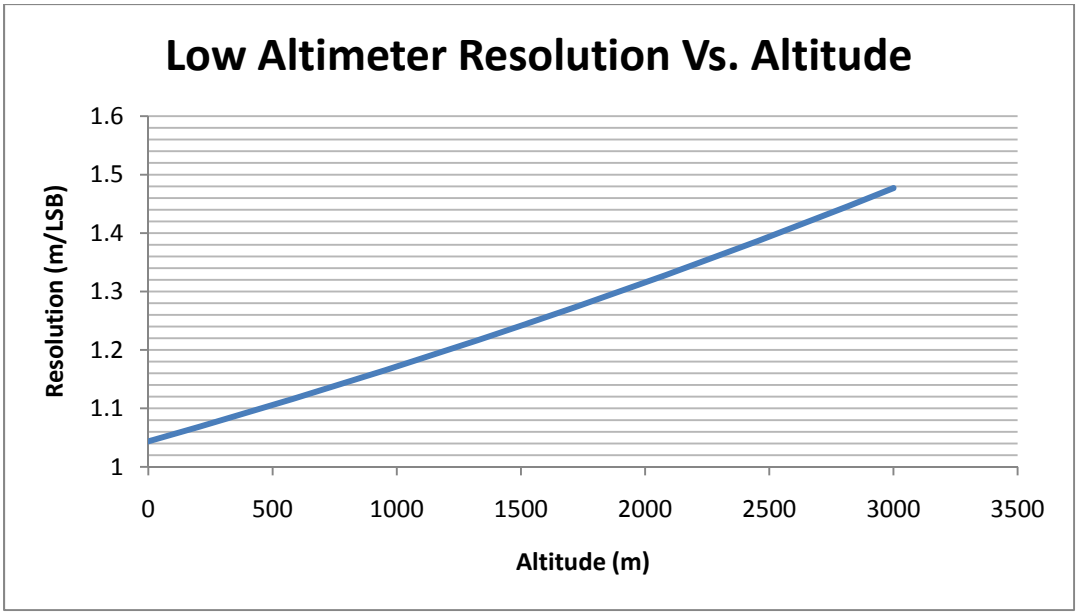


Figure 13: Low Altimeter Resolution vs. Altitude

From Figure 13 it is obvious that this implementation yields a much improved altitude resolution at the cost of range.

5.1.6f Altimeter Testing

The individual altimeters were first tested by maintaining constant altitude and tracking the local barometric pressure recorded in Grover Beach, 93433 by the following web site: http://www.idcide.com/weather/ca/grover-beach.htm

	Low Alt	High Alt	Low P	High P	Barometric		SiteUpdate
02-17-10	**(m)**	**(m)**	**(kPa)**	**(kPa)**	**(inHg)**	**(kPa)**	
1:20PM	32.9473	33.1	100.905	100.901	29.94	101.3885	12:59PM
3:45PM	34.7332	35.1	100.882	100.888	29.92	101.3208	2:56PM
11:28PM	48.3447	34.23	100.713	100.928	29.95	101.4224	10:56PM

Table 2: Altimeter Testing – Barometric Pressure

Table 2 simply shows that the pressure readings do not drift far from the Barometric pressure at the time. More elevation testing can be found in the System Testing section later in this report.

5.2 Navigation Computer – Software

All of the source code was written by the author of this report, Kyle Howen. However, I would like to give credit to mikroElektronika's online book *Programming dsPIC (Digital Signa Controllers) in C* by Zoran Milivojević and Djordje Šaponjić for jumpstarting my education concerning dsPIC programming in C [5]. The Microchip *dsPIC30F Family Reference Manual* was used extensively in designing the Navigation Computer Software [6]. The *MPLAB® C30 C Compiler User's Guide* was also used to reconcile my knowledge in C to C30 Compiler specifications [7].

This section covers the Navigation Computer software functions that are not hardware specific. These functions fall into two categories: navigation functions, and main program functions.

5.2.1 Navigation Functions

The navigation functions in general operate on the NAV structure variables to create and maintain a Flight Model. The functions mentioned here are located in the NAV.c source file.

`NAV_Init()` – This resets the DCM, Attitude, and PI Controller gains and calls NAV_InitialValues().

`NAV_InitialValues()` – This function takes 100 ADC samples and IMU data transmissions and averages them to formulate values for present air density (Equation 23), initial absolute pressure (Equation 25), and take off altitude (Equation 26).

`NAV_Trig()` – This function simply calculates the sine and cosine of the present attitude angles.

`NAV_Sensors(double)` – This function is where most of the sensor data is processed into Flight Model data.

- The pressure sensor samples and supply voltage samples are passed through a 16 tap digital averaging filter. This filter is managed by a custom 16-integer deep circular buffer.
- The present temperature is calculated. (Equation 28)
- High Altimeter pressure and altitude are calculated. (Equations 33,34)
- The altitude is examined to determine if the Low Altimeter can be used.
- If so, the Low Altimeter pressure and altitude is calculated and these values override the High Altimeter data. (Equations 26,37)

- Fluid velocity is determined from the ASI's differential pressure data. (Equations 22,32)
- If a valid GPS Ground Speed is available, the ground speed is used to calculate Airspeed. (Equation 24) Otherwise the ASI's fluid velocity is assumed to be the best guess for Flight Model airspeed.
- Angular velocities and apparent acceleration are given appropriate unit conversion and axis conventions.

`NAV_Editgains(double,double,double,double)` – This function simply displaces the PI Controller gains by the function parameters.

5.2.1a Direction Cosine Matrix Code

See the chapter "XINS Navigation Theory" to describe the operation of these functions.

`NAV_DCM()` – This function takes the current attitude angles and calculates a DCM. (Equation 1)

`NAV_DCM_Decode()` – This function formulates attitude angles from the present DCM. See section "Deriving Attitude from the DCM"

`NAV_DCM_Copy(int)` – This function simply makes a carbon copy of the DCM for use in DCM calculations.

`NAV_DCM_WCorrection(double)` – This function is the digital implementation of the PI Controller that computes an appropriate angular correction. See section "DCM Error Corrections"

`NAV_DCM_Update(double)` – This function updates the DCM with present gyro data and results from the NAV_DCM_WCorrection(double) function. See section "Updating the DCM with Gyro Data"

`NAV_DCM_OrthoNorm()` – This function ortho-normalizes the present DCM. See section DCM Maintenance.

5.2.1b Custom Circular Buffer

`NAV_IncrementIndex()` – This function increments the present index of the custom circular buffer, wrapping properly from beginning to end or vice versa.

`NAV_DisplaceIndex(int)` – This function returns the circular buffer index displaced by the function argument.

`NAV_MakeOIA()` – This function makes an array of integers whose contents are the indexes of the circular buffer ordered from oldest to newest. OIA stands for ordered index array.

5.2.2 Main Program Functions

The main program functions are located in XINS.c and ISR.c. Only three are worth mentioning here: HardwareInit(), _T2Interrupt(), and main().

`HardwareInit()` – This function initializes the dsPIC hardware and peripherals. The sequence follows:

- Set interrupt priorities, GPS UART2 is top priority, Timer 2 is second priority, as of this point no other interrupts are critical.
- Enable all used interrupt vectors except GPS receive interrupts.
- Configure ADC Registers
- Configure UART1 Module registers
- Configure UART2 Module registers
- Configure SPI1 Module registers
- Configure TMR1,2, & 4 registers
- Run UART, IMU, and GPS initialization functions
- Wait 1 second
- Run Navigation initialization functions
- Enable the Timer2 State timer
- Enable GPS receive Interrupts

_T2Interrupt() – This function is the Interupt Service Routine for Timer 2. Timer 2 is configured to have a period of about 20ms.

If a GPS transmission is not current in progress:

This ISR samples the IMU data and the ADC data, then sets flags that will engage all of the Navigation code.

If a GPS transmission interrupts the _T2Interrupt ISR:

The Timer 2 interrupt has been designed so that it can safely be interrupted by a GPS transmission. The Timer 2 interrupt will be completed as the GPS transmission code pauses for UART data flight time.

This allows the Flight Model to be updated at a rate of 50Hz. This function also checks if the GPS has stopped responding and sets a flag to notify other functions that the GPS signal is lost.

NAV_Main() – This function runs the HardwareInit() function then enters an infinite while loop. This while loop interacts with the ISRs by reading and resetting flags.

When the Timer 2 ISR finishes acquiring new data, the main function will respond by executing all of the sensor/DCM navigation functions.

When a GPS transmission is complete and new GPS data flag is set the main function will respond by executing GPS_Update() to parse the received NMEA sentence. After the parsing is complete the navigation functions will incorporate the new GPS data into the Flight Model.

When a GPS transmission has been detected the RS232 transmission will be initialized. This allows data to be sent to the Ground Station software at a rate of 5Hz. If the GPS has stopped responding the RS232 transmissions will be initialized every 10 states.

Upon each cycle of the main while loop data received from the Ground Station through the RS232 connection is processed. If any valid console commands have been received they are executed in RS232_Update()

5.2.3 Function Execution Benchmarks

The XINS code is obviously time critical. Therefore good measure went into benchmarking the mainloop code to ensure that all possible interrupt scenarios would be manageable within the 20ms state time. Table 3 below contains the benchmarks recorded for time critical subroutines.

	(s)	(us)	
Timer4 -> Seconds	8.68E-06	8.680556	
The times measured here are generally the uninterrupted processing times			
2/4/2010			
Navigation Code:	**Ticks**	**Time (us)**	
State Time Calcs	1	8.680556	
All Sensor Calcs	97	842.0139	
Wcorrection	20	173.6111	
DCM Maintenance	63	546.875	
Attitude Decode	45	390.625	
TOTAL	**226**	**1961.806**	<-Initial implementation is fast enough for 50Hz refresh.
GPS Code:	**Ticks**	**Time (us)**	
GPS_Update() - Parsing	110	954.8611	
GPS RX Sequence	1428	12395.83	
RS232 Code:	**Ticks**	**Time (us)**	
RTX permission from GPS	_	_	<-This is nearly always 481
(min)	41	355.9028	
(max)	271	2352.431	
ASCII Assemble+TX (23 Bytes)	4300	37326.39	
Binary Assemble+TX (29 Bytes)	481	4175.347	

Table 3: Navigation Computer Subroutine Benchmarks

From Table 3 we draw the following conclusions:

- The GPS RX Sequence can consume over 50% of the 20ms state time. However this is a UART transmission managed by the UART2 Receive ISR. Through benchmark testing we have discovered that ALL remaining _T2Interrupt code, ALL navigation function code, and ALL RS232 transmission initialization code can be executed concurrently and completed before the end of a GPS RX Sequence of 13ms.

- The navigation code takes less than 10% of the state time. For this reason the DSP library was not invoked.

- RS232 ASCII transmissions take up to 9 times longer than equivalent binary transmissions. This has two causes.
  1) As a rule of thumb, representing a number in ASCII will require more memory which will increase the UART transmission time.
  2) Assembling a string of characters from data types such as double, long, and integers can take more than 10 microseconds with the sprintf function.

- RS232 Binary Transmission time almost consists entirely of UART data flight time. Assembling a string of binary data from any data type usually takes a few microseconds.

5.2.4 Digital PI Controller

The current attitude is fed back and compared with Roll, Pitch, and Yaw correction vectors to form an error signal. The error signal is conditioned by the PI Controller in the forward loop. The Gyro Data rotates the DCM directly and can be modeled as a disturbance that is injected between the PI Controller and the DCM Rotation blocks. The simplified block diagram model including the PI controller can be found in Figure 14.

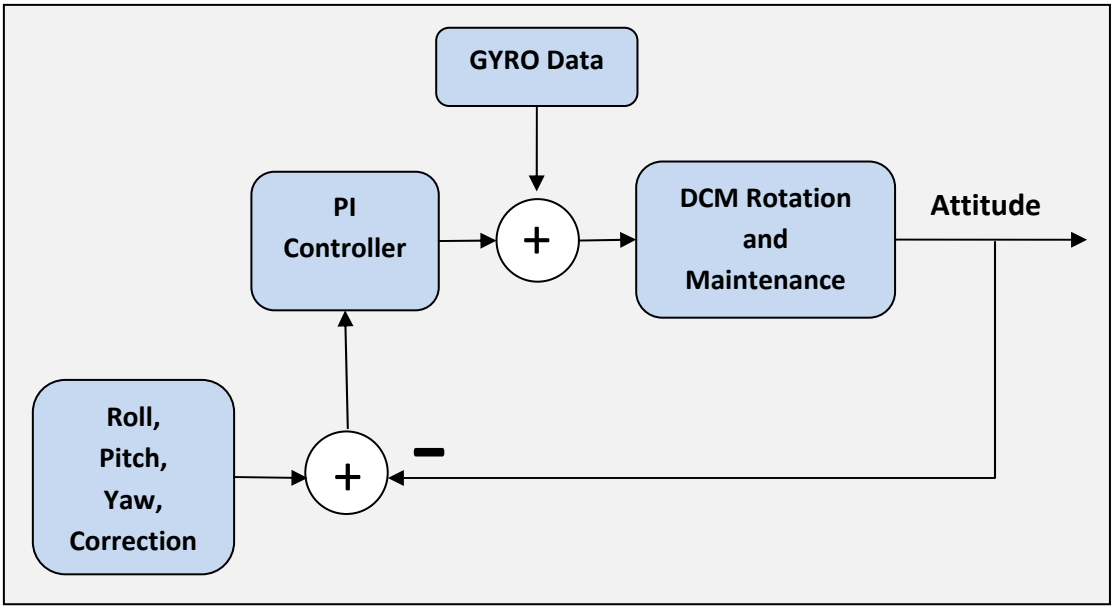


Figure 14: Simplified Block Diagram Model for the DCM Attitude Control Loop

Modeling and step response testing of this model can be found in the System Testing section.

5.3 XINS Ground Station – Communication

A navigation computer would normally communicate with an onboard autopilot servo controller or communicate with a ground station via wireless data link. The XINS project is still in the prototyping stage and neither an autopilot servo controller nor wireless data links are available. Therefore all communication happens between a personal computer and the XINS Navigation Computer through an RS232 serial connection.

There are three distinct forms of communication between the Navigation Computer and the Ground Station:

1. ASCII Data Transmitted to the Ground Station
2. ASCII Console Commands Transmitted to the Navigation Computer
3. Binary Data Transmitted to the XINSGUI

5.3.1 ASCII Data Transmitted to the Ground Station

Navigation Computer variables can be assembled into ASCII strings by the Nav Computer and transmitted to a COM port for viewing on a terminal program. Figure 15 shows a program called RealTerm displaying integer data gathered from the ADIS16365 IMU on the Ground Station PC.

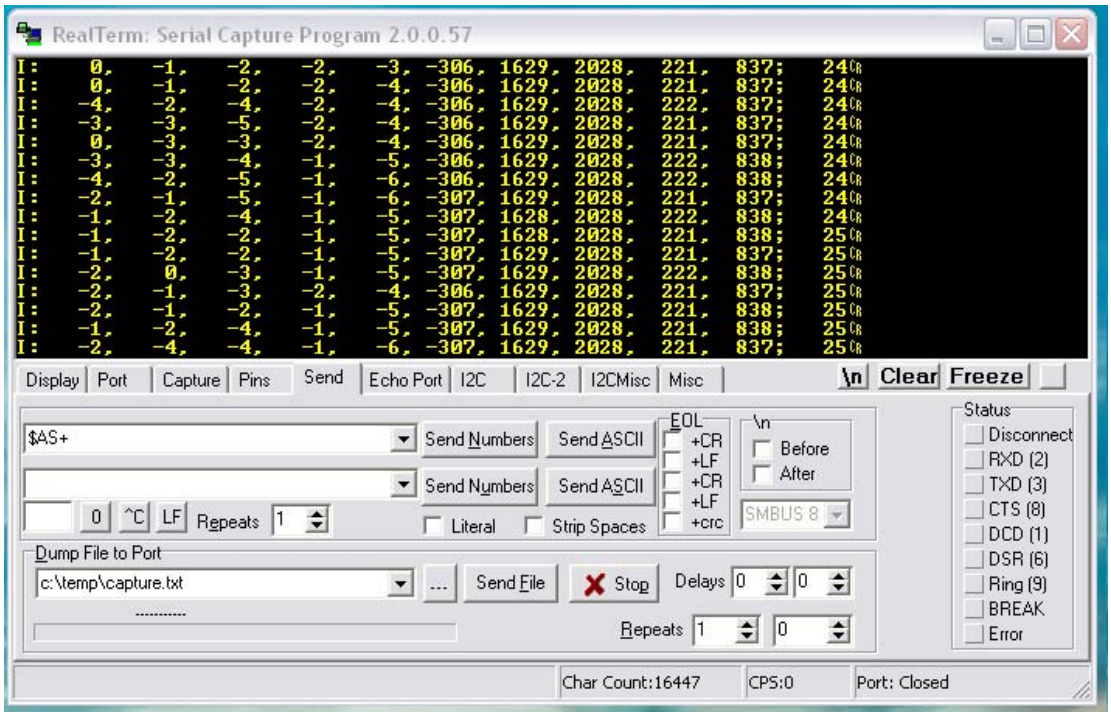


Figure 15: RealTerm Displaying IMU Integer Data in ASCII Format

Data sent from the Navigation Computer

This method of reading data is particularly useful for experimental testing and quick debugging of Navigation Computer software since new data sentences can be easily programmed and modified in the UART.c source file.

5.3.2 ASCII Console Commands Transmitted to the Ground Station

Any 8-bit characters to leave the Ground Station's serial com port end up being assembled into receive buffers and processed by the Navigation Computer. If the data received by the Nav Computer happens to be an ASCII '$' followed by a 3 character console command the Nav Computer will execute a preprogrammed subroutine.

This means any serial port program, including Realterm and Matlab, has access to the Nav Computer's command console. Figure 15 shows the command "$AS+" awaiting transmission by RealTerm. This particular command increments the ASCII Transmission selector so that a different ASCII sentence will be transmitted 5 times a second.

The command console is defined in the RS232_Update() function from the UART.c source file.

5.3.3 Binary Data Transmitted to the XINSGUI

Recall that Table 3 shows binary transmissions are far more economical for large data transfers in terms of time and processor overhead. Binary data is not readily understood

form terminal programs such as RealTerm and requires a more complicated Ground Station application to make the binary data accessible to the end user.

5.3.3a The XINSGUI

Sources: The source used to develop the XINSGUI'smatlab code was the website Mathworks.com [8]. This site aided in finding code to operate a customized Matlab GUI, managing Document Object Models and XML files, and utilizing serial connections. The source used to create a Google Earth KML plugin was the Google's own KML Documentation website [9].

The XINSGUI is a custom written application designed to receive and process binary transmissions sent from the Navigation Computer. The XINSGUI was designed in Matlab using GUIDE and compiled for standalone execution with the MCR (Matlab Component Runtime Library). Figure 16 shows the front-end user interface provided by the XINSGUI.

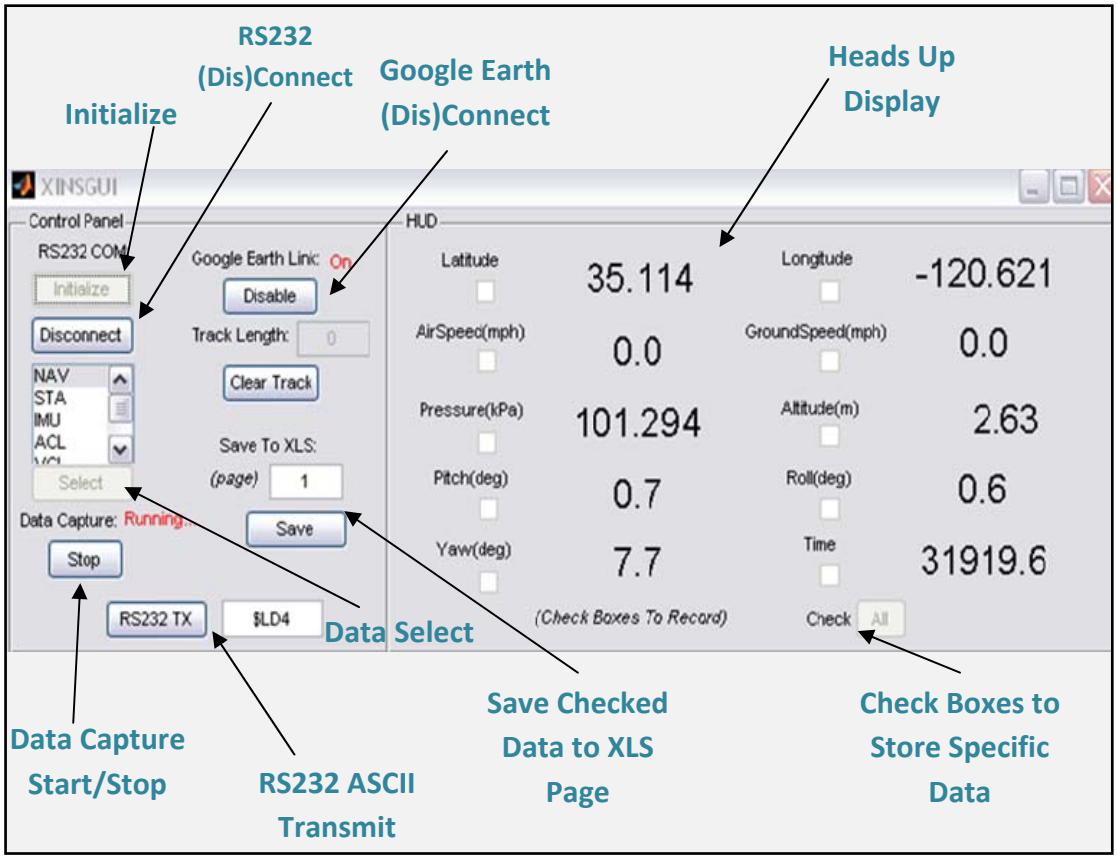


Figure 16: XINSGUI Front-End

The XINSGUI requests binary data sets from the XINS Navigation Computer, displays received data in real-time on the heads up display. It stores check-marked data in a matrix that can be saved in a page of an XLS spreadsheet. A Google Earth KML plug-in can be

activated to transmit Flight Model data to Google Earth for live tracking. The XINSGUI can also send custom RS232 console commands to the Navigation Computer.

Using the XINSGUI follows a sequential path of events. This path is described below. Refer to Figure 16 for handle names.

- **Initialize** – Upon starting the XINSGUI the only available interaction is to press the Initialize button. This button initializes all of the global variables used by the XINSGUI. This includes setting up the XML node structure for the Document Object Model (DOM) to be written as a Google Earth KML plug-in.

- **RS232: Connect/Disconnect -** The next available interaction after Initialize is Connect. This sets up the RS232 serial connection parameters then attempts to open the serial connection in Matlab. This button now toggles between Connect and Disconnect where it will create/open or close/destroy the serial connection.

- **Data : Select** – After successfully opening a serial connection a data set must be selected. The codenames are listed in the selection box as NAV, STA, IMU, etc. Pressing select will transmit an appropriate console command to the NavComputer requesting the selected data set, and fill in the appropriate data labels in the HUD (Heads Up Display). At this point any of the available check boxes in the HUD can be selected. The data who's check boxes are selected will be stored in a matrix for later.

- **Data Capture: Start/Stop** – After a data set has been selected the Data Capture: Start button can be pressed. This locks the check boxes, and begins to process received binary data. All received data is immediately posted to the HUD. Selected data is stored in a matrix.

  This function runs as a background task in a semi infinite loop. After sufficient experimenting it was discovered that when a button is pressed from a Matlab GUI the button calls a unique instance of its callback function. Pressing the Start button creates a unique instance of its callback that enters an infinite data gathering loop governed by global variables. The only way to terminate the loop is by running another unique instance of the SAME callback that modifies the state of the global control variables. The first instance of the callback responds to the change in control global variables and terminates its loop.

  Be warned: this exploitation may be a source of code legacy incompatibility in future

versions of Matlab .

- **RS232 TX -** When this button is pressed the contents of the adjacent text box are transmitted to the Nav Computer. This allows customizable control in the XINSGUI, for instance a user can send the console command, "$NVI", to re-initialize the Flight Model.

- **Save To XLS: Save –** When all Data Capture instances are stopped, the matrix that stored all check-marked HUD data can be saved to a custom page in the XINS.xls in the XINSGUI's root directory.

  Be warned: re-initializing a Data Capture will reset the storage matrix, even if the same data set is being transmitted. For this reason: remember to save desired data immediately after stopping a Data Capture instance.

- **Google Earth Link: Enable/Disable –** Enabling the Google Earth Link will cause the Document Object Model (DOM) to be updated with received Flight Model data including: longitude, latitude, heading, airspeed. After the DOM is updated the xmlwrite() Matlab function is used to write the DOM to XINS-Data.kml located in the XINSGUI root directory. Figure 17 illustrates the XINS plug-in actively transmitting navigation data to Google Earth.

  XINS.kml is the static read-only file plug-in for Google Earth located in the XINSGUI root directory. XINS.kml points to the XINS-Data.kml as the dynamic source of data for XINS Google Earth plug-in. This allows the XINS plugin to be installed to Google Earth without the XINSGUI active. As soon as the XINSGUI updates XINS-Data.kml Google Earth will respond to any changes through the XINS plugin.

  Note: The Google Earth Link only works if the NAV data set is selected.

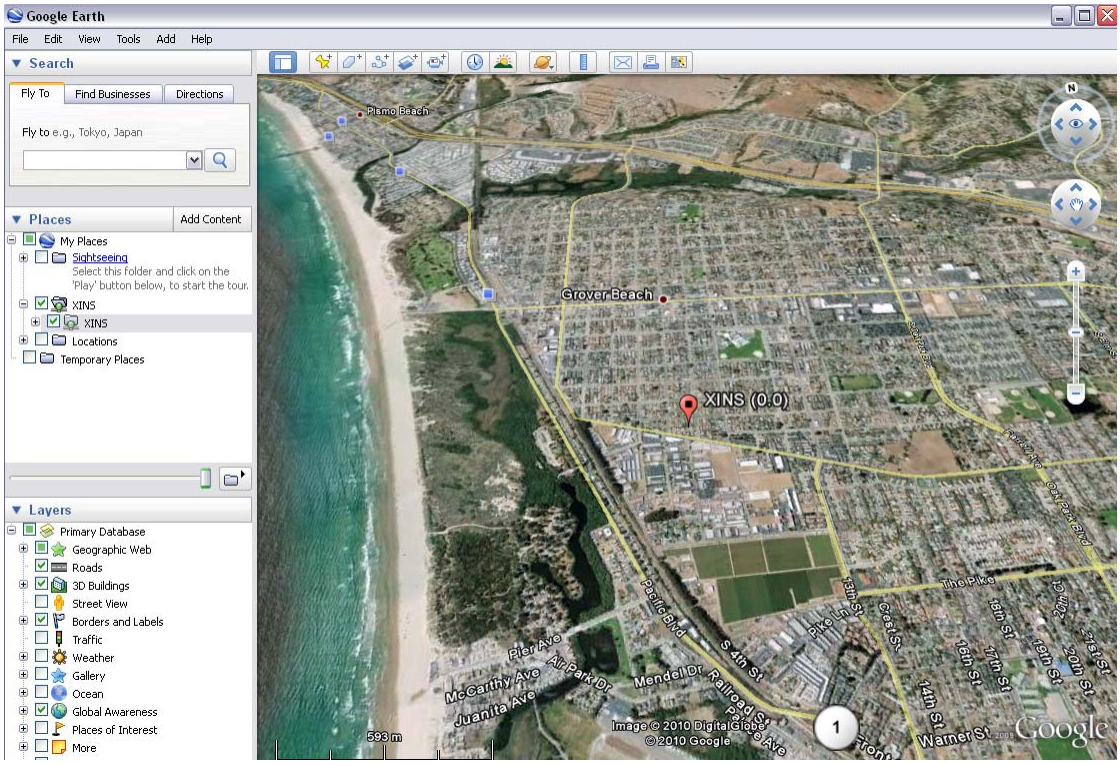


Figure 17: Google Earth Utilizing the XINSGUI's KML Plug-in Labeled "XINS"

6. System Testing

System wide testing took place in three major categories. These are:

- Altitude and Airspeed Testing along the Cuesta Grade
- Latitude and Longitude Testing with Google Earth Tracks
- Attitude Testing: Gyro Integrity & PI-Controller Correction Response

These tests were primarily conducted with the XINS Navigation Computer and Ground Station onboard an automobile. The automobile at hand was a 1989 Honda CRX equipped with a 120VAC inverter to supply power to the dsPIC development board and to the Dell Inspiron 600m running the Ground Station software.

Attitude PI-Controller testing was conducted in lab in addition to onboard the automobile. The in-lab testing includes computer simulated step responses as well as experimental step-responses.

6.1 Altitude and Airspeed Testing

The altimeter responded to changes in elevation in a very fluid manner. The chart in Figure 18 shows a continuous Altitude plot.

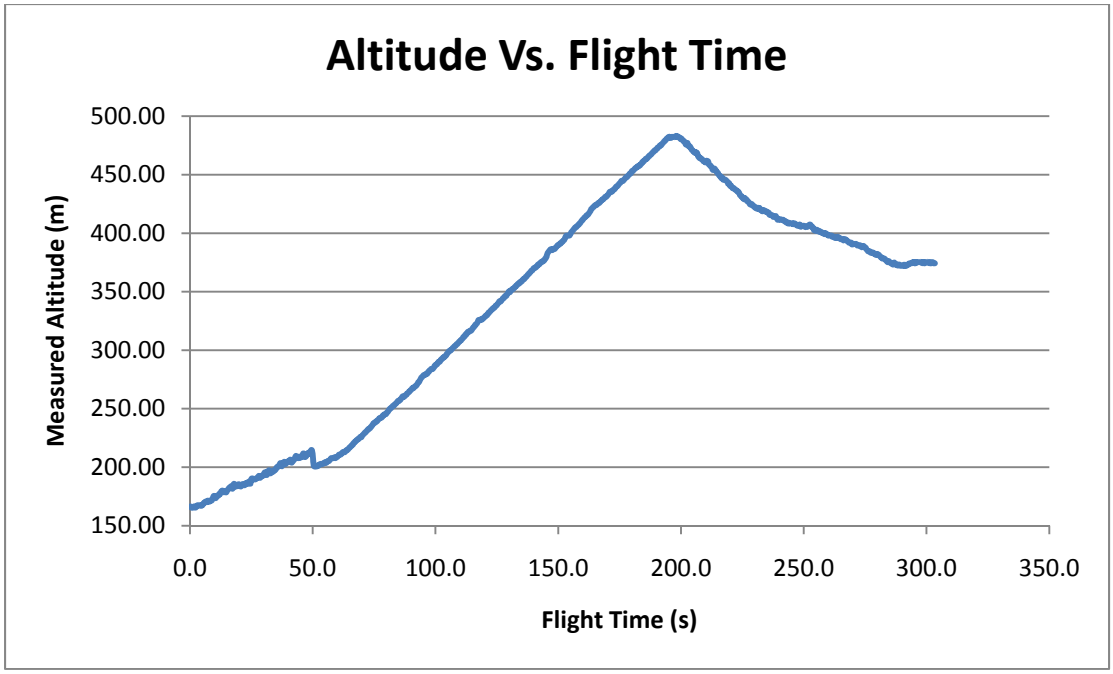


Figure 18: Cuesta Grade – Altitude vs. Flight Time

The accuracy of each specific measurement is hard to quantify. However the peak altitude measured can be checked against the posted altitude of 473 meters.

Variable	Value	Units
max recorded altitude:	483.0858927	m
posted Cuesta Grade max alt:	473.0496	m
~Error:	2.12	%

Table 4: Cuesta Grade - Max Recorded Altitude vs. Posted Max Altitude

Given the approximate altitude-pressure model 2.12% error is acceptable. A more accurate model depending on present atmospheric conditions could possibly help to minimize this error. However aircraft flight would not require better accuracy than 2% since there should be a good deal of altitude to work with when in flight.

Airspeed was easy to verify while driving. The Airspeed and Ground Speed stay relatively close to each other since the magnitude of the pitch attitude while driving cannot be much greater than a few degrees. The Airspeed measured responded more quickly to changes in speed than the vehicles own speedometer. Table 5 shows a simple integration of a test run over the Cuesta Grade.

Variable	Value	Units
Average Airspeed	61.59	MPH
Average Ground Speed	61.47	MPH
Flight Time	295.0	s
Integrated Flight Distance	5.047	miles
Flight Distance	5.0	miles per Google Earth
~Error	0.94	%

Table 5: Integrating Airspeed to Approximate Distance Travelled

The average Airspeed and Ground Speed just under the posted speed limit of 65 MPH is predictable given initial acceleration and final decelerations during the trip. When the airspeed is integrated with respect to flight time we estimate a total distance travelled of 5.047 miles. This correlated with Google Earth's estimate of 5.0 miles. Since Google Earth only gives estimates to the tenth's place we can assume an Error of about 1%.

6.2 Latitude and Longitude Testing with Google Earth Tracks

As mentioned before, the XINS does not yet process changes in latitude and longitude to develop a local model of distance. At this point the best confirmation that our GPS unit is functioning as desired is to overlay the latitude and longitude coordinates on a predetermined route. This is done quite easily while using Google Earth. Figure 19 shows the

coordinates for same test run used for Altitude and Airspeed above as they are mapped along Highway 101.

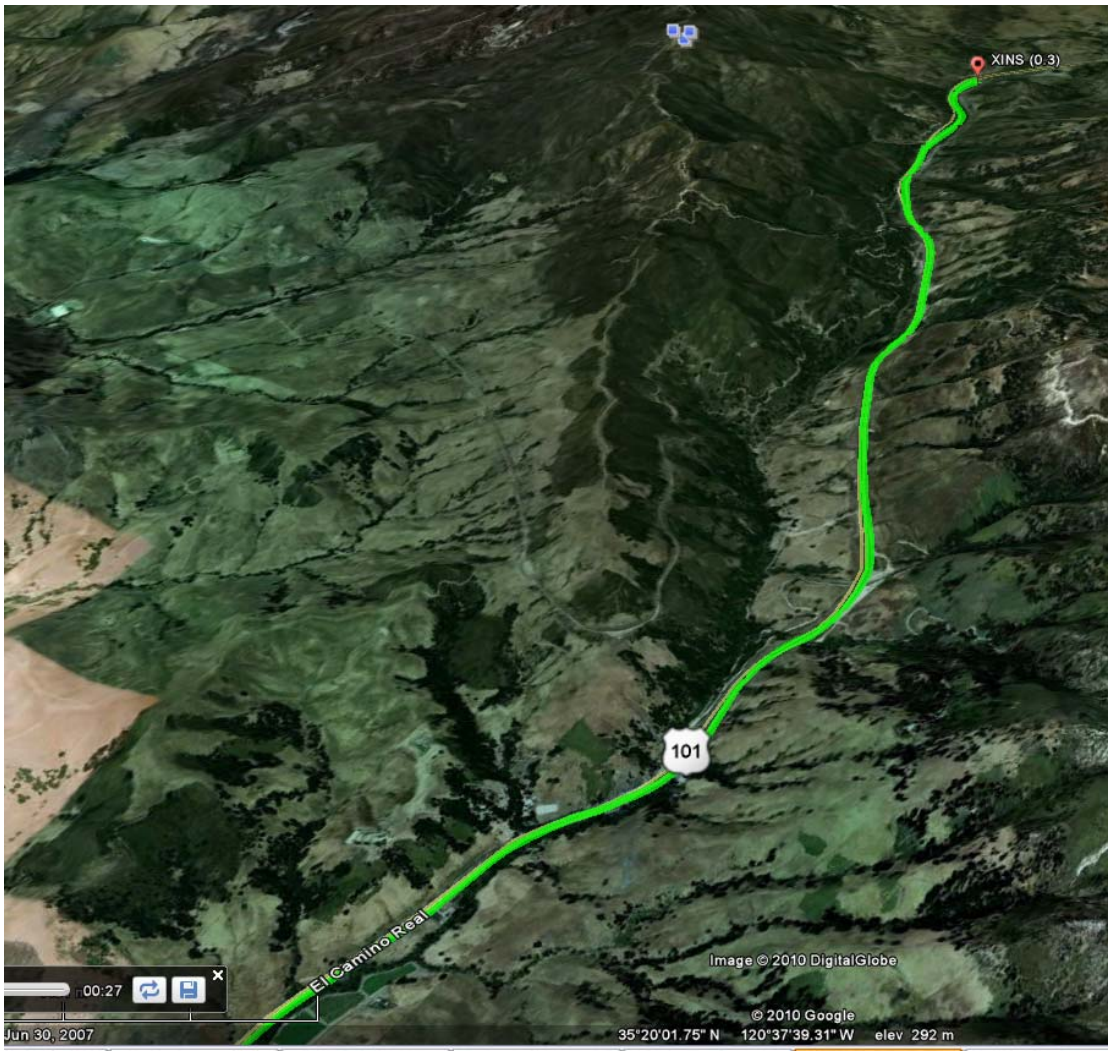


Figure 19: Measured Latitude and Longitude along Cuesta Grade

–overlaying US Highway 101 on the Google Earth application.

From a rough visual the latitude and longitude coordinates correlate well enough to begin flight testing the GPS data on a UAV.

6.3 Attitude Testing

Attitude testing during the previously mentioned Cuesta Grade test was unspectacular. An automobile is tied to many attitude constraints. However we can draw some conclusions from the automobile tests.

First, the maximum magnitude for the roll attitude was measured to be 6.5°. This seems reasonable given that the vehicle banks slightly during the turns in the highway. However there was no identifiable drift past 6.5° which indicates that the Roll/Pitch correction is functioning.

Second, notice in Table 6 that as the Yaw angle shows a turn from left to right the bank in the road is a positive roll attitude. This roll angle is to be expected so that the accelerations from the turn are absorbed more uniformly across the vehicle. During turns from right to left the bank in the road is a negative roll attitude.

Roll(deg)	Yaw(deg)	Flight Time(s)
3.3	-37.7	16.6
3.2	-35.5	16.8
3.2	-34.3	17
3.2	-31.4	17.2
3.3	-28.6	17.4
3.2	-26.4	17.6
3.2	-24.7	17.8
3.2	-23	18
3.2	-20.7	18.2
3.2	-20.1	18.4
3.1	-18.4	18.6
3	-15.9	18.8
2.8	-14.4	19
2.7	-12	19.2
2.6	-9.6	19.4
2.4	-8.2	19.6
2.2	-6.4	19.8

Table 6: Highway Banking Shown in Yaw and Roll Attitude Angles

Lastly, the pitch attitude makes an identifiable shift at the peak of the ascent from positive to negative. This transition can be seen in Table 7.

Altitude(m)	Pitch(deg)	Flight Time(s)
479.16	4.2	193.6
480.21	4.2	194.0
481.25	3.2	194.4
481.71	2.6	194.8
482.30	2.4	195.2
482.11	2.0	195.6
481.58	2.1	196.0
482.30	1.4	196.4
482.30	0.8	196.8
482.30	0.4	197.2
482.30	-0.1	197.6
483.09	-0.5	198.0
482.50	-0.9	198.4
482.30	-0.9	198.8
482.04	-1.3	199.2
481.12	-1.8	199.6
480.97	-2.8	200.0
480.21	-3.0	200.4
479.10	-3.3	200.8

Table 7: Pitch Attitude Transition Highlighting the Altitude Peak

6.3.1 PI Controller Testing

The PI Controller explained on page was simulated in Matlab. The desired step response depends on the aircraft capabilities. However in general, it would seem that the Yaw attitude correction should have a slower step response than the Roll / Pitch correction. This is because the GroundCourse vector is updated at a maximum of 5Hz, therefore its correction should be slow enough to allow for obsolete heading information to expire.

The Roll/Pitch correction comes from the gravity vector calculated from IMU data. This vector is updated at a maximum of 50Hz therefore the step response settling time can be increased with less concern for obsolete data. However care must be taken since the gravity vector can be tainted with accelerations due to thrust/drag, lift/weight, and wind.

The response characteristics chosen were:

- Yaw Correction Step Response Settling Time: **12 seconds**
- Roll/Pitch Correction Step Response: **6 seconds**
- No overshoot, No steady state error

Despite the large amount of 3-space vector math involved in calculating DCM correction rotations, the system's block diagram from Figure 14 can be simplified further for analysis. Figure 20 shows the closed loop system with transfer function boxes.

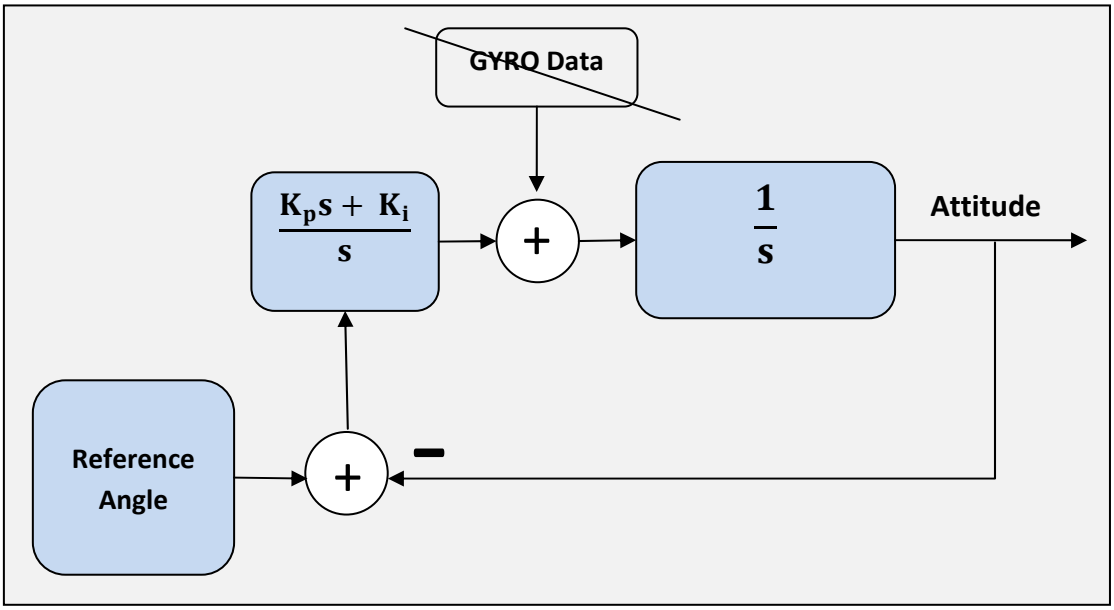


Figure 20: Closed Loop System with Transfer Functions

The Gyro Data is ignored for analysis. The transfer functions for this model can be found below:

$$G = \frac{K_p s + K_i}{s^2} \qquad \textit{Open Loop Gain} \qquad \textit{Equation 39}$$

$$H = 1 \qquad \textit{Feed Back Gain} \qquad \textit{Equation 40}$$

$$T = \frac{K_p s + K_i}{s^2 + K_p s + K_i} \qquad \textit{Closed Loop Gain} \qquad \textit{Equation 41}$$

6.3.2 Transfer Function Gain Selection

The correction in general should be soft. The integral controller only exists to filter out long term Gyro drift so it should be especially soft. The Ki and Kp values were chosen to be 0.5 and 0.005 respectively.

If you recall from Equation 18 that the Yaw correction and the Roll/Pitch correction have individual coefficients that can be modeled as independent gains in the forward path of Figure 20. These individual can be seen in Equation 42.

$$T_{Yaw} = \frac{W_Y\, G}{1 + W_Y\, G} \quad \& \; T_{Roll-Pitch} = \frac{W_{RP}\, G}{1 + W_{RP}\, G} \qquad \textit{Equation 42}$$

The step response for Equation 41 applied to the angular corrections can be found in Figure 21.

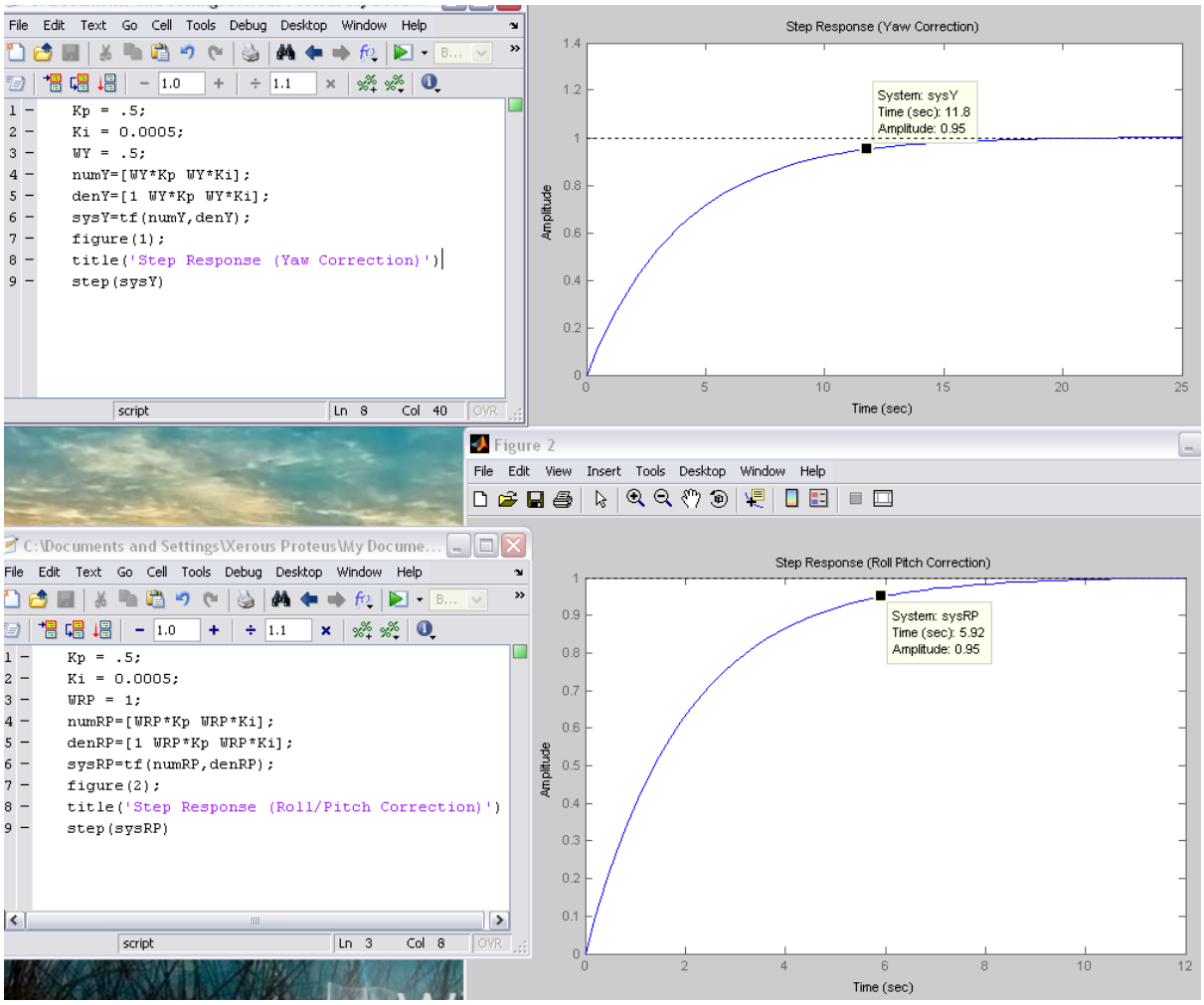


Figure 21: Matlab Step Response Simulations for the Attitude Correction

The gains $\mathbf{W_Y}$ and $\mathbf{W_{RP}}$ were selected to be **0.5** and **1** respectively to obtain simulated settling times of **11.8s** and **5.92**s for Yaw and Roll-Pitch rotations.

6.3.3 Testing the Selected Gains on the XINS

These selected gains were appropriately set on the XINS. The correction reference angles were set to change from a steady state of an attitude with Roll = Pitch = Yaw = 0. Figure 22 shows the Yaw correction experimental step response and settling time, and Figure 23 shows the Roll/Pitch experimental step response.

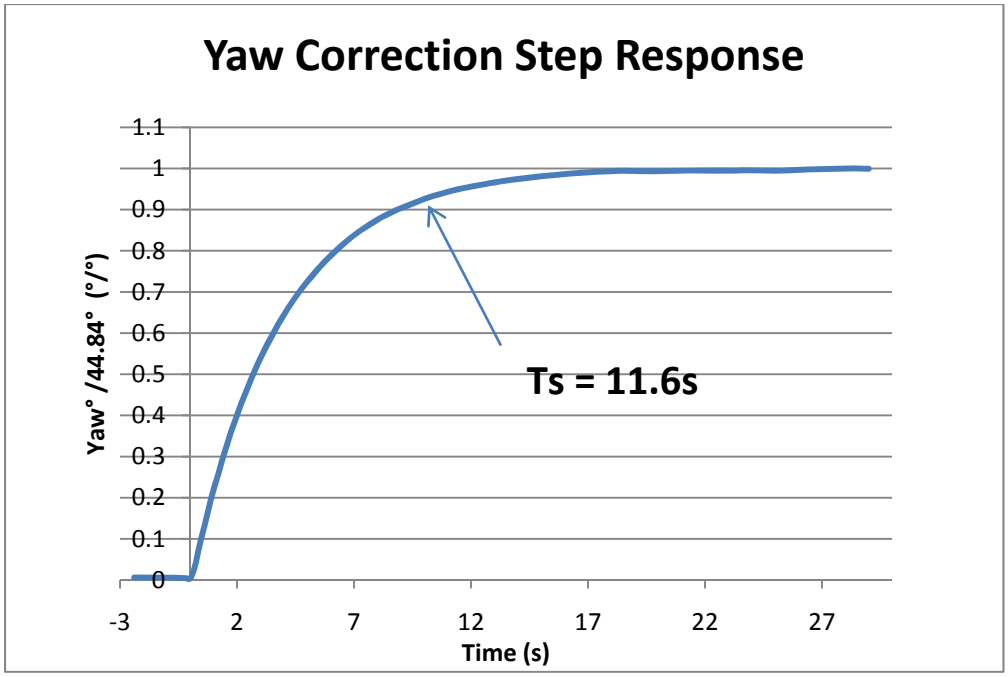


Figure 22: Experimental Yaw Step Response

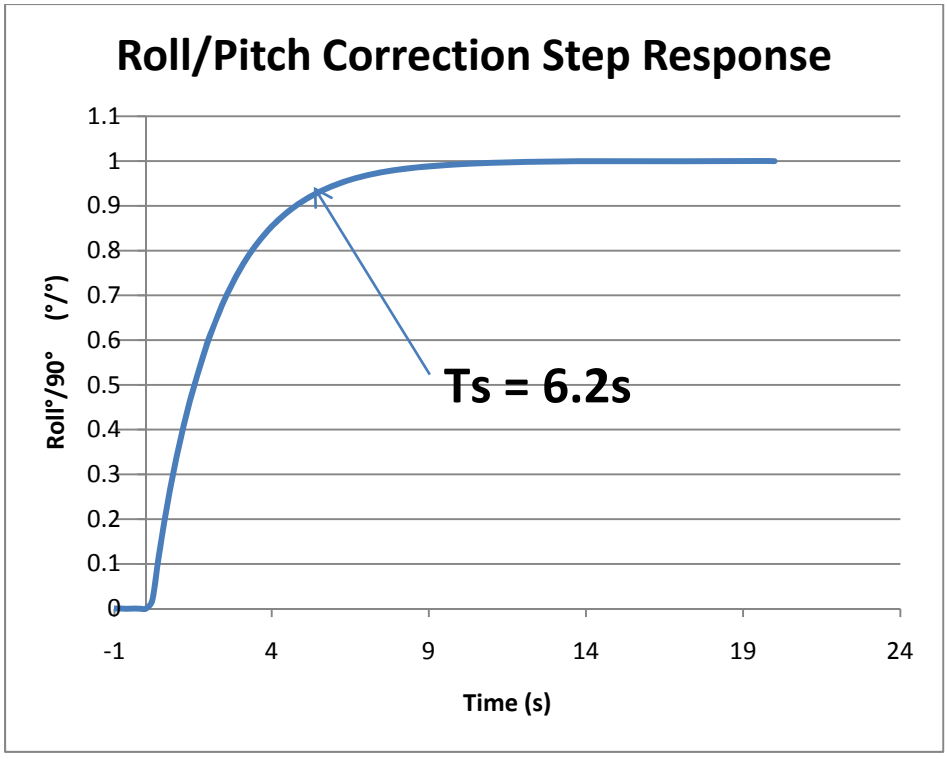


Figure 23: Experimental Roll/Pitch Step Response

The experimental results were within specifications. Table 8 summarizes the PI Controller results.

Gains:	**Ki**	**Kp**	**W_Y**	**W_{RP}**
	0.005	0.5	0.5	1
	Simulation	**Experimental**	**Error (%)**	
Ts (Yaw) (s)	11.8	11.6	-1.69	
Ts (Roll Pitch) (s)	5.92	6.2	4.73	

Table 8: PI Controller Specs and Test Results

7. Future Design Goals

The ultimate goal for the XINS is to be integrated into a fully functional UAV autopilot. However there are some key design goals for the XINS that must be fulfilled before this autopilot integration is possible.

7.1 Power Supply with Battery

A custom power supply unit must be designed. It must be capable of powering the dsPIC, IMU, Op-Amps, and GPS for an extended period of time on a battery capable of being installed on a small UAV.

7.2 Custom Printed Circuit Board

A custom circuit board must be designed to significantly cut down the size of the Navigation Computer. The dsPIC development board currently used in the XINS prototype is a large waste of power and space, therefore designing a compact PCB will enable the XINS Navigation Computer to physically fit onboard a small UAV. It could also increase the reliability of the interconnections and solder joints.

7.3 Wireless Data-Link

A fairly long-ranged low-power wireless data-link is necessary to communicate with the XINS Navigation Computer during flight.

8. Conclusion

The XINS as a whole has come together as planned. The microcontroller was able to interface all the required sensors to create a Flight Model. The IMU data, pressure data, and GPS data was available at speeds and accuracy sufficient for developing an autopilot. The microcontroller software coding was made fast, reliable, and readable with the C-programming language. The XINSGUI developed in Matlab made communicating with the XINS Navigation Computer much easier than what was initially expected. And overall ground testing showed results that would be expected from an automobile.

This XINS prototype has satisfied all of the original design goals of this senior project. It provides a verifiable model of Flight Data to a Ground Station. Although more testing is necessary for a complete autopilot implementation, much of the testing would be best accomplished onboard an aircraft. Therefore we can only hope the future design goals will be addressed soon!

Bibliography

1) Premerlani, William, and Paul Bizard. *Direction Cosine Matrix IMU: Theory*. Tech. May 2009. Web. 9 Mar. 2010. <gentlenav.googlecode.com/files/DCMDraft2.pdf>.

2) *Pitot-Static Tube*. Rep. NASA. Web. 9 Mar. 2010. <http://www.grc.nasa.gov/WWW/K-12/airplane/pitot.html>.

3) Nave, C.R. *The Barometric Formula*. Rep. Georgia State University, 2005. Web. 9 Mar. 2010. <http://hyperphysics.phy-astr.gsu.edu/hbase/Kinetic/barfor.html>.

4) Brewer, Jeff. "Op-Amp Design." Personal interview. 2009.

5) Milivojevi?, Zoran, and Djordje ?aponji?. *Programming DsPIC (Digital Signal Controllers) in C*. *MikroElektronika*. Web. 9 Mar. 2010. <http://www.mikroe.com/en/books/dspicbook/mikroc/>.

6) *DsPIC30F Family Reference Manual*. Rep. Microchip, 2005. Print.

7) *MPLAB C30 C Compiler User's Guide*. Microchip, 2007. Print.

8) *The MathWorks - MATLAB and Simulink for Technical Computing*. Web. 09 Mar. 2010. <http://www.mathworks.com/>.

9) "KML Documentation Introduction - KML -." *Google Code*. Web. 09 Mar. 2010. <http://code.google.com/apis/kml/documentation/>.

Appendices

Appendix A - Parts List & Cost,

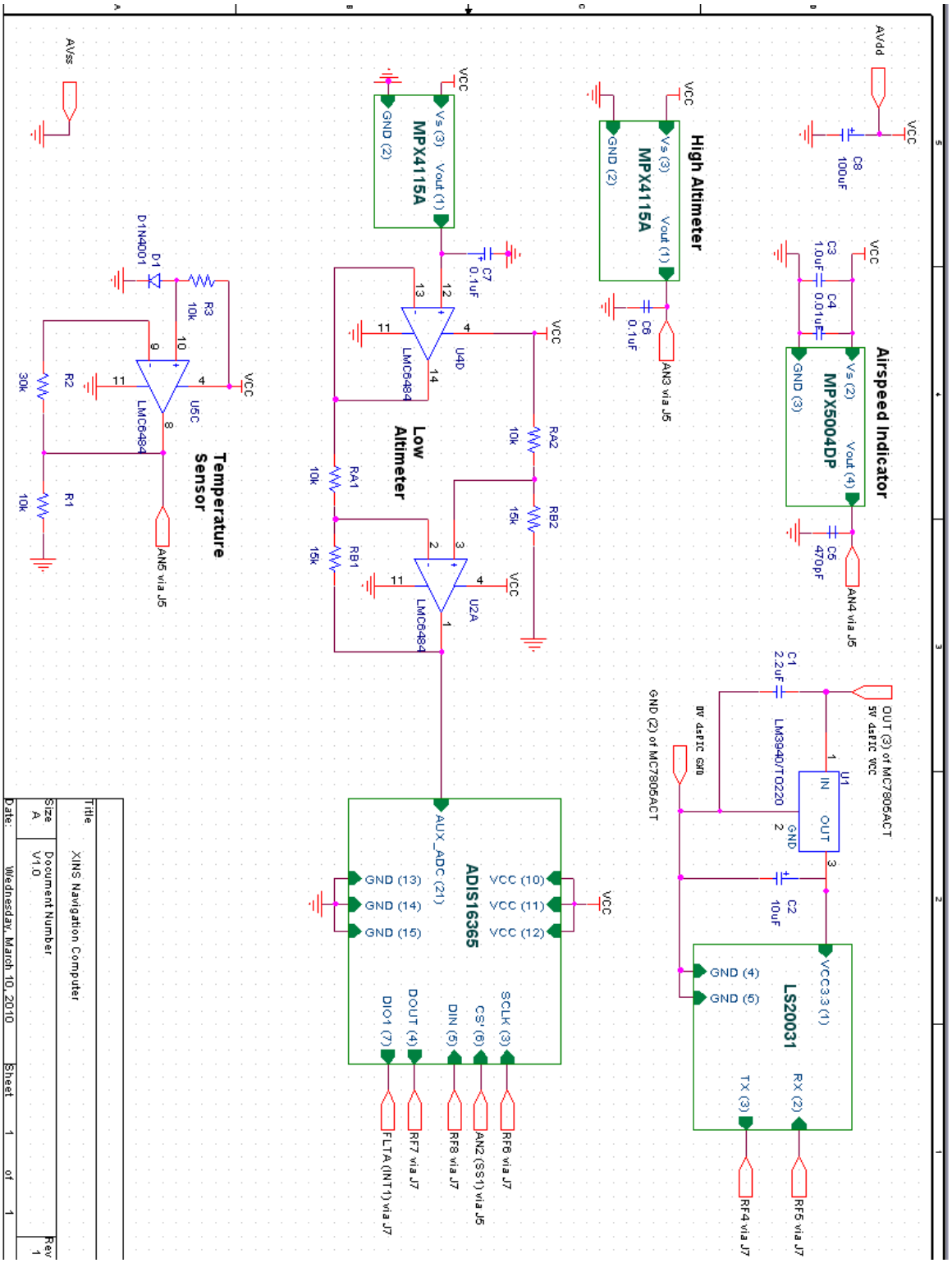
Part	Cost
20 - 10Kohms 1% Tolerance Resistors *Metal Film 1/4watt*	$2.80
1 – LMC6484 Quad Rail-to-Rail OPAMP	$3.88
2 – 0.1uF 100V Ceramic Capacitors	$0.50
2-MPX4115AP	$21.90
1- ADIS16365-IMU	Acquired (~$600.00)
1-Microchip dsPIC6010F with MC1 Development Board	Acquired (~$300.00)
1-MPX5004DP with Breakout Board / Recommended Capacitors and PITOT Tube	$19.76
1-LM3940-3.3V-LDR	(Samples)
1-LS20031-GPS	$47.96
1-Samtec Female Header for ADIS16365	(Samples)
1-SchmartBoard for ADIS16365 with 1mm Pitch	$9.99
Total Cost	$106.79 ($1006.79+ Value)

Note: *This is a base cost of the parts required to recreate the XINS prototype hardware. This is assuming breadboards, prototype wires, wire-wrap, solder, RS232 serial cables, etc. are supplied by the designer.*

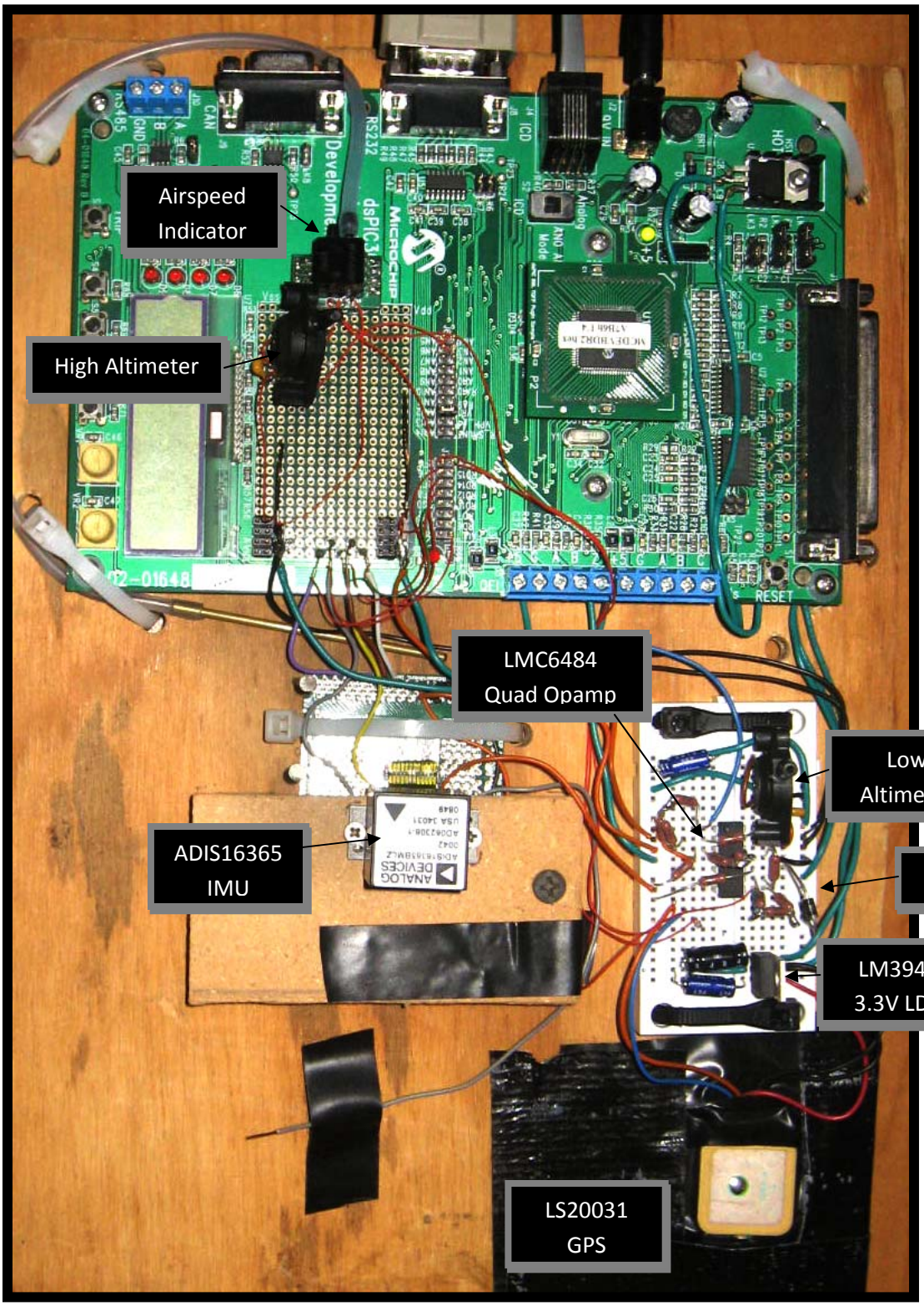
Appendix B – Design Schedule

	Name	Start	Finish	Work	Complete	Notes
1	Project Brainstorm	Jul 6	Jul 10	5d	100%	Brainstormed ideas for what components a UAV would need to navigate. The conclusion was at least a microprocessor, a GPS, and an IMU (Inertial Measurement Unit). Since I had little previous experience with any of this equipment I decided to tackle the separate hardware one piece at a time. Then block by block I could assemble the whole system.
2	**Hardware**	Jul 13	Aug 11	29d		**Acquire & Familiarize**
2.1	dsPIC Familiarization	Jul 13	Jul 17	5d	100%	Acquired the dsPIC30f6010 MC1 development board and began to familiarize myself with the software, programming, and hardware control.
2.2	GPS Familiarization	Jul 20	Jul 31	10d	100%	Purchased the Locosys LS20031 GPS smart antenna module along with a TTL to USB widget to begin configuring the module through HyperTerminal and MATLAB. Then the LS20031 was interfaced with the dsPIC through a direct TTL connection and the appropriate software was coded.
2.3	IMU Familiarization	Jul 27	Aug 7	10d	100%	The Analog Devices IMU "ADIS16365" was acquired. A delay was found here when a special surface mount female connector from SAMTEC was required to physically connect to the IMU. The female connector was soldered to a Schmartboard to allow wired connections to the dsPIC. Then the SPI interface was programmed with much troubleshooting. Using SPI Interrupts proved fault ridden therefore the entire SPI data transfer had to be handled by the main loop code.
2.4	Assemble Pressure Sensor BreadBoard	Aug 6	Aug 11	4d	100%	Three pressure sensors were acquired. The differential pressure sensor already had a breakout board with suggested noise filtering capacitors attached. The two absolute pressure sensors needed filtering capacitors as well as amplification before being sampled by the dsPIC's ADC's. The amplification was done with rail-to-rail quad op amps, resistors, and voltage references to zone in on the specific pressure reading's I wanted each sensor to provide. So far the existing pressure system is functional but needs improvement. It appears the pressure readings on all 3 have a significant amount of variation. This is the next foreseeable task to complete.
3	Software Familiarization	Jul 27	Aug 21	20d	100%	The core navigation system software was written from the start of dsPIC hardware interfacing. Each peripheral usually added another unique ISR to transfer data. By the end of August I was toying with digital control loops. However that remains as a large project to be completed later.
4	BREAK	Aug 24	Jan 4	96d	100%	FOR VACATION / FALL2009 QUARTER
5	Review and Plan Project Tasks	Jan 4	Jan 5	2d	100%	Returned from 100day break. Refresher necessary! At this point the dsPIC hardware is completely configured. Also the GPS and IMU data is accurate and dependable. However: -Pressure Sensor data must be more accurate to prove useful -A digital control system must be implemented on the raw data to provide a model of flight data. -The UAV-NAV-SYS must output the flight data to something (RS232 UART to PC) -The flight data must be recorded and plotted to properly evaluate system performance. -All must be reported! Wed 06 Jan 2010, 12:15
6	Improve Pressure Data Accuracy	Jan 6	Jan 15	8d	100%	One of these must be improved for sensor accuracy: -Analog Filtering -Digital Filtering -Analog Differential Amplification Wed 06 Jan 2010, 12:17
7	Implement Kalman-like Filter on sensor data in C	Jan 18	Jan 29	10d	100%	This will involve C-programming and will likely include the dsPIC's DSP library to efficiently execute linearly algebraic instruction
8	GUI Design	Feb 1	Feb 12	10d	100%	Create MATLAB Application to process, record, and plot Aircraft Flight Data
9	Test Flight (Honda CRX Freeway Driving)	Feb 15	Feb 19	5d	100%	We don't have a UAV with the hardware to record flight data, or the power to sustain the UAV-NAV-SYS in flight. Therefore this project will simply test flight a Honda CRX on freeways near sea level to give the illusion of continuous fixed wing flight.
10	Write Report	Feb 22	Mar 5	10d	100%	

Appendix C – XINS Navigation Computer Circuit Diagram



Appendix D – XINS Navigation Computer Prototype Layout



Appendix E - Navigation Computer C Source Code

Source code is attached below…

Appendix F – Ground Station XINSGUI Matlab Source Code

Source code is attached below…

Appendix E - XINS Source Code

The following source code was written for the dsPIC6010F on Microchip's C30 compiler with the MPLAB IDE. The source code listed below has been formatted to fit a printable document. The source files are listed in the following order:

XINS.c

```
//XINS.c
//Author: Kyle Howen - 03-08-2010
//Desription: This contains the main Nav Computer initialization
//            and loop code.
#include "XPIC.h"
/*Oscillator Setup*/
_FOSC(0x008307); //FCY = 29.4912MHz = 7.3728MHz * (16PLL / 4)
_FWDT(WDT_OFF);
//_FBORPOR(PBOR_ON & BORV_20 & PWRT_64 & MCLR_EN);

//Initialize External Globals
struct ButtonSystem Button;
struct IMUSystem  IMU;
struct UARTSystem       RS232;
struct GPSSystem  GPS;
struct NAVSystem  NAV;

//Non-External Globals (Specific to INS.c)
    //None at the moment!

int main(void)
{
    HardwareInit(); //Complete Hardware/NAVSystem Initialization

    while(1)//Main Loop
    {
        if(NAV.NewData == 1){
            //Calculate the time passed in seconds.
            NAV.StateTime = NAV.DataAge * NAVSTATETIME * T2SecondsPerTick;

            //Crunch all sensor code.
            NAV_Sensors(NAV.StateTime);

            //Create a rotation correction from the PI Controller
            NAV_DCM_WCorrection(NAV.StateTime);

            //Update DCM with Gyro Measurements
```

```
            NAV_DCM_Update(NAV.StateTime);

            //Normalize and Orthogonalize the DCM
            NAV_DCM_OrthoNorm();

            //Make a copy of the DCM for calculation use
            NAV_DCM_Copy(0);

            //Decode Attitude from DCM
            NAV_DCM_Decode();

            //Reset the data flags
            NAV.NewData = 0;
            NAV.DataAge = 0;
        }

        if(GPS.Lock == 0 && GPS.NewData == 1){
            GPS_Update(1);      //Process GPS Data when available
            GPS.NewData = 0;    //Reset the data flag
        }

        if( (GPS.Lock == 1  && GPS.AllowRS232 >= RS232TXRATE     )  ||
            (NAV.GPSLOST    && NAV.StateCounter > 10*RS232TXRATE ) )
        {   //RS232 transmissions are processed during every GPS recieve
            //sequence OR every 10 states if the GPS unit is not
            //functioning.
            RS232_TX();
            GPS.AllowRS232 = 0;
            if(NAV.GPSLOST)
                NAV.StateCounter = 0;
        }

        //Check for any recieved console commands from the ground station
        RS232_Update();

        //ProcessButtons();
        //The UART Console has taken the place of button processing.
    }
      return 0;
}

void HardwareInit(void)
{
      //Interrupt Priorities
      IPC1 = 0x1511; //Timer-2 to 5
      IPC2 = 0x4113; //ADC Priority 6, SPI Priority 4
      IPC6 = 0x1117; //UART2RX-7, Top Priority

      //Interrupt Enabling
      IEC0   |= 0x0040;   //T2
      IEC0   |= 0x0400;       //U1TX
      IEC0   |= 0x0200;       //U1RX
      IEC1   |= 0x0200;   //U2TX
      //IEC0 |= 0x0008;   //T1
      //IEC0 |= 0x0800;   //ADC
      //IEC0 |= 0x0100;   //SPI1
      //IEC1 |= 0x0001;   //INT1
```

```
//IEC1 |= 0x0100;   //U2RX

//External Interrupt 1
TRISE |= 0x0300; //(RE8 and RE9 inputs)
IFS1  &= 0xFFFE;  //Clear Interrupt Flag

//ADC   - PRESSURE GAUGES & VR2
TRISB   = 0xFFFB; //Analog Input Ports
ADCHS   = 0x0027; //TRISB pins 3,4,5,& 7 Analog Inputs
ADPCFG  = 0xFF47; //TRISB pins 3,4,5 & 7 Analog Inputs
ADCON3  = 0x043A; //TAD = 1us, 4 TAD per Sample
ADCON2  = 0x030C; //Int upon 2 Samp/Conv
ADCON1  = 0x00E8; //Manual Sampling
ADCON1 |= 0x8000; //ADC on*/

//UART1 - RS232 TERMINAL OUTPUT
U1BRG   = 31; //9600Baud @ 7.3728MHz, 38.4kbps @ 29.4912MHz
U1MODE |= 0b1000000000000000; //UART Enable
U1STA  |= 0b0000010000000000; //Tx Enable,
U1STA  |= 0b1000000000000000; //Interrupt on Empty Tx FIFO
IFS0   &= 0xFBFF;
RS232.Lock = 0;
RS232.TxComplete = 1;
RS232.TxIndex = 0;

//UART2  - LOCOSYS GPS TRANSMISSION
U2MODE   = 0x8000;  //Enable UART2
U2STA   &= 0xFF3F;  //Interrupt upon first received byte in TX FIFO
//U2STA &= 0x0080;  //Interrupt on 3/4ths full buffer.
//U2STA |= 0x00C0;  //Interrupt upon full RX FIFO
IFS1    &= 0xFCFF;  //Clear UART2 Interrupt Flags (RX & TX)
//IFS1  &= 0xFFDF;  //Clear UART2 TX IFlag //IFS1  &= 0xFFEF;
U2BRG        = 31;  //Baud rate of 38.4kbps @ 29.4912MHz
GPS.Lock = 0;

//SPI1        - ADIS16365 IMU SCOM
TRISG    |= 0x0002; //DIO1 Data Read Input Flag
TRISG    &= 0xFFFE; //SS1 Output Control Line
SPI1CON   = 0b0000010011101101; //368.640 kHz SPI Clock
SPI1STAT  = 0x8000; //Enable SPI1
IFS0         &= 0xFEFF; //Clear SPI Interrupt Flag
SPI1STAT &= 0xFFBF; //Clear the overflow flag

//TMR1 INIT
TMR1         = 0;
PR1          = 50000;   //Interrupt period 2Hz
//T1CON  = 0x8030;      //Timer1 enabled (clock divided by 256)

//TMR2 INIT The State Time Keeper
TMR2         = 0;
PR2          = NAVSTATETIME;//Set this arbitrarily high...
T2CON        = 0x0020;      //Timer2 (internal clock divided by 64)

//TMR4 INIT The Sample Interval Controller
TMR4    = 0;
PR4     = 60000;
T4CON   = 0x0030;
```

```
	//LEDs And Switches
	TRISA       &= 0b0011100111111111; //LED Ports to Output
	TRISG       |= 0b0000001111010000; //Switch Ports to Input

	//Final Hardware Init Code
	UART_Init(); //Initialize RS232 transmission variables
	IMU_Init();  //Send configuration commands to the IMU
	GPS_TX(2);   //Transmit the setup sentence to the GPS.

	delay_ms(1000);  //Allow a second for sensors to initialize.
	NAV_Init();      //Initialize navigation startup variables.

	T2CON |= 0x8000; //Startup our State Period Governer (Timer 2)
	GPSI_ENABLE;     //Enable GPS Interrupts
}

//General Software Induced Delay - Milliseconds
void delay_ms(unsigned int ms)
{
	unsigned int x,a;       // Keep for counter loop
	for(x=0;x<ms;x++)
	{
		for(a=0;a<(6200);a++);//empirically tested @ 29.4912 MHz
	}
}

//General Software Induced Delay - Microseconds
void delay_us( unsigned long us)
{
	unsigned long x;        // Keep for counter loop
	for(x=0;x<us;x++)
	{
		//empirically tested @ 29.4912 MHz
		asm("clrwdt");asm("clrwdt");asm("clrwdt");asm("clrwdt");
		asm("clrwdt");asm("clrwdt");asm("clrwdt");asm("clrwdt");
		asm("clrwdt");asm("clrwdt");asm("clrwdt");asm("clrwdt");
		asm("clrwdt");asm("clrwdt");asm("clrwdt");asm("clrwdt");
	}
}

//Process the buttons considering them as each a latch.
void ProcessButtons(void)
{
    //BUTTON 1////////////////////////////////
	if(!BUTTON1 && !Button.Toggle1)
	{
	  if(RS232.AsciiSelect < 2)
	  {
		RS232.AsciiSelect++;
		//RS232.TxIndex = 0;
	  }
		Button.Toggle1 = 1;
	}
	if(BUTTON1 && Button.Toggle1)
	{
		delay_ms(1);
```

```
            Button.Toggle1 = 0;
      }

      //BUTTON 2////////////////////////////////
      if(!BUTTON2 && !Button.Toggle2)
      {
        if(RS232.AsciiSelect != 0)
            RS232.AsciiSelect--;
            Button.Toggle2 = 1;
      }
      if(BUTTON2 && Button.Toggle2)
      {
            delay_ms(1);
            Button.Toggle2 = 0;
      }
      //BUTTON 3////////////////////////////////
      if(!BUTTON3 && !Button.Toggle3) //Toggle Auto Print
      {
            //Do something
            Button.Toggle3 = 1;
      }
      if(BUTTON3 && Button.Toggle3)
      {
            delay_ms(1);
            Button.Toggle3 = 0;
      }
      //BUTTON 4////////////////////////////////
      if(!BUTTON4 && !Button.Toggle4)
      {
            //Dp something
            Button.Toggle4 = 1;
      }
      if(BUTTON4 && Button.Toggle4)
      {
            delay_ms(1);
            Button.Toggle4 = 0;
      }
}

//A function to reset and start the benchmark timer (TMR4)
void Benchmark_Start(void){
    /*IMU.Benchmark[0]=0;
    IMU.Benchmark[1]=0;
    IMU.Benchmark[2]=0;
    IMU.Benchmark[3]=0;*/
    TMR4     = 0;       //Benchmarker
    T4CON  |= 0x8000;   //Start Timer
    return;
}

//A function to record benchmarks to any 1 of 4 benchmarker variables.
void Benchmark_Stop(int Benchmark_Select){

    switch(Benchmark_Select){
        case 0: IMU.Benchmark[0] = TMR4; break;

        case 1: IMU.Benchmark[1] = TMR4; break;
```

```
        case 2: IMU.Benchmark[2] = TMR4; break;

        case 3: IMU.Benchmark[3] = TMR4; break;
      }
    if(Benchmark_Select == IMU.Benchmarker){
           T4CON        &= 0x7FFF;  //Shut off timer
           TMR4    = 0;       //Benchmarker
    }
    return;
}
```

XPIC.h

```
//XPIC.h
//Author: Kyle Howen
//Desription: Header for my particular dsPIC30F6010 setup!
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "p30f6010.h"

///////Definitions//////////////////////////////////////////////////////////
      //LEDS
#define LED1 PORTAbits.RA9
#define LED2 PORTAbits.RA10
#define LED3 PORTAbits.RA14
#define LED4 PORTAbits.RA15
      //BUTTONS
#define BUTTON1 PORTGbits.RG6
#define BUTTON2 PORTGbits.RG7
#define BUTTON3 PORTGbits.RG8
#define BUTTON4 PORTGbits.RG9
      //ADIS SPI
#define     SS1  PORTGbits.RG0
#define DIO1 PORTGbits.RG1
    //UART RS232 Communication
#define RS232TXRATE 1 //RS232 TX about once every X * 10 States
      //NAVSys
#define NAVSTATETIME 8620
#define NAVFIFO 16
#define NAV_PRESSURE_AVERAGES 9
#define T2SecondsPerTick 0.00000217013888889 //Seconds per every T2 Tick
#define T4SecondsPerTick 0.00000868055555556 //Seconds per every T4 Tick
    //PI
#define TWO_PI      6.28318531
#define PI_LOCAL    3.14159265 //From Google!
#define PI_OVER_2   1.57079633
#define PI_OVER_4   0.78539816
#define RAD2D       57.2957795
      //Scientific
#define PSEALEVEL 101.325 //kPa
#define KPA2PSI         0.145038
#define MS2MPH          2.23693629
#define MS2KTS          1.9438449
#define KTS2MPH     1.15077945
#define MPH2KTS     0.868976242
    //GPS Communication Strings
#define NMEA_OUTPUT_5HZ
      "$PMTK314,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0*29\r\n" //Set RMC to 5.0Hz
#define NMEA_OUTPUT_2p5HZ
      "$PMTK314,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0*2A\r\n" //Set RMC to 2.5Hz
#define LOCOSYS_REFRESH_RATE_250    "$PMTK220,250*29\r\n" //250 milliseconds
#define LOCOSYS_REFRESH_RATE_100    "$PMTK220,200*2C\r\n" //100 milliseconds
#define LOCOSYS_BAUD_RATE_38400     "$PMTK251,38400*27\r\n"
#define LOCOSYS_BAUD_RATE_57600     "$PMTK251,57600*2C\r\n"
#define LOCOSYS_BAUD_RATE_9600           "$PMTK251,9600*17\r\n"
#define SBAS_OFF                         "$PMTK313,0*2F\r\n"
```

```
//Intterupt
Prototypes/////////////////////////////////////////////////////////////////
void __attribute__((__interrupt__, auto_psv)) _T1Interrupt(void);
void __attribute__((__interrupt__, auto_psv)) _T2Interrupt(void);
void __attribute__((__interrupt__, auto_psv)) _ADCInterrupt(void);
void __attribute__((__interrupt__, auto_psv)) _U1TXInterrupt(void);
void __attribute__((__interrupt__, auto_psv)) _U1RXInterrupt(void);
void __attribute__((__interrupt__, auto_psv)) _U2TXInterrupt(void);
void __attribute__((__interrupt__, auto_psv)) _U2RXInterrupt(void);
void __attribute__((__interrupt__, auto_psv)) _INT1Interrupt(void);

//Global/////////////////////////////////////////////////////////////////////
      //Variables
        /*none*/
      //Structures
struct ButtonSystem
{
    int Toggle1,Toggle2,Toggle3,Toggle4;
}; extern struct ButtonSystem Button;

struct IMUSystem
{
      //SPI Variables
      int RxData, TxData, ReadAll, NewData;
      //IMU Data Variables
      int SupplyOut, XGyro, YGyro, ZGyro, XAccl, YAccl, ZAccl, XTemp, YTemp,
ZTemp, AuxADC, Diagnostic;
    //External Sensor Data
      int hAlt, ASI, VR2, Temp;
      //Failure Storage Variables
    int Benchmark[4];
    int Benchmarker;
}; extern struct IMUSystem IMU;

struct UARTSystem
{
      //TX - Variables
    char    OutputSelect,AsciiSelect; //For selection of output Data, 0-
ASCII,1-PSCAL
    int     OutputLength[10];
      int   TxComplete, TxIndex, TxStringIndex;
    char    TxBuffer[100]; //For ASCII Data
    char    TxArray[100];  //For Binary Data

    //TX - Pointers
    char    *pTime,*pLat,*pLon,*pGroundSpeed; //GPS Pointers
    char    *plAlt,*phAlt,*pASI,*pVcc;        //IMU Pointers
    char    *pAirSpeedIndicator, *pAirSpeed, *pPressure, *pAltitude;
    char    *pYaw, *pPitch, *pRoll;           //NAV Pointers

    char    *pXGyro,*pYGyro,*pZGyro;
    char    *pXAccl,*pYAccl,*pZAccl;

    char    *pBench0,*pBench1,*pBench2,*pBench3;
    char    *pIntCount,*pStatesPerGPS;
    char    *pGPSLOST,*pGPSValidData;
```

```
    char    *pPressureHA,*pPressureLA,*pPressureD,*pAltitudeH,*pAltitudeL;
    char    *pTemp, *pASI_Rho;

    char    *pKp,*pKi,*pWeightYaw,*pWeightRP,*pdeltaGain;

    //RX
      int         RxIndex,Lock,NewData;
      char  RxBuffer[3],cBuffer;
}; extern struct UARTSystem RS232;

struct GPSSystem
{
      //Flight GPS Variables
      float  UTCTime, Latitude, Longitude, GroundSpeed, Course;
      int    NewGroundSpeed;
      char  ValidData, North, East, Date[7];
      //UART NMEA Read Variables
      char  TxBuffer[100];                              //Transmit
      int   TxComplete, TxIndex, TxChecksum;
      char  Buffer[100], cBuffer;               //Receive
      int   Index, Lock, ChecksumFlag, ReadChecksum, CalcChecksum,
ParseNumber, NewData;
    int     AllowRS232;
}; extern struct GPSSystem GPS;

struct NAVSystem
{
    //Measured + Filter Corrected Angular Velocities that will update the DCM
    double  WX,WY,WZ;
    //Gyro Offset Bias (AVERAGED INTEGER DATA) (not in rad/s)
    double  WXOff, WYOff, WZOff;
    //Yaw Correction Vector & Filter Weight
    double  YawXCorr,YawYCorr,YawZCorr, WeightYaw;
    //Roll+Pitch Correction Vector & Filter Weight
    double  RPXCorr, RPYCorr, RPZCorr, WeightRP;
    //Total Correction
    double  TotalCorrX, TotalCorrY, TotalCorrZ;
    //Proprotional Filter + Weights
    double  WPCorrX, WPCorrY, WPCorrZ, Kp;
    //Integral Filter + Weights
    double  WICorrX, WICorrY, WICorrZ, Ki;
    //Final Gyro Correction Vector
    double  WXCorr, WYCorr, WZCorr;

    double  deltaGain; //Used to increment the gains from RS232 Console

    int     GroundSpeedNotASI, GroundCourseValid; //GPS Navigation Data Flags

    //Airspeed and xb-Acceleration
    double  AirSpeed, PastAirSpeed, AirAcceleration, AirAccelerationTime;

    double Yaw,Pitch,Roll; //Cardinal Rotations
    double sinYaw, sinPitch, sinRoll; //for repetitive use in calcs
    double cosYaw, cosPitch, cosRoll; //for repetitive use in calcs
    double XAccl, YAccl, ZAccl;

    double DCM[3][3];   //Direction Cosine Matrix
```

```
    double oDCM[3][3];  //Old (Last State's) DCM Carbon copy
    double dtX,dtY,dtZ; //Delta Theta in X Y Z rotations

    double aIMUADC[NAVFIFO], aIMUVCC[NAVFIFO], aIMUH2V[NAVFIFO],
                  aIMUD2V[NAVFIFO]; //Arrays to produce rolling averages
    double IMUADC, IMUVCC, IMUH2V, IMUD2V; //Rolling averages
    double iIMUADC, iIMUVCC, iIMUH2V,iIMUD2V; //Initialization Variables
    double iIMUTEMP;

    double PressureDOFF;//Differential Pressure Offset
    double LowAltGain;  //Adjusted Gain calibrated from High Altimeter
    double Temp;        //Cabin Temperature
    double ASI_Rho;     //Air Density Estimate
    double TakeOffAlt;  //Initial Altitude, (Autorecorded at Startup)

    //Pressure measured from Low Altitude Pressure Sensor (the IMU's AUX_ADC)
    double PressureLA;

    //Pressure measured from HighAltitude Pressure Sensor (picADC)
    double     PressureHA;

    //Pressure measured from Differential Pressure Sensor (picADC)
    double PressureD;

    double Pressure;

    //Airspeed derived from Differential Pressure Sensor
    double AirSpeedIndicator;
    //Altitude derived from Low Altitude Pressure Sensor
    double AltitudeL;
    //Altitude derived from High Altitude Pressure Sensor
    double AltitudeH;
    double Altitude;

    double StateTime; //Clock ticks converted to seconds
    unsigned int  NewData, DataAge; //State Ages
    unsigned int  CriticalTask, IntCount; //Mainloop State Progress Flags

    //Used to detect and mark State Cycles per GPS Transmission
    unsigned int  GPSValidData, StatesPerGPS, StateCounter;

    int GPSLOST; //Flags to time RS232...
    int GPSDONE; //...Output with GPS

    double dIndex;//Circular Indexing for Averaging Filter
    unsigned int  Index, OIA[NAVFIFO]; //Ordered Index Array
}; extern struct NAVSystem NAV;

//Function
Prototypes////////////////////////////////////////////////////////////
      //Program Functions
void HardwareInit(void);
void ProcessButtons(void);

      //Time Functions (Empirically tested for 29.4912MHz)
void delay_us( unsigned long);
void delay_ms(unsigned int);
```

```
void Benchmark_Start(void);
void Benchmark_Stop(int);

      //UART Functions
void UART_Init(void);
int  RS232_Update(void);
int  RS232_TX(void);
int  GPS_Update(int);
int  GPS_TX(int);
#define T2_START        T2CON  |= 0x8000;
#define T2_STOP         T2CON  &= 0x7FFF;
#define GPSI_ENABLE     IEC1  |= 0x0100
#define GPSI_DISABLE    IEC1  &= 0xFEFF
#define URXI_ENABLE     IEC0  |= 0x0200;
#define URXI_DISABLE    IEC0  &= 0xFDFF;

      //ADIS IMU & SPI Functions
void IMU_Init(void);
void IMU_Update(void);
void SPI1_TX(void);
void SPI1_TXRX(void);
void SignExtend(int* , int );

      //NavSys Functions
void              NAV_Init(void);
void            NAV_InitialValues(void);
void            NAV_Trig(void);
void            NAV_Sensors(double);
void            NAV_EditGains(double,double,double,double);

void            NAV_DCM(void);
void            NAV_DCM_Decode(void);
void            NAV_DCM_Copy(int);
void            NAV_DCM_WCorrection(double);
void            NAV_DCM_Update(double);
void            NAV_DCM_OrthoNorm(void);

void              NAV_IncrementIndex(void);
unsigned int      NAV_DisplaceIndex(int);
void              NAV_MakeOIA(void);
```

ISR.c

```
//ISR.c
//Author: Kyle Howen - 03-08-2010
//Desription: This houses all the Interrupt Service Routines.
#include "XPIC.h"

//Timer 1 - Interrupt is not yet used.
void _T1Interrupt(void)
{
    IFS0    &= 0xFFF7;        //Interrupt flag reset
}

//Timer 2 - The State-Time Interval Controller (20ms)
void _T2Interrupt(void)
{
    IFS0    &= 0xFFBF;     //Interrupt Flag Reset

    NAV.DataAge++;
    if(GPS.Lock == 0){
    //If ~19.4ms has passed this state (2.1701388889 us per tick)
        LED2 ^= 1;

        NAV.CriticalTask = 1;
        ADCON1bits.SAMP = 1;    //Start ADC Sample
        IMU_Update(); //Alledgedly this function can be interrupted
        NAV.CriticalTask = 0;

        if(IMU.NewData){
            //Gather ADC Values since enough time should have passed!
            IMU.VR2  = (ADCBUF0 + ADCBUF4 + ADCBUF8 + ADCBUFC) / 4;
            IMU.ASI  = (ADCBUF2 + ADCBUF6 + ADCBUFA + ADCBUFE) / 4;
            IMU.hAlt = (ADCBUF1 + ADCBUF5 + ADCBUF9 + ADCBUFD) / 4;
            IMU.Temp = (ADCBUF3 + ADCBUF7 + ADCBUFB + ADCBUFF) / 4; //AN5

            NAV.NewData = 1;
        }

        if(NAV.StateCounter > 41 && NAV.GPSLOST == 0)
            NAV.GPSLOST = 1;

        NAV.StateCounter++;

    }
}

//ADC - Interrupt is not yet used.
void _ADCInterrupt(void)
{
    IFS0    &= 0xF7FF; //Reset IFlag
}

//UART1 TX - This manages RS232 Transmissions to the Ground Station
void _U1TXInterrupt(void){
    IFS0 &= 0xFBFF;
    while(!(U1STA & 0x0200) && (!RS232.TxComplete))
    //while UT1BFull flag is low & transmit in progress fill the Fifo!{
```

```
    {
        if(RS232.OutputSelect == 0){ //ASCII Transmissions
            if( (RS232.TxBuffer[RS232.TxStringIndex] == '\0') ||
                (RS232.TxStringIndex > 99) ){ //If end of Transmission
                    int i;
                   //Prep the variables for new setence
                    for(i = 0; i < 100; i++)

                          RS232.TxBuffer[i] = 0;
                RS232.TxStringIndex = 0;
                    RS232.TxComplete = 1;
                    break;
              }
              else{ //else load Fifo + Increment
                    U1TXREG = RS232.TxBuffer[RS232.TxStringIndex];

                    RS232.TxStringIndex++;
              }
        }
        else{                       //Binary Transmissions
            U1TXREG = RS232.TxArray[RS232.TxIndex];
            if(RS232.TxIndex >
                      RS232.OutputLength[(int)RS232.OutputSelect]-2){
            //If last word of transmission
                    RS232.TxIndex = 0;
                    RS232.TxComplete = 1;
                    break;
              }
            else
                  RS232.TxIndex++;
        }
    }
}

//UART1 RX - This manages RS232 console commands recieved from the Ground
Station
void _U1RXInterrupt(void)//RS232 Rx Interrupt
{
    IFS0 &= 0xFDFF; //Clear UART1 Rx Flag
    //OVERRUN ERROR HANDLING
    if(U1STA & 0x0002){
        LED4 = 1;
            U1STA &= 0xFFFD;
            RS232.Lock = 0;
      }
    //OBTAINING LOCK
      if(!RS232.Lock){
            while(U1STA & 0x0001){ //RxData Available
                  RS232.cBuffer = U1RXREG;//Hot off the FIFO
                  if(RS232.cBuffer == '$'){
                RS232.cBuffer = 0;
                        RS232.RxBuffer[0] = 0;
                RS232.RxBuffer[1] = 0;
                RS232.RxBuffer[2] = 0;
                        RS232.Lock = 1;
                RS232.RxIndex = 0;
                RS232.NewData = 0;
```

```
                        break;
                    }
                }
            }
        //ASSEMBLING 3-Character COMMAND
            if(RS232.Lock){ //Incoming Terminal Data Locked In
              while((U1STA & 0x0001) && (RS232.Lock)){
              //While Data available and locked on!
                        RS232.RxBuffer[RS232.RxIndex] = U1RXREG;//Hot off the FIFO
                    if(RS232.RxIndex == 2){
                        RS232.Lock = 0;
                            RS232.NewData = 1;
                        break;
                    }
                      else
                            RS232.RxIndex++;
                    }
            }
}

//UART2 TX - This manages NMEA sentence transmissions to the LS20031 GPS
//           Unit.
void _U2TXInterrupt(void)//UART2 Tx Interrupt (GPS)
{
      IFS1 &= 0xFDFF;
      while(!(U2STA & 0x0200) && (!GPS.TxComplete))
      //while UT1BFull flag is low & transmit in progress fill the Fifo!{
      {
            if( (GPS.TxBuffer[GPS.TxIndex] == '\0') || (GPS.TxIndex > 99) )
             //If end of Transmission
            {
                  int i;
                  //Prep the variables for new setence
                  for(i = 0; i < 100; i++)
                        GPS.TxBuffer[i] = 0;
                  GPS.TxIndex = 0;
                  GPS.TxComplete = 1;
                  break;
            }
            else
            { //else load Fifo + Increment
                  U2TXREG = GPS.TxBuffer[GPS.TxIndex];
                  GPS.TxIndex++;
            }
      }
}

//UART2 RX - This assembles NMEA sentences and runs checksums to validate
data as it is recieved from the LS20031 GPS unit.
void _U2RXInterrupt(void)
{
      IFS1  &= 0xFEFF; //Clear UART2 RX IFlag

    if((NAV.CriticalTask == 1))
    //If we interrupted a critical IMU/ADC Measurement
    {
        LED3 ^= 1;
```

```
    NAV.IntCount++;
}
if(U2STA & 0x0002) //If Overflow Error
  {
    U2STA &= 0xFFFD; //Clear flag, resulting in FIFO loss
        GPS.Lock = 0; //Disengage Lock, GPS.Buffer = Useless Data
    GPS.NewData = 0;
  }
  int i;
  if(!GPS.Lock) //Scan for $ to obtain lock
  {
        while(U2STA & 0x0001) //While URXDAvailable flag is asserted
        {
              GPS.cBuffer = U2RXREG; //Hot off the FIFO
              if(GPS.cBuffer  == '$')
              {   //Prep the variables for new setence
                    for(i = 0; i < 100; i++)
                    GPS.Buffer[i] = 0;
                    GPS.Lock = 1; GPS.Index = 1; GPS.ChecksumFlag = 0;
                    GPS.NewData = 0;
                    GPS.Buffer[0] = '$'; GPS.cBuffer = 0;
                    GPS.ReadChecksum = 0; GPS.CalcChecksum = 0;
                    GPS.AllowRS232++;
                    break;
                    //get out of here and initiate lock-on sequence below
              }
        }
  }
  if(GPS.Lock) //NMEA Sentence Locked on!
  {
        while((U2STA & 0x0001) && (GPS.Lock))
        //While URXDAvailable flag is asserted & Lock remains
        {
              GPS.Buffer[GPS.Index] = U2RXREG;//Hot off the FIFO

              if(GPS.Buffer[GPS.Index] == '*')
                    GPS.ChecksumFlag = 1;
              else
              {
                    switch(GPS.ChecksumFlag)
                    {
                          case 2: //Checksum Found, Second Step
                                //null terminate the string
                              GPS.Buffer[GPS.Index+1] = 0;
                                //read the checksum
           sscanf(&GPS.Buffer[GPS.Index-1],"%2x",&GPS.ReadChecksum);
                          GPS.Lock = 0; GPS.NewData = 1;
                          NAV.StatesPerGPS = NAV.StateCounter;
                          NAV.StateCounter = 0; NAV.GPSLOST = 0;
                          break;

                          case 1: //Checksum Found, First Step
                          //Flag the upcoming second digit

                          GPS.ChecksumFlag = 2;
                          break;
```

```
                                case 0: //Pre-Checksum Data
                                GPS.CalcChecksum ^= GPS.Buffer[GPS.Index];
                                //Continue calculating our own checksum
                                break;
                        }
                }
                if(GPS.Index < 100)
                        //Safely increment our index for next loop
                        GPS.Index++;
                else
                        GPS.Lock = 0;
        }
    }
}

//External Interrupt 1 -    The ADIS16365-IMU Data-Ready line is connected to
//                          the INT1 pin, however this ISR is not yet used.
void _INT1Interrupt(void)
{
    IFS1 &= 0xFFFE; //Clear Interrupt Flag
}
```

NAV.c

```
//NAV.c
//Author: Kyle Howen - 03-08-2010
//Desription: This houses all the functions that operate primarily
//            on the NAV Structure data to ultimately provide a Flight Model.
//Note: A lot of the theory behind the code in this document comes from the
//      following sources:
//          Direction Cosine Matrix IMU:
//                Theory By William Premerlani and Paul Bizard
//          gentlenav.googlecode.com/files/DCMDraft2.pdf
//
//          Pitot Tube Theory By NASA
//          www.grc.nasa.gov/WWW/K-12/airplane/pitot.html
//
//          Pressure/Altitude Relationship Theory By GSU
//          hyperphysics.phy-astr.gsu.edu/hbase/Kinetic/barfor.html

#include "XPIC.h"

//Local Definitions
#define SINROLL         NAV.sinRoll
#define SINPITCH        NAV.sinPitch
#define SINYAW          NAV.sinYaw
#define COSROLL         NAV.cosRoll
#define COSPITCH        NAV.cosPitch
#define COSYAW          NAV.cosYaw

#define GYROTHRESH      0.005
#define ACCLTHRESH      0.1

//NAV_Init() initializes all the navigation data to 0 and calculates Gyro
//offsets.
void NAV_Init(void)
{
    T2_STOP;      //Startup our State Period Governer (Timer 2)
    GPSI_DISABLE; //Enable GPS Interrupt

    NAV.Roll = 0;
    NAV.Pitch = 0;
    NAV.Yaw = 0;

    NAV_Trig();

    NAV_DCM();
    NAV_DCM_Copy(0);

    NAV_InitialValues();

    NAV.GroundSpeedNotASI = 1;

    NAV.WeightYaw   = 0.1;
    NAV.WeightRP    = 0.10204;
    NAV.Kp          = 1.0;
    NAV.Ki          = 0.05;

    T2_START; //Startup our State Period Governer (Timer 2)
```

```
    GPSI_ENABLE; //Enable GPS Interrupt
}

//NAV_Sensors(double) takes the sensor data and formulates navigation values
//from them in SI units.
void NAV_Sensors(double dTime)
{
    //Moving Averages
    NAV.aIMUADC[NAV.Index] = (double)(IMU.AuxADC) * 0.81;
    NAV.aIMUVCC[NAV.Index] = (double)(IMU.SupplyOut) * 2.42;
    NAV.aIMUH2V[NAV.Index] = (double)IMU.hAlt/1023;
    NAV.aIMUD2V[NAV.Index] = (double)IMU.ASI /1023;
    NAV_IncrementIndex();

    int i;
    NAV.IMUADC = 0; NAV.IMUVCC = 0; NAV.IMUH2V = 0; NAV.IMUD2V = 0;
    for(i = 0; i < NAVFIFO; i++){
        NAV.IMUADC += NAV.aIMUADC[i];
        NAV.IMUD2V += NAV.aIMUD2V[i];
        NAV.IMUH2V += NAV.aIMUH2V[i];
        NAV.IMUVCC += NAV.aIMUVCC[i];
    }
    NAV.IMUADC /= NAVFIFO; NAV.IMUVCC /= NAVFIFO; NAV.IMUH2V /= NAVFIFO;
      NAV.IMUD2V /= NAVFIFO;

    //Temperature
    NAV.Temp = 295.372222 - (NAV.IMUVCC *
                             (double)(IMU.Temp-441))/(1023.0 * 2.5 * 4);

    //Pressures and Altitudes: kPa and meters
    NAV.PressureHA = (NAV.IMUH2V + 0.095)/0.009;
    NAV.AltitudeH = -7935*log(NAV.PressureHA/PSEALEVEL);
    if(NAV.PressureHA > 75.0){
        NAV.PressureLA = 121.66666667 - (NAV.IMUADC/NAV.IMUVCC) *
                                              (111.1111111/NAV.LowAltGain);
        NAV.AltitudeL = -7935*log(NAV.PressureLA/PSEALEVEL);
        NAV.Pressure = NAV.PressureLA;
        NAV.Altitude = NAV.AltitudeL;
    }
    else{
        NAV.PressureLA = 0;
        NAV.AltitudeL = 0;
        NAV.Pressure = NAV.PressureHA;
        NAV.Altitude = NAV.AltitudeH;
    }
    NAV.PressureD = ((NAV.IMUD2V - 0.2)/0.2) - NAV.PressureDOFF;
    if(NAV.PressureD < 0)
            NAV.PressureD = 0;

    //Airspeed & Velocity & Air Acceleration Selection
    NAV.AirSpeedIndicator = 1.9438449 * sqrt(2000*NAV.PressureD/NAV.ASI_Rho);
    if(NAV.GPSLOST){    //NO GPS DATA AVAILALBE//
        NAV.GroundCourseValid = 0;
        NAV.PastAirSpeed = NAV.AirSpeed;
        NAV.AirSpeed = NAV.AirSpeedIndicator;
        NAV.AirAcceleration = (NAV.AirSpeed - NAV.PastAirSpeed)
                                        / (1.9438449 * dTime); //in m/s^2
```

```
    }else{              //GPS DATA IS AVAILABLE//
        if(NAV.GroundSpeedNotASI){
            if((NAV.Pitch > 1.04719755)||(NAV.Pitch < -1.04719755))
                NAV.AirSpeed = NAV.AirSpeedIndicator;
            else
                NAV.AirSpeed = GPS.GroundSpeed / NAV.cosPitch;

            NAV.AirAccelerationTime += dTime;
            if(GPS.NewGroundSpeed == 1){
                NAV.AirAcceleration = (NAV.AirSpeed - NAV.PastAirSpeed)
                         / (1.9438449 * NAV.AirAccelerationTime); //in m/s^2
                NAV.PastAirSpeed = NAV.AirSpeed;
                NAV.AirAccelerationTime = 0;
                GPS.NewGroundSpeed = 0;
            }
        }
        else{
            NAV.PastAirSpeed = NAV.AirSpeed;
            NAV.AirSpeed = NAV.AirSpeedIndicator;
            NAV.AirAcceleration = (NAV.AirSpeed - NAV.PastAirSpeed)
                                        / (1.9438449 * dTime); //in m/s^2
        }
    }

    //Airspeed Threshhold
    if(NAV.AirSpeed<3){ //Less than 3 m/s
        NAV.AirSpeed = 0;
        NAV.GroundCourseValid = 0;
    }
    else if(!NAV.GPSLOST)
        NAV.GroundCourseValid = 1;

    //If ground course not valid then clear the integral filter (it can grow
    //to over 100 and take forever to clear up)
    if(NAV.GroundCourseValid == 0)
    {
        NAV.WICorrX = 0;
        NAV.WICorrY = 0;
        NAV.WICorrZ = 0;
    }
    //Measured Angular Velocities
    NAV.WX = ((double)(IMU.XGyro)-NAV.WXOff) * 0.00087264625;
    NAV.WY = ((double)(IMU.YGyro)-NAV.WYOff) * -0.00087264625;
    NAV.WZ = ((double)(IMU.ZGyro)-NAV.WZOff) * -0.00087264625;
                            //degrees/s * (PI/180) * deltaT = Rad

    //Measured Body Frame Accelerations
    NAV.XAccl = (double)(IMU.XAccl) * -0.032361945; //      m/s^2
      NAV.YAccl = (double)(IMU.YAccl) * 0.032361945;
            //   (Accel_ticks * (3.3E-3)g/tick * 9.80665 m/ g*s^2)
      NAV.ZAccl = (double)(IMU.ZAccl) * 0.032361945;
}

//NAV_DCM() Calculates the nav Direction Cosine Matrix from attitude
//rotations.
void NAV_DCM(void)
{
```

```
      NAV.DCM[0][0] = COSPITCH*COSYAW;
      NAV.DCM[0][1] = SINROLL*SINPITCH*COSYAW-COSROLL*SINYAW;
      NAV.DCM[0][2] = COSROLL*SINPITCH*COSYAW+SINROLL*SINYAW;
      NAV.DCM[1][0] = COSPITCH*SINYAW;
      NAV.DCM[1][1] = SINROLL*SINPITCH*SINYAW+COSROLL*COSYAW;
      NAV.DCM[1][2] = COSROLL*SINPITCH*SINYAW-SINROLL*COSYAW;
      NAV.DCM[2][0] = (-1)*SINPITCH;
      NAV.DCM[2][1] = SINROLL*COSPITCH;
      NAV.DCM[2][2] = COSROLL*COSPITCH;
}

//NAV_DCM_Decode() decodes attitude rotations from the nav DCM.
void NAV_DCM_Decode(void)
{
    NAV.Pitch = -asin(NAV.DCM[2][0]);
    NAV.cosPitch = cos(NAV.Pitch);
    NAV.Roll  = asin(NAV.DCM[2][1]/NAV.cosPitch);
    //NAV.Yaw = asin(NAV.DCM[1][0]/(NAV.cosPitch));
    NAV.Yaw = atan(NAV.DCM[1][0]/NAV.DCM[0][0]);

    //Roll Circulation
    if(NAV.DCM[2][2] < 0){ //Up Side Down
        if(NAV.Roll  < 0)
            NAV.Roll = -PI_LOCAL - NAV.Roll;
        else
            NAV.Roll =  PI_LOCAL - NAV.Roll;
    }
    //Yaw Circulation
    if(NAV.DCM[0][0] < 0){ //Pointed South
        if(NAV.Yaw <0)
            NAV.Yaw = PI_LOCAL + NAV.Yaw;
        else
            NAV.Yaw = -PI_LOCAL + NAV.Yaw;
    }
}

//NAV_DCM_WCorrection(double) calculates a correction rotation then feeds
this correction through a PI controller.
void NAV_DCM_WCorrection(double deltaT)
{
    //Ground Course Based Correction (Yaw)
    if(NAV.GroundCourseValid){
        double Cx,Cy, Yx,Yy;
        double YErrorE;
        Cx = cos(GPS.Course/RAD2D);
        Cy = sin(GPS.Course/RAD2D);
        Yx = cos(NAV.Yaw);
        Yy = sin(NAV.Yaw);
        YErrorE = asin(Cy * Yx - Cx * Yy);
        //YErrorE = GPS.Course/RAD2D - NAV.Yaw;
        NAV.YawXCorr = YErrorE * NAV.DCM[2][0];
        NAV.YawYCorr = YErrorE * NAV.DCM[2][1];
        NAV.YawZCorr = YErrorE * NAV.DCM[2][2];
    }
    else{
        NAV.YawXCorr = 0;
        NAV.YawYCorr = 0;
```

```
        NAV.YawZCorr = 0;
    }

    //Accelerometer Based Correction (Roll / Pitch)
    double GXRef, GYRef, GZRef;
    GXRef = NAV.XAccl - NAV.AirAcceleration; //This, and d(NAV.AirSpeed)/d(t)
                             //... however we may want to ignore this;
    GYRef = NAV.YAccl - NAV.AirSpeed * NAV.WZ / 1.9438449; //m/s^2
    GZRef = NAV.ZAccl - NAV.AirSpeed * NAV.WY / 1.9438449; //m/s^2

    double normalize;
    normalize = sqrt(GXRef*GXRef+GYRef*GYRef+GZRef*GZRef);
    GXRef /= normalize; GYRef /= normalize; GZRef /= normalize;


    NAV.RPXCorr = asin(NAV.DCM[2][1] * GZRef - NAV.DCM[2][2] * GYRef);
    NAV.RPYCorr = asin(NAV.DCM[2][2] * GXRef - NAV.DCM[2][0] * GZRef);
    NAV.RPZCorr = asin(NAV.DCM[2][0] * GYRef - NAV.DCM[2][1] * GXRef);

    //Total Correction Vector
    NAV.TotalCorrX = NAV.WeightRP * NAV.RPXCorr
                             + NAV.WeightYaw * NAV.YawXCorr;
    NAV.TotalCorrY = NAV.WeightRP * NAV.RPYCorr
                             + NAV.WeightYaw * NAV.YawYCorr;
    NAV.TotalCorrZ = NAV.WeightRP * NAV.RPYCorr
                             + NAV.WeightYaw * NAV.YawZCorr;

    //Proportional Control
    NAV.WPCorrX = NAV.Kp * NAV.TotalCorrX;
    NAV.WPCorrY = NAV.Kp * NAV.TotalCorrY;
    NAV.WPCorrZ = NAV.Kp * NAV.TotalCorrZ;

    //Integral Control
    NAV.WICorrX += NAV.Ki * deltaT * NAV.TotalCorrX;
    NAV.WICorrY += NAV.Ki * deltaT * NAV.TotalCorrY;
    NAV.WICorrZ += NAV.Ki * deltaT * NAV.TotalCorrZ;

    //Combined PI Control Error Signal
    NAV.WXCorr = NAV.WPCorrX + NAV.WICorrX;
    NAV.WYCorr = NAV.WPCorrY + NAV.WICorrY;
    NAV.WZCorr = NAV.WPCorrZ + NAV.WICorrZ;
}

//NAV_DCM_Update(double) rotates a DCM based on small short-term gyro
//measurements
void NAV_DCM_Update(double deltaT)
{
    //Angular Displacements + Correction
    NAV.dtX = (NAV.WX + NAV.WXCorr) * deltaT;
    NAV.dtY = (NAV.WY + NAV.WYCorr) * deltaT;
    NAV.dtZ = (NAV.WZ + NAV.WZCorr) * deltaT;

    NAV.DCM[0][0] = NAV.oDCM[0][0]
                  - NAV.oDCM[0][2]*NAV.dtY + NAV.oDCM[0][1]*NAV.dtZ;
    NAV.DCM[0][1] = NAV.oDCM[0][1]
                  - NAV.oDCM[0][0]*NAV.dtZ + NAV.oDCM[0][2]*NAV.dtX;
    NAV.DCM[0][2] = NAV.oDCM[0][2]
```

```
                    - NAV.oDCM[0][1]*NAV.dtX + NAV.oDCM[0][0]*NAV.dtY;
    NAV.DCM[1][0] = NAV.oDCM[1][0]
                    - NAV.oDCM[1][2]*NAV.dtY + NAV.oDCM[1][1]*NAV.dtZ;
    NAV.DCM[1][1] = NAV.oDCM[1][1]
                    - NAV.oDCM[1][0]*NAV.dtZ + NAV.oDCM[1][2]*NAV.dtX;
    NAV.DCM[1][2] = NAV.oDCM[1][2]
                    - NAV.oDCM[1][1]*NAV.dtX + NAV.oDCM[1][0]*NAV.dtY;
    NAV.DCM[2][0] = NAV.oDCM[2][0]
                    - NAV.oDCM[2][2]*NAV.dtY + NAV.oDCM[2][1]*NAV.dtZ;
    NAV.DCM[2][1] = NAV.oDCM[2][1]
                    - NAV.oDCM[2][0]*NAV.dtZ + NAV.oDCM[2][2]*NAV.dtX;
    NAV.DCM[2][2] = NAV.oDCM[2][2]
                    - NAV.oDCM[2][1]*NAV.dtX + NAV.oDCM[2][0]*NAV.dtY;
}

//NAV_DCM_OrthoNorm() normalizes and orthogonalizes the rows of the DCM
//matrix
void NAV_DCM_OrthoNorm(void)
{
    double half_error = ( NAV.DCM[0][0]*NAV.DCM[1][0] +
      NAV.DCM[0][1]*NAV.DCM[1][1] + NAV.DCM[0][2]*NAV.DCM[1][2] ) / 2;
    NAV_DCM_Copy(1); //No Z Copy

    //ORTHO
      NAV.DCM[0][0] -= half_error*NAV.oDCM[1][0];
      NAV.DCM[1][0] -= half_error*NAV.oDCM[0][0];
      NAV.DCM[2][0] = NAV.DCM[0][1]*NAV.DCM[1][2]
                                   - NAV.DCM[1][1]*NAV.DCM[0][2];
      NAV.DCM[0][1] -= half_error*NAV.oDCM[1][1];
      NAV.DCM[1][1] -= half_error*NAV.oDCM[0][1];
      NAV.DCM[2][1] = NAV.DCM[0][2]*NAV.DCM[1][0]
                                   -  NAV.DCM[1][2]*NAV.DCM[0][0];
      NAV.DCM[0][2] -= half_error*NAV.oDCM[1][2];
      NAV.DCM[1][2] -= half_error*NAV.oDCM[0][2];
      NAV.DCM[2][2] = NAV.DCM[0][0]*NAV.DCM[1][1]
                                   - NAV.DCM[1][0]*NAV.DCM[0][1];

    double normalize;
    //XNORM
    normalize = (3 -(NAV.DCM[0][0]*NAV.DCM[0][0]+NAV.DCM[0][1]*NAV.DCM[0][1]+
                     NAV.DCM[0][2]*NAV.DCM[0][2]))/2;
    NAV.DCM[0][0] *= normalize;
    NAV.DCM[0][1] *= normalize;
    NAV.DCM[0][2] *= normalize;

    //YNORM
    normalize = (3 -(NAV.DCM[1][0]*NAV.DCM[1][0]+NAV.DCM[1][1]*NAV.DCM[1][1]+
                     NAV.DCM[1][2]*NAV.DCM[1][2]))/2;
    NAV.DCM[1][0] *= normalize;
    NAV.DCM[1][1] *= normalize;
    NAV.DCM[1][2] *= normalize;

    //ZNORM
    normalize = (3 -(NAV.DCM[2][0]*NAV.DCM[2][0]+NAV.DCM[2][1]*NAV.DCM[2][1]+
                     NAV.DCM[2][2]*NAV.DCM[2][2]))/2;
    NAV.DCM[2][0] *= normalize;
    NAV.DCM[2][1] *= normalize;
```

```
    NAV.DCM[2][2] *= normalize;
}

//NAV_DCM_Copy(int) copies the DCM to an older DCM for calculation purposes.
void NAV_DCM_Copy(int NoZ)
{
      NAV.oDCM[0][0] = NAV.DCM[0][0];
      NAV.oDCM[0][1] = NAV.DCM[0][1];
      NAV.oDCM[0][2] = NAV.DCM[0][2];
      NAV.oDCM[1][0] = NAV.DCM[1][0];
      NAV.oDCM[1][1] = NAV.DCM[1][1];
      NAV.oDCM[1][2] = NAV.DCM[1][2];
    if(NoZ)
        return;
      NAV.oDCM[2][0] = NAV.DCM[2][0];
      NAV.oDCM[2][1] = NAV.DCM[2][1];
      NAV.oDCM[2][2] = NAV.DCM[2][2];
}

//NAV_Trig() simply calculates the sines and cosines of the attitude angles
//and stores the results.
void NAV_Trig(void)
{
      SINROLL = sin(NAV.Roll);
      SINPITCH = sin(NAV.Pitch);
      SINYAW = sin(NAV.Yaw);
      COSROLL = cos(NAV.Roll);
      COSPITCH = cos(NAV.Pitch);
      COSYAW = cos(NAV.Yaw);
}

//NAV_InitialValues() solves for and sets the appropriate initial navigation
variables.
void NAV_InitialValues(void)
{
    int averages; double daverages;
    NAV.WXOff = 0; NAV.WYOff = 0; NAV.WZOff = 0;
    NAV.iIMUADC = 0; NAV.iIMUH2V = 0; NAV.iIMUVCC = 0;
    NAV.iIMUD2V = 0; NAV.iIMUTEMP = 0;
    for(averages = 1; averages < 100; averages++) //Chose 100 arbitrarily! :)
    {
        daverages = (double)averages;
        //Gather Data
        ADCON1bits.SAMP = 1;
        IMU_Update();
        IMU.VR2  = (ADCBUF0 + ADCBUF4 + ADCBUF8 + ADCBUFC) / 4;
        IMU.ASI  = (ADCBUF2 + ADCBUF6 + ADCBUFA + ADCBUFE) / 4;
        IMU.hAlt = (ADCBUF1 + ADCBUF5 + ADCBUF9 + ADCBUFD) / 4;
        IMU.Temp = (ADCBUF3 + ADCBUF7 + ADCBUFB + ADCBUFF) / 4;

        //Average Data
        NAV.iIMUVCC = ((double)IMU.SupplyOut/daverages)+((daverages-1)*
                  NAV.iIMUVCC/ daverages);
        NAV.iIMUH2V = ((double)IMU.hAlt/daverages)  + ( (daverages-1) *
                  NAV.iIMUH2V/ daverages);
        NAV.iIMUADC = ((double)IMU.AuxADC/daverages)+ ( (daverages-1) *
                  NAV.iIMUADC/ daverages);
```

```
        NAV.iIMUD2V  = ((double)IMU.ASI/daverages)   + ( (daverages-1) *
                  NAV.iIMUD2V / daverages);
        NAV.iIMUTEMP = ((double)IMU.Temp/daverages) + ( (daverages-1) *
                  NAV.iIMUTEMP / daverages);
        NAV.WXOff   = ((double)IMU.XGyro/daverages) + ( (daverages-1) *
                  NAV.WXOff  / daverages);
        NAV.WYOff   = ((double)IMU.YGyro/daverages) + ( (daverages-1) *
                  NAV.WYOff  / daverages);
        NAV.WZOff   = ((double)IMU.ZGyro/daverages) + ( (daverages-1) *
                  NAV.WZOff  / daverages);

    }

    NAV.iIMUADC *= 0.81; //mV
    NAV.iIMUVCC *= 2.42; //mV
    NAV.PressureHA = (NAV.iIMUH2V/1023 + 0.095)/0.009; //kPa

    //Low Altimeter Gain Adjustment & Limitting
    double denominator = NAV.iIMUVCC * 0.009
                  * (121.66666667 - NAV.PressureHA);
    NAV.LowAltGain = NAV.iIMUADC / denominator; //(V/V)
    if(NAV.LowAltGain < 1.4 || 1.6 < NAV.LowAltGain)
        NAV.LowAltGain = 1.5;

    //Differential Pressure Offset Calculation
    NAV.PressureDOFF = (NAV.iIMUD2V/1023 - 0.2)/0.2; //kPa

    //RHO CALCULATION
    NAV.Temp =      295.372222 - (NAV.iIMUVCC
                        * (double)(IMU.Temp-441))/(1023.0 * 2.5 * 3);
    NAV.ASI_Rho = (NAV.PressureHA * 1000)/(287.05 * NAV.Temp);
    if(NAV.ASI_Rho == 0)
        NAV.ASI_Rho = (90.0 * 1000.0)/(287.05 * 292.0); // 90kPa, 292K

    //Take off Altitude
    NAV.TakeOffAlt = -7935*log(NAV.PressureHA/PSEALEVEL);
}

//Nav_EditGains(double,double,double,double) is used to change the PI
//Controller gain values.
void NAV_EditGains(double lWeightYaw, double lWeightRP,
                        double lKp, double lKi)
{
    NAV.WeightYaw   += lWeightYaw;
    NAV.WeightRP    += lWeightRP;
    NAV.Kp          += lKp;
    NAV.Ki          += lKi;
}

//CIRCULAR BUFFER/ FIFO FUNCTIONS//////////////////////////////////
void NAV_IncrementIndex(void) //Increment the circular index
{
      if(NAV.Index > (NAVFIFO-2))
            NAV.Index = 0;
      else
            NAV.Index++;
}
```

```
unsigned int NAV_DisplaceIndex(int disp) //Displace the index by +/- integer
{
      disp += NAV.Index;
      while(disp > (NAVFIFO - 1)) //Correct for Positive Overrun
            disp -= NAVFIFO;
      while(disp < 0)               //Correct for Negative Overrun
            disp += NAVFIFO;
      return disp;
}
void NAV_MakeOIA(void) //Make an Ordered Index Array
{
      int i;
      for(i = 0; i > (-1*NAVFIFO); i--)
            NAV.OIA[i] = NAV_DisplaceIndex(i);
}
////////////////////////////////////////////////////////////////////
```

UART.c

```
//UART.c
//Author: Kyle Howen - 03-08-2010
//Desription: This contains functions relating to UART1
//            and UART2 for GPS and PC communication.
#include "XPIC.h"

//RS232 Console Definitions
#define OPEN if(0)asm("nop");
#define CONSOLE else if(strcmp(RS232.RxBuffer,
#define EXECUTE )==0)

//RS232_TX() Transmits binary data to a personal computer for processing
int RS232_TX(void)
{
    if(RS232.TxComplete){     //If the last transmit is all done
            RS232.TxComplete = 0; //Flag a transmission in progress

        switch(RS232.OutputSelect){

        ////////////////////////////////////////////////////////
        //ASCII Representation of Data ////////////////////////////
        //NOTE - SPRINTF IS TERRIBLY INEFFICIENT //////////////////
        case 0:
            switch(RS232.AsciiSelect)
            {
                case 0: //Pressure and Air Acceleration
            sprintf(RS232.TxBuffer,"Plha:%3.6f,\t%3.6f,\t%3.6f,\t%3.6f\r",
            NAV.PressureLA,NAV.PressureHA,NAV.PressureD,NAV.AirAcceleration);
                    break;
                case 1: //Altimeters, ASI, and Airspeed
                    sprintf(RS232.TxBuffer,"LHS:%3.2f,%3.2f,%3.2f,%3.2f\r",
NAV.AltitudeL,NAV.AltitudeH,NAV.AirSpeedIndicator*MS2MPH, NAV.AirSpeed);
                    break;
                case 2: //Variables calculated upon Power-Up
sprintf(RS232.TxBuffer,"INIT:%1.6f,%1.6f,%1.6f,%1.6f,%1.6f,%1.6f,%3.2f\r",
NAV.LowAltGain,NAV.ASI_Rho,NAV.PressureDOFF,NAV.WXOff,NAV.WYOff,NAV.WZOff,
NAV.TakeOffAlt);
                    break;
                case 3: //Benchmarkers
sprintf(RS232.TxBuffer,"BENCH:%d,%d,%d,%d;;%1.2f,%1.2f,%1.2f,%1.2f\r",
IMU.Benchmark[0],IMU.Benchmark[1],IMU.Benchmark[2],IMU.Benchmark[3],
                            NAV.WeightYaw,NAV.WeightRP,NAV.Kp,NAV.Ki);
                    break;
                case 4: //State Management Variables
                    sprintf(RS232.TxBuffer,"STATE:%d,%d,%d,%d\r",
                        NAV.IntCount,NAV.StatesPerGPS,PR2,NAV.GPSLOST);
                    break;
                case 5: //Pitch, Roll, & Yaw
                    sprintf(RS232.TxBuffer,"PRY:%3.2f,%3.2f,%3.2f\r",
                    NAV.Pitch*RAD2D,NAV.Roll*RAD2D,NAV.Yaw*RAD2D);
                    break;
                case 6: //Raw IMU Integer Data
sprintf(RS232.TxBuffer,"I:%5d,%5d,%5d,%5d,%5d,%5d,%5d,%5d,%5d,%5d;%5d\r",
                        IMU.XGyro,IMU.YGyro,IMU.ZGyro,
                            IMU.XAccl,IMU.YAccl,IMU.ZAccl,
```

```
                            IMU.AuxADC,IMU.SupplyOut,
                                IMU.ASI,IMU.hAlt,
                                    NAV.IntCount);
                    break;
                case 7: //GPS Data after parsing
sprintf(RS232.TxBuffer,"G:%s,T:%6.3f,Lat:%f,Lon:%f,S:%.2f,C:%.2f\r",
GPS.Date,(double)GPS.UTCTime,(double)GPS.Latitude,(double)GPS.Longitude,
                            (double)GPS.GroundSpeed,(double)GPS.Course);
                    break;

                default:
                    RS232.AsciiSelect = 0;
                    break;
            }
            break;

        ////////////////////////////////////////////////////////
        //Binary Representation of Data//////////////////////////
        case 1: //XINS TX Protocol V1.0 - Codename: NAV

            RS232.TxArray[0]  = 0x69;
            RS232.TxArray[1]  = RS232.OutputSelect;
            RS232.TxArray[2]  = *(RS232.pLat+0);
            RS232.TxArray[3]  = *(RS232.pLat+1);
            RS232.TxArray[4]  = *(RS232.pLat+2);
            RS232.TxArray[5]  = *(RS232.pLat+3);
            RS232.TxArray[6] = *(RS232.pLon+0);
            RS232.TxArray[7] = *(RS232.pLon+1);
            RS232.TxArray[8] = *(RS232.pLon+2);
            RS232.TxArray[9] = *(RS232.pLon+3);
            RS232.TxArray[10] = *(RS232.pAirSpeed+0);
            RS232.TxArray[11] = *(RS232.pAirSpeed+1);
            RS232.TxArray[12] = *(RS232.pAirSpeed+2);
            RS232.TxArray[13] = *(RS232.pAirSpeed+3);
            RS232.TxArray[14] = *(RS232.pGroundSpeed+0);
            RS232.TxArray[15] = *(RS232.pGroundSpeed+1);
            RS232.TxArray[16] = *(RS232.pGroundSpeed+2);
            RS232.TxArray[17] = *(RS232.pGroundSpeed+3);
            RS232.TxArray[18] = *(RS232.pPressure+0);
            RS232.TxArray[19] = *(RS232.pPressure+1);
            RS232.TxArray[20] = *(RS232.pPressure+2);
            RS232.TxArray[21] = *(RS232.pPressure+3);
            RS232.TxArray[22] = *(RS232.pAltitude+0);
            RS232.TxArray[23] = *(RS232.pAltitude+1);
            RS232.TxArray[24] = *(RS232.pAltitude+2);
            RS232.TxArray[25] = *(RS232.pAltitude+3);
            RS232.TxArray[26] = *(RS232.pPitch+0);
            RS232.TxArray[27] = *(RS232.pPitch+1);
            RS232.TxArray[28] = *(RS232.pPitch+2);
            RS232.TxArray[29] = *(RS232.pPitch+3);
            RS232.TxArray[30] = *(RS232.pRoll+0);
            RS232.TxArray[31] = *(RS232.pRoll+1);
            RS232.TxArray[32] = *(RS232.pRoll+2);
            RS232.TxArray[33] = *(RS232.pRoll+3);
            RS232.TxArray[34] = *(RS232.pYaw+0);
            RS232.TxArray[35] = *(RS232.pYaw+1);
            RS232.TxArray[36] = *(RS232.pYaw+2);
```

```
    RS232.TxArray[37] = *(RS232.pYaw+3);
    RS232.TxArray[38]  = *(RS232.pTime+0);
    RS232.TxArray[39]  = *(RS232.pTime+1);
    RS232.TxArray[40]  = *(RS232.pTime+2);
    RS232.TxArray[41]  = *(RS232.pTime+3);
    RS232.TxArray[42] = *(RS232.pAirSpeedIndicator+0);
    RS232.TxArray[43] = *(RS232.pAirSpeedIndicator+1);
    RS232.TxArray[44] = *(RS232.pAirSpeedIndicator+2);
    RS232.TxArray[45] = *(RS232.pAirSpeedIndicator+3);
    break;

case 2: //Status Data - Codename: STA

    RS232.TxArray[0]  = 0x69;
    RS232.TxArray[1]  = RS232.OutputSelect;
    RS232.TxArray[2]  = *(RS232.pBench0+0);
    RS232.TxArray[3]  = *(RS232.pBench0+1);
    RS232.TxArray[4]  = *(RS232.pBench1+0);
    RS232.TxArray[5]  = *(RS232.pBench1+1);
    RS232.TxArray[6]  = *(RS232.pBench2+0);
    RS232.TxArray[7]  = *(RS232.pBench2+1);
    RS232.TxArray[8]  = *(RS232.pBench3+0);
    RS232.TxArray[9]  = *(RS232.pBench3+1);
    RS232.TxArray[10] = *(RS232.pIntCount+0);
    RS232.TxArray[11] = *(RS232.pIntCount+1);
    RS232.TxArray[12] = *(RS232.pStatesPerGPS+0);
    RS232.TxArray[13] = *(RS232.pStatesPerGPS+1);
    RS232.TxArray[14] = *(RS232.pGPSLOST+0);
    RS232.TxArray[15] = *(RS232.pGPSLOST+1);
    RS232.TxArray[16] = *(RS232.pGPSValidData+0);
    RS232.TxArray[17] = *(RS232.pGPSValidData+1);
    break;

case 3: //IMU Data - Codename: IMU
    RS232.TxArray[0]  = 0x69;
    RS232.TxArray[1]  = RS232.OutputSelect;
    RS232.TxArray[2]  = *(RS232.pVcc+0);
    RS232.TxArray[3]  = *(RS232.pVcc+1);
    RS232.TxArray[4]  = *(RS232.pXGyro+0);
    RS232.TxArray[5]  = *(RS232.pXGyro+1);
    RS232.TxArray[6]  = *(RS232.pYGyro+0);
    RS232.TxArray[7]  = *(RS232.pYGyro+1);
    RS232.TxArray[8]  = *(RS232.pZGyro+0);
    RS232.TxArray[9]  = *(RS232.pZGyro+1);
    RS232.TxArray[10] = *(RS232.pXAccl+0);
    RS232.TxArray[11] = *(RS232.pXAccl+1);
    RS232.TxArray[12] = *(RS232.pYAccl+0);
    RS232.TxArray[13] = *(RS232.pYAccl+1);
    RS232.TxArray[14] = *(RS232.pZAccl+0);
    RS232.TxArray[15] = *(RS232.pZAccl+1);
    RS232.TxArray[16] = *(RS232.plAlt+0);
    RS232.TxArray[17] = *(RS232.plAlt+1);
    RS232.TxArray[18] = *(RS232.phAlt+0);
    RS232.TxArray[19] = *(RS232.phAlt+1);
    RS232.TxArray[20] = *(RS232.pASI+0);
    RS232.TxArray[21] = *(RS232.pASI+1);
    break;
```

```
case 4: //Altitude Calibration - Codename: ACL
    RS232.TxArray[0]  = 0x69;
    RS232.TxArray[1]  = RS232.OutputSelect;
    RS232.TxArray[2]  = *(RS232.pLat+0);
    RS232.TxArray[3]  = *(RS232.pLat+1);
    RS232.TxArray[4]  = *(RS232.pLat+2);
    RS232.TxArray[5]  = *(RS232.pLat+3);
    RS232.TxArray[6] = *(RS232.pLon+0);
    RS232.TxArray[7] = *(RS232.pLon+1);
    RS232.TxArray[8] = *(RS232.pLon+2);
    RS232.TxArray[9] = *(RS232.pLon+3);
    RS232.TxArray[10] = *(RS232.pPressureLA+0);
    RS232.TxArray[11] = *(RS232.pPressureLA+1);
    RS232.TxArray[12] = *(RS232.pPressureLA+2);
    RS232.TxArray[13] = *(RS232.pPressureLA+3);
    RS232.TxArray[14] = *(RS232.pPressureHA+0);
    RS232.TxArray[15] = *(RS232.pPressureHA+1);
    RS232.TxArray[16] = *(RS232.pPressureHA+2);
    RS232.TxArray[17] = *(RS232.pPressureHA+3);
    RS232.TxArray[18] = *(RS232.pAltitudeL+0);
    RS232.TxArray[19] = *(RS232.pAltitudeL+1);
    RS232.TxArray[20] = *(RS232.pAltitudeL+2);
    RS232.TxArray[21] = *(RS232.pAltitudeL+3);
    RS232.TxArray[22] = *(RS232.pAltitudeH+0);
    RS232.TxArray[23] = *(RS232.pAltitudeH+1);
    RS232.TxArray[24] = *(RS232.pAltitudeH+2);
    RS232.TxArray[25] = *(RS232.pAltitudeH+3);
    RS232.TxArray[26] = *(RS232.plAlt+0);
    RS232.TxArray[27] = *(RS232.plAlt+1);
    RS232.TxArray[28] = *(RS232.phAlt+0);
    RS232.TxArray[29] = *(RS232.phAlt+1);
    RS232.TxArray[30]  = *(RS232.pVcc+0);
    RS232.TxArray[31]  = *(RS232.pVcc+1);

    break;
case 5: //Velocity Calibration - Codename: VCL
    RS232.TxArray[0]  = 0x69;
    RS232.TxArray[1]  = RS232.OutputSelect;
    RS232.TxArray[2] = *(RS232.pAirSpeedIndicator+0);
    RS232.TxArray[3] = *(RS232.pAirSpeedIndicator+1);
    RS232.TxArray[4] = *(RS232.pAirSpeedIndicator+2);
    RS232.TxArray[5] = *(RS232.pAirSpeedIndicator+3);
    RS232.TxArray[6] = *(RS232.pGroundSpeed+0);
    RS232.TxArray[7] = *(RS232.pGroundSpeed+1);
    RS232.TxArray[8] = *(RS232.pGroundSpeed+2);
    RS232.TxArray[9] = *(RS232.pGroundSpeed+3);
    RS232.TxArray[10] = *(RS232.pPitch+0);
    RS232.TxArray[11] = *(RS232.pPitch+1);
    RS232.TxArray[12] = *(RS232.pPitch+2);
    RS232.TxArray[13] = *(RS232.pPitch+3);
    RS232.TxArray[14] = *(RS232.pASI_Rho+0);
    RS232.TxArray[15] = *(RS232.pASI_Rho+1);
    RS232.TxArray[16] = *(RS232.pASI_Rho+2);
    RS232.TxArray[17] = *(RS232.pASI_Rho+3);
    RS232.TxArray[18] = *(RS232.pTemp+0);
    RS232.TxArray[19] = *(RS232.pTemp+1);
    RS232.TxArray[20] = *(RS232.pTemp+2);
```

```
            RS232.TxArray[21] = *(RS232.pTemp+3);
            RS232.TxArray[22] = *(RS232.pASI+0);
            RS232.TxArray[23] = *(RS232.pASI+1);
            RS232.TxArray[24]  = *(RS232.pVcc+0);
            RS232.TxArray[25]  = *(RS232.pVcc+1);

            break;
        case 6: //Gain Display - Codename: GNS
            RS232.TxArray[0]  = 0x69;
            RS232.TxArray[1]  = RS232.OutputSelect;
            RS232.TxArray[2]  = *(RS232.pKp+0);
            RS232.TxArray[3]  = *(RS232.pKp+1);
            RS232.TxArray[4]  = *(RS232.pKp+2);
            RS232.TxArray[5]  = *(RS232.pKp+3);
            RS232.TxArray[6] = *(RS232.pKi+0);
            RS232.TxArray[7] = *(RS232.pKi+1);
            RS232.TxArray[8] = *(RS232.pKi+2);
            RS232.TxArray[9] = *(RS232.pKi+3);
            RS232.TxArray[10] = *(RS232.pWeightYaw+0);
            RS232.TxArray[11] = *(RS232.pWeightYaw+1);
            RS232.TxArray[12] = *(RS232.pWeightYaw+2);
            RS232.TxArray[13] = *(RS232.pWeightYaw+3);
            RS232.TxArray[14] = *(RS232.pWeightRP+0);
            RS232.TxArray[15] = *(RS232.pWeightRP+1);
            RS232.TxArray[16] = *(RS232.pWeightRP+2);
            RS232.TxArray[17] = *(RS232.pWeightRP+3);
            RS232.TxArray[18] = *(RS232.pdeltaGain+0);
            RS232.TxArray[19] = *(RS232.pdeltaGain+1);
            RS232.TxArray[20] = *(RS232.pdeltaGain+2);
            RS232.TxArray[21] = *(RS232.pdeltaGain+3);
            RS232.TxArray[22] = *(RS232.pdeltaGain+0);
            break;

        default:
            RS232.OutputSelect = 0;
            break;
        }
            IFS0 |= 0x0400; //Manually Trigger an Interrupt
            return 1; //Return Success!
    }
    else    //Transmission in progress
        return 0;
}

//RS232_Update()    -Manages recieved ascii sentences recieved from a PC via
//RS232 connection that are decoded into internal program operations. (Much
//like a console)

int RS232_Update(void)
{
    if(RS232.NewData == 1 && RS232.Lock == 0)
    //Ingnore wether the checksum is valid or not for now!
    {
      URXI_DISABLE;
        RS232.NewData = 0;
        //Prevent from processing the same command twice!
```

```
////////////////////////////////////////////////////////////////////
//Command Console (Put The Most Commonly Executed Commands at the Top)
//  -The first character of the command '$' should be excluded as seen below:
            OPEN
        CONSOLE "NAV"      EXECUTE RS232.OutputSelect = 1;
        CONSOLE "STA"      EXECUTE RS232.OutputSelect = 2;
        CONSOLE "IMU"      EXECUTE RS232.OutputSelect = 3;
        CONSOLE "ACL"      EXECUTE RS232.OutputSelect = 4;
        CONSOLE "VCL"      EXECUTE RS232.OutputSelect = 5;
        CONSOLE "GNS"      EXECUTE RS232.OutputSelect = 6;
        CONSOLE "KY+"      EXECUTE NAV.WeightYaw += NAV.deltaGain;
        CONSOLE "KY-"      EXECUTE NAV.WeightYaw -= NAV.deltaGain;
        CONSOLE "KR+"      EXECUTE NAV.WeightRP += NAV.deltaGain;
        CONSOLE "KR-"      EXECUTE NAV.WeightRP -= NAV.deltaGain;
        CONSOLE "KP+"      EXECUTE NAV.Kp += NAV.deltaGain;
        CONSOLE "KP-"      EXECUTE NAV.Kp -= NAV.deltaGain;
        CONSOLE "KI+"      EXECUTE NAV.Ki += NAV.deltaGain;
        CONSOLE "KI-"      EXECUTE NAV.Ki -= NAV.deltaGain;
        CONSOLE "GO+"      EXECUTE NAV.deltaGain += 1.0;
        CONSOLE "GO-"      EXECUTE NAV.deltaGain -= 1.0;
        CONSOLE "GT+"      EXECUTE NAV.deltaGain += 0.1;
        CONSOLE "GT-"      EXECUTE NAV.deltaGain -= 0.1;
        CONSOLE "GH+"      EXECUTE NAV.deltaGain += 0.01;
        CONSOLE "GH-"      EXECUTE NAV.deltaGain -= 0.01;
        CONSOLE "RTX"      EXECUTE RS232_TX();
        CONSOLE "OS+"      EXECUTE RS232.OutputSelect++;
        CONSOLE "OS-"      EXECUTE RS232.OutputSelect--;
        CONSOLE "AS+"      EXECUTE RS232.AsciiSelect++;
        CONSOLE "AS-"      EXECUTE RS232.AsciiSelect--;
        CONSOLE "LD1"      EXECUTE LED1 ^= 1;
        CONSOLE "LD2"      EXECUTE LED2 ^= 1;
        CONSOLE "LD3"      EXECUTE LED3 ^= 1;
        CONSOLE "LD4"      EXECUTE LED4 ^= 1;
        CONSOLE "P2-"      EXECUTE PR2 -= 10;
        CONSOLE "P2+"      EXECUTE PR2 += 10;
        CONSOLE "NVI"      EXECUTE NAV_Init();
        //CONSOLE "G56"      EXECUTE GPS_TX(2);


////////////////////////////////////////////////////////////////////

        URXI_ENABLE;
            return 1; //Command Processed
      }
      return 0; //No Commands Processed
}

//GPS_TX(int) transmits premade NMEA strings to the LOCOSYS GPS via UART2
int GPS_TX(int TxOption)
{
      if(GPS.TxComplete){
            GPS.TxComplete = 0;

        switch(TxOption){
            case 0: sprintf(GPS.TxBuffer,"%s", LOCOSYS_REFRESH_RATE_100);
            break;
```

```
            case 1: sprintf(GPS.TxBuffer,"%s", LOCOSYS_REFRESH_RATE_250);
            break;
            case 2: sprintf(GPS.TxBuffer,"%s", LOCOSYS_BAUD_RATE_57600);
            break;
        }
            /*
            //XP-NMEA Formatting
            int i;
            for(i = 0; i < 96; i++) //calc backup calc checksum
                  GPS.TxChecksum ^= GPS.TxBuffer[i];
            sprintf(GPS.TxBuffer,"$%s*%x\r\n",GPS.TxBuffer,GPS.TxChecksum);
            */
            IFS1 |= 0x0200;
            //Return Success!
            return 1;
      }
      else
            IFS1 |= 0x0200;
      return 0;
}

//GPS_Update manages recieved ascii data from the LOCOSYS GPS and decodes the
data into local numerical data.
int GPS_Update(int bDateTime)
{
      if( (GPS.ChecksumFlag == 2) && (GPS.ReadChecksum == GPS.CalcChecksum) )
      //We have valid GPS Data to parse!
      {
            GPSI_DISABLE;
            GPS.ChecksumFlag = 0; //Prevent us from parsing same data twice

            //Example RMC Parse Code
"$GPRMC,053740.200,A,1234.5678,N,11223.3445,E,2.69,79.65,100106,,,A*69"
            if(bDateTime)
                  GPS.ParseNumber =
sscanf(GPS.Buffer,"%*6s,%f,%c,%f,%c,%f,%c,%f,%f,%6s,%*s",&GPS.UTCTime,
&GPS.ValidData,&GPS.Latitude,&GPS.North,&GPS.Longitude,&GPS.East,
&GPS.GroundSpeed,&GPS.Course,GPS.Date);
            else
                  GPS.ParseNumber =
sscanf(GPS.Buffer,"%*17s,%c,%f,%c,%f,%c,%f,%f,%*s",&GPS.ValidData,
&GPS.Latitude,&GPS.North,&GPS.Longitude,&GPS.East,&GPS.GroundSpeed,
&GPS.Course);

        if(GPS.ValidData != 'A'){
            NAV.GPSValidData = 0;
                  return GPS.ParseNumber; //0 for bad parse
        }
            else
            {
            GPS.NewGroundSpeed = 1;
            NAV.GPSValidData = 1;
                  //LAT  - Decimal Degree Formatting
                  double fractpart,intpart;
                  fractpart = modf((GPS.Latitude/100),&intpart);
                  GPS.Latitude = intpart + (fractpart / 0.6);
                  if(GPS.North == 'S')
```

```
                        GPS.Latitude *= -1;
                  //LONG - Decimal Degree Formatting
                  fractpart = modf((GPS.Longitude/100),&intpart);
                  GPS.Longitude = intpart + (fractpart / 0.6);
                  if(GPS.East == 'W')
                        GPS.Longitude *= -1;
            }

            GPSI_ENABLE;
            return 1; //1 for completed parse
      }
      return 2; //2 for no checksummed data
}
//UART_Init() initializes variables needed for UART1 Transmissions
void UART_Init(void){
    RS232.OutputSelect = 1;
    RS232.OutputLength[1] = 46;
    RS232.OutputLength[2] = 18;
    RS232.OutputLength[3] = 22;
    RS232.OutputLength[4] = 32;
    RS232.OutputLength[5] = 26;
    RS232.OutputLength[6] = 23;

    //NAV BINARY DATA VARIABLES
    RS232.pTime         = (char *)(&GPS.UTCTime);
    RS232.pLat          = (char *)(&GPS.Latitude);
    RS232.pLon          = (char *)(&GPS.Longitude);
    RS232.pGroundSpeed  = (char *)(&GPS.GroundSpeed);

    RS232.pPressure     = (char *)(&NAV.Pressure);
    RS232.pAltitude     = (char *)(&NAV.Altitude);
    RS232.pAirSpeed     = (char *)(&NAV.AirSpeed);
    RS232.pAirSpeedIndicator     = (char *)(&NAV.AirSpeedIndicator);

    RS232.pPitch        = (char *)(&NAV.Pitch);
    RS232.pRoll         = (char *)(&NAV.Roll);
    RS232.pYaw          = (char *)(&NAV.Yaw);

    //IMU DATA
    RS232.plAlt         = (char *)(&IMU.AuxADC);
    RS232.phAlt         = (char *)(&IMU.hAlt);
    RS232.pASI          = (char *)(&IMU.ASI);
    RS232.pVcc          = (char *)(&IMU.SupplyOut);
    RS232.pXGyro        = (char *)(&IMU.XGyro);
    RS232.pYGyro        = (char *)(&IMU.YGyro);
    RS232.pZGyro        = (char *)(&IMU.ZGyro);
    RS232.pXAccl        = (char *)(&IMU.XAccl);
    RS232.pYAccl        = (char *)(&IMU.YAccl);
    RS232.pZAccl        = (char *)(&IMU.ZAccl);

    //STATUS DATA
    RS232.pBench0       = (char *)(&IMU.Benchmark[0]);
    RS232.pBench1       = (char *)(&IMU.Benchmark[1]);
    RS232.pBench2       = (char *)(&IMU.Benchmark[2]);
    RS232.pBench3       = (char *)(&IMU.Benchmark[3]);
    RS232.pIntCount     = (char *)(&NAV.IntCount);
    RS232.pStatesPerGPS = (char *)(&NAV.StatesPerGPS);
```

```
    RS232.pGPSLOST      = (char *)(&NAV.GPSLOST);
    RS232.pGPSValidData = (char *)(&NAV.GPSValidData);

    //CALIBRATION EXTRAS
    RS232.pPressureHA   = (char *)(&NAV.PressureHA);
    RS232.pPressureLA   = (char *)(&NAV.PressureLA);
    RS232.pPressureD    = (char *)(&NAV.PressureD);
    RS232.pAltitudeH    = (char *)(&NAV.AltitudeH);
    RS232.pAltitudeL    = (char *)(&NAV.AltitudeL);
    RS232.pASI_Rho      = (char *)(&NAV.ASI_Rho);
    RS232.pTemp         = (char *)(&NAV.Temp);

    //Gain Pointers
    RS232.pKp           = (char *)(&NAV.Kp);
    RS232.pKi           = (char *)(&NAV.Ki);
    RS232.pWeightYaw    = (char *)(&NAV.WeightYaw);
    RS232.pWeightRP     = (char *)(&NAV.WeightRP);
    RS232.pdeltaGain    = (char *)(&NAV.deltaGain);
}
```

IMU.c

```
//IMU.c
//Author: Kyle Howen - 03-08-2010
//Desription: This houses the functions that apply to configuring
//            and reading data from the ADIS16365 IMU. The IMU
//            structure contains variables for all the data used
//            to configure the IMU and all data recieved from the IMU.
//            Refer to the ADIS16365 Datasheet for more info.
#include "XPIC.h"

//IMU.c.

//IMU_Init() is run at startup to configure the IMU for default use
void IMU_Init(void)
{
    IMU.TxData = 0xB904; //Angular Rate Sensitivity 300deg/s
      SPI1_TXRX();

      IMU.TxData = 0xB806; //Sample Taps to 16 per stage
      SPI1_TXRX();

      //Linear Acceleration Bias Compensation (Low Freq Accl Correction)
      IMU.TxData = 0xB486;
      SPI1_TXRX();
}

//IMU_Update() sequentially requests all the required XINS IMU data
//and stores what is returned over the SPI. No interrupts are used
//in this function.
void IMU_Update(void)
{
    IMU.ReadAll = 0;
    IMU.NewData = 0;
      while(1){
        switch(IMU.ReadAll){
            case 0:
                    IMU.TxData = 0x0200;
                break;
            case 1:
                IMU.TxData = 0x0400;
                break;
            case 2:
                IMU.SupplyOut = IMU.RxData & 0x0FFF;
                    IMU.TxData = 0x0600;
                break;
            case 3:
                IMU.XGyro = IMU.RxData;
                    SignExtend(&IMU.XGyro,12);
                        IMU.TxData = 0x0800;
                break;
            case 4:
                IMU.YGyro = IMU.RxData;
                        SignExtend(&IMU.YGyro,12);
                        IMU.TxData = 0x0A00;
```

```
                break;
            case 5:
                IMU.ZGyro = IMU.RxData;
                SignExtend(&IMU.ZGyro,12);
                IMU.TxData = 0x0C00;
                break;
            case 6:
                IMU.XAccl = IMU.RxData;
                SignExtend(&IMU.XAccl,12);
                IMU.TxData = 0x0E00;
                break;
            case 7:
                IMU.YAccl = IMU.RxData;
                SignExtend(&IMU.YAccl,12);
                IMU.TxData = 0x1000;
                break;
            case 8:
                IMU.ZAccl = IMU.RxData;
                SignExtend(&IMU.ZAccl,12);
                IMU.TxData = 0x1200;
          break;
            case 9:
                IMU.XTemp = IMU.RxData;
                SignExtend(&IMU.XTemp,10);
                IMU.TxData = 0x1400;
                break;
            case 10:
                IMU.YTemp = IMU.RxData;
                SignExtend(&IMU.YTemp,10);
                IMU.TxData = 0x1600;
                break;
            case 11:
                IMU.ZTemp = IMU.RxData;
                SignExtend(&IMU.ZTemp,10);
                IMU.TxData = 0x1600;    // for good measure
                break;
            case 12:
                IMU.AuxADC = IMU.RxData & 0x0FFF;
                IMU.NewData = 1;
                IMU.ReadAll = 0;
                break;
    default:
        IMU.NewData = 1;
        break;
    }
    if(IMU.NewData)
          break;

    SPI1_TXRX();
IMU.ReadAll++;

    if(IMU.RxData & 0x4000){ //Error found, try again
          IMU.ReadAll = 0;
          IMU.TxData = 0x3C00;
          SPI1_TXRX();
          SPI1_TXRX();
          IMU.Diagnostic = IMU.RxData;
```

```
                continue;
            }
        }
}

//SPI1_TXRX is an Uninterruptable Transmit and Recieve Function (For
Exclusive Communication)
void SPI1_TXRX(void)
{
    //Disable GPS Interrupts for this sensitive transfer
    // NOTE: Measured time for this period is no greater than 50us
    GPSI_DISABLE;
    if(SPI1STAT & 0x0040)     //If Overflow then Clear it.
            SPI1STAT &= 0xFFBF;

    SS1  = 1; //THIS IS TIME SENSITIVE
      SS1  = 0; //...MUST HAPPEN EARLY ENOUGH BEFORE SPI1-TX IS CALLED!

      if(SPI1STAT & 0x0001) //if SPIRBF (received data waiting)
            IMU.RxData = SPI1BUF;
      while(SPI1STAT & 0x0002)
            asm("nop");//wait while SPITBF flag is high
    SPI1BUF = IMU.TxData; //write data to the transmit buffer when all clear

    while(!(SPI1STAT & 0x0001))
        asm("nop");//wait while we recieve
    IMU.RxData = SPI1BUF;

    SS1 = 1;
    GPSI_ENABLE;
    //Enable GPS Interrupts again:

    delay_us(10);
}

//Sign Extend operates on the data recieved from the IMU by extending the
sign based on the TwosComp flag and the BitLenght of the data.
void SignExtend(int* TwosComp, int BitLength)
{
    switch(BitLength)
    {
    case 12:
        if(*TwosComp & 0x2000)  //Sign Extend
            *TwosComp |= 0xC000;
      else
            *TwosComp &= 0x3FFF;
        break;

    case 10:
        if(*TwosComp & 0x0800)
            *TwosComp |= 0xF000;
            else
                *TwosComp &= 0x0FFF;
        break;
    default: break;
    }
}
```

Appendix F - XINSGUI Source Code

The following source code was written for the XINGUI Matlab 7.5.0 application. XINGUI frontend figure was modeled in Matlab's GUIDE and compiled by Matlab's DeployTool. The source code listed below has been formatted to fit a printable document and comes from XINGUI.m

XINSGUI.m

```
%File: XINSGUI.m
%Author(s): Kyle Howen - 03-09-2010
%           MATLAB's GUIDE Automatic Code Generation

function varargout = XINSGUI2(varargin)
% XINSGUI2 M-file for XINSGUI2.fig
%      XINSGUI2, by itself, creates a new XINSGUI2 or raises the existing
%      singleton*.
%
%      H = XINSGUI2 returns the handle to a new XINSGUI2 or the handle to
%      the existing singleton*.
%
%      XINSGUI2('CALLBACK',hObject,eventData,handles,...) calls the local
%      function named CALLBACK in XINSGUI2.M with the given input arguments.
%
%      XINSGUI2('Property','Value',...) creates a new XINSGUI2 or raises the
%      existing singleton*.  Starting from the left, property value pairs are
%      applied to the GUI before XINSGUI2_OpeningFcn gets called.  An
%      unrecognized property name or invalid value makes property application
%      stop.  All inputs are passed to XINSGUI2_OpeningFcn via varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help XINSGUI2

% Last Modified by GUIDE v2.5 17-Feb-2010 22:11:46

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @XINSGUI2_OpeningFcn, ...
                   'gui_OutputFcn',  @XINSGUI2_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
```

```
end


% End initialization code - DO NOT EDIT

% --- Executes just before XINSGUI2 is made visible.
function XINSGUI2_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to XINSGUI2 (see VARARGIN)

% Choose default command line output for XINSGUI2
handles.output = hObject;

%XINS MATLAB VARIABLES
% Update handles structure
guidata(hObject, handles);

% UIWAIT makes XINSGUI2 wait for user response (see UIRESUME)
% uiwait(handles.figure1);


% --- Outputs from this function are returned to the command line.
function varargout = XINSGUI2_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;


% --- Executes on button press in pbCapture.
function pbCapture_Callback(hObject, eventdata, handles)
    %XINS-MATLAB Global Declarations
    global capturing; global ACCDATA; global index; global sRS232;
    global TX_SELECT; global TX_FLOATS; global TX_WORDS;

    global GoogleEarth;
    global geLongitude;
    global geLatitude;
    global geAltitude;
    global geHeading;
    global geGroundSpeed;
    GECounter = 0; GEDivider = 5;

    set(handles.pbSave,'enable','on');

    if(capturing == 0)
        set(handles.pbCapture,'string','Stop');
        set(handles.tCapture,'string','Running...');
```

```
        set(handles.tCapture,'ForegroundColor','red');
        capturing = 1;
        set(handles.pbSelect,'enable','off');
        set(handles.pbCheckAll,'enable','off');

        cb(1) = get(handles.cb1,'value'); set(handles.cb1,'enable','off');
        cb(2) = get(handles.cb2,'value'); set(handles.cb2,'enable','off');
        cb(3) = get(handles.cb3,'value'); set(handles.cb3,'enable','off');
        cb(4) = get(handles.cb4,'value'); set(handles.cb4,'enable','off');
        cb(5) = get(handles.cb5,'value'); set(handles.cb5,'enable','off');
        cb(6) = get(handles.cb6,'value'); set(handles.cb6,'enable','off');
        cb(7) = get(handles.cb7,'value'); set(handles.cb7,'enable','off');
        cb(8) = get(handles.cb8,'value'); set(handles.cb8,'enable','off');
        cb(9) = get(handles.cb9,'value'); set(handles.cb9,'enable','off');
        cb(10) = get(handles.cb10,'value'); set(handles.cb10,'enable','off');
        cb(11) = get(handles.cb11,'value'); set(handles.cb11,'enable','off');
        cb(12) = get(handles.cb12,'value'); set(handles.cb12,'enable','off');

        %XINS-MATLAB INIT CODE
        index = 1; %Reset data index to 1;
        ACCDATA = zeros(1,2); %Reset data matrix
        if(sRS232.BytesAvailable > 0) %Clean out the buffer before we start
logging
            fread(sRS232,sRS232.BytesAvailable);
        end
        try
            %XINS-MATLAB Mainloop
            while(capturing == 1)
                %RECIEVE BINARY DATA VIA readasync AND fread
                if(sRS232.BytesAvailable > 0)

                    %Check for 0x69 Header
                    TX_Checksum = 0;
                    while(TX_Checksum ~= 105)
                        TX_Checksum = fread(sRS232,1,'uchar');
                    end

                    %Validate the Output Selection
                    OutputSelect = fread(sRS232,1,'uchar');
                    if(OutputSelect ~= TX_SELECT)
                        capturing = 0;
                        break;
                    end

                    %Read out the recieved data
                    CURRENTDATA = [0 0 0 0 0 0 0 0 0 0 0 0]; %Reset current
data matrix
                    for iW = 1:TX_FLOATS(TX_SELECT)
                       CURRENTDATA(iW) = fread(sRS232,1,'single');
                    end
                    for iW = TX_FLOATS(TX_SELECT)+1:TX_WORDS(TX_SELECT)
                        CURRENTDATA(iW) = fread(sRS232,1,'int16');
                    end
```

```
                     %Store Selected Data to Accumulated Data Matrix: ACCDATA
                    x_index = 1; %Column Index for Accumulated Data Matrix:
ACCDATA
                    for iS = 1:12
                        if(cb(iS))
                            ACCDATA(index,x_index) = CURRENTDATA(iS);
                            x_index = x_index + 1;
                        end
                    end

                    %Post recieved data to HUD:
                    set(handles.tGauge1,'string',CURRENTDATA(1));
                    set(handles.tGauge2,'string',CURRENTDATA(2));
%Homogeneous Gauges
                    set(handles.tGauge5,'string',CURRENTDATA(5));
                    switch TX_SELECT %Output Selection Specific Code ie.
Google Earth
                        case {1}       %Additional Gauges
                            CURRENTDATA(3) = CURRENTDATA(3)*1.15077945;
                            CURRENTDATA(4) = CURRENTDATA(4)*1.15077945;
                            CURRENTDATA(7) = CURRENTDATA(7)*57.2957795;
                            CURRENTDATA(8) = CURRENTDATA(8)*57.2957795;
                            CURRENTDATA(9) = CURRENTDATA(9)*57.2957795;
                            CURRENTDATA(11) = CURRENTDATA(11)*1.15077945;


set(handles.tGauge3,'string',sprintf('%.1f',CURRENTDATA(3)));

set(handles.tGauge4,'string',sprintf('%.1f',CURRENTDATA(4)));

set(handles.tGauge5,'string',sprintf('%.3f',CURRENTDATA(5)));

set(handles.tGauge6,'string',sprintf('%.1f',CURRENTDATA(6)));

set(handles.tGauge7,'string',sprintf('%.1f',CURRENTDATA(7)));

set(handles.tGauge8,'string',sprintf('%.1f',CURRENTDATA(8)));

set(handles.tGauge9,'string',sprintf('%.1f',CURRENTDATA(9)));

set(handles.tGauge10,'string',sprintf('%6.1f',CURRENTDATA(10)));

set(handles.tGauge11,'string',sprintf('%.1f',CURRENTDATA(11)));
                            if(GoogleEarth && (GECounter > GEDivider))
                                GECounter = 0;
                                geLatitude = CURRENTDATA(1);
                                geLongitude = CURRENTDATA(2);
                                geAltitude = CURRENTDATA(6);
                                geHeading = CURRENTDATA(9);
                                geGroundSpeed = CURRENTDATA(4);
                                KMLUpdate();
                            else
                                GECounter = GECounter + 1;
                            end

                        case {3}       %Additional Gauges
```

```
                            set(handles.tGauge3,'string',CURRENTDATA(3));
                            set(handles.tGauge4,'string',CURRENTDATA(4));
                            set(handles.tGauge5,'string',CURRENTDATA(5));
                            set(handles.tGauge6,'string',CURRENTDATA(6));
                            set(handles.tGauge7,'string',CURRENTDATA(7));
                            set(handles.tGauge8,'string',CURRENTDATA(8));
                            set(handles.tGauge9,'string',CURRENTDATA(9));
                            set(handles.tGauge10,'string',CURRENTDATA(10));
                        case 2
                            set(handles.tGauge3,'string',CURRENTDATA(3));
                            set(handles.tGauge4,'string',CURRENTDATA(4));
                            set(handles.tGauge5,'string',CURRENTDATA(5));
                            set(handles.tGauge6,'string',CURRENTDATA(6));
                            set(handles.tGauge7,'string',CURRENTDATA(7));
                            set(handles.tGauge8,'string',CURRENTDATA(8));
                        case 4
                            set(handles.tGauge3,'string',CURRENTDATA(3));
                            set(handles.tGauge4,'string',CURRENTDATA(4));
                            set(handles.tGauge5,'string',CURRENTDATA(5));
                            set(handles.tGauge6,'string',CURRENTDATA(6));
                            set(handles.tGauge7,'string',CURRENTDATA(7));
                            set(handles.tGauge8,'string',CURRENTDATA(8));
                            set(handles.tGauge9,'string',CURRENTDATA(9));
                        case 5
                            CURRENTDATA(1) = CURRENTDATA(1)*1.15077945;
                            CURRENTDATA(2) = CURRENTDATA(2)*1.15077945;
                            CURRENTDATA(3) = CURRENTDATA(3)*57.2957795;

                            set(handles.tGauge1,'string',CURRENTDATA(1));
                            set(handles.tGauge2,'string',CURRENTDATA(2));
                            set(handles.tGauge3,'string',CURRENTDATA(3));
                            set(handles.tGauge4,'string',CURRENTDATA(4));
                            set(handles.tGauge5,'string',CURRENTDATA(5));
                            set(handles.tGauge6,'string',CURRENTDATA(6));
                            set(handles.tGauge7,'string',CURRENTDATA(7));
                        case 6
                            set(handles.tGauge3,'string',CURRENTDATA(3));
                            set(handles.tGauge4,'string',CURRENTDATA(4));
                    end

                    %increment row data index
                    index = index + 1;
                end
                pause(0.01); % Temporary Processing Dummy (to allow code
break)
            end

        set(handles.tCapture,'string','Stopped');
        set(handles.tCapture,'ForegroundColor','blue');
        set(handles.pbCapture,'enable','on');
        set(handles.pbCapture,'string','Start');
        set(handles.pbSelect,'enable','on');


        catch
            set(handles.tCapture,'string','Error: Save& ReConnect');
```

```
            capturing = 0;
            set(handles.pbCapture,'string','Start');
            set(handles.pbSelect,'enable','off');
            set(handles.pbCapture,'enable','off');
            disp('Capture Loop Failed...Save your data and reconnect to your
device');
        end

        %set(handles.cb2,'enable','on');
        %set(handles.cb2,'enable','on');
        %set(handles.cb4,'enable','on');
        %set(handles.cb4,'enable','on');
        %set(handles.cb6,'enable','on');
        %set(handles.cb6,'enable','on');
        %set(handles.cb8,'enable','on');
        %set(handles.cb8,'enable','on');
        %set(handles.cb10,'enable','on');
        %set(handles.cb10,'enable','on');
        %set(handles.cb12,'enable','on');
        %set(handles.cb12,'enable','on');

    %XINS-MATLAB BREAK/CLEANUP CODE
    else
        set(handles.tCapture,'string','Stopping...');
        capturing = 0;
        set(handles.pbCapture,'enable','off');
    end

% hObject    handle to pbCapture (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
function KMLUpdate()
    global geLongitude;
    global geLatitude;
    global geAltitude;
    global geHeading;
    global geGroundSpeed;

    global kLong; global kLat; global kHeading; global kPName1; global
kPCoordinates1;
    global kCoordinates; global kNode; global kmlFileName;
    global TrackLength;
    sLon = sprintf('%.13f',geLongitude);
    sLat = sprintf('%.13f',geLatitude);
    sCoord = sprintf('%s,%s,%.1f',sLon,sLat,geAltitude);

    %Replacements
    kLong.replaceChild(kNode.createTextNode(sLon),getFirstChild(kLong));
    kLat.replaceChild(kNode.createTextNode(sLat),getFirstChild(kLat));

kHeading.replaceChild(kNode.createTextNode(sprintf('%.1f',geHeading')),getFir
stChild(kHeading));
    kPName1.replaceChild(kNode.createTextNode(sprintf('XINS
(%.1f)',geGroundSpeed)),getFirstChild(kPName1));
```

```
kPCoordinates1.replaceChild(kNode.createTextNode(sCoord),getFirstChild(kPCoor
dinates1));


    %Appendages
    kCoordinates.appendChild(kNode.createTextNode(sprintf('%s\n',sCoord)));
    if(TrackLength ~= 0)
        kCoordChildren = kCoordinates.getChildNodes;
        if(kCoordChildren.getLength >= (TrackLength+1))
            kCoordinates.removeChild(kCoordChildren.item(0));
        end
    end

    xmlwrite(kmlFileName,kNode);

% --- Executes on button press in pbRS232.
function pbRS232_Callback(hObject, eventdata, handles)
    global connected;
    global sRS232;
    if(connected == 0)
        %Create RS232 Com Connection
        sRS232 = serial('COM1');  %sGPS.Parity = 'none';
        sRS232.BaudRate = 57600;  %sGPS.StopBits = 1;
        sRS232.DataBits = 8;      %sGPS.Terminator = 'T';
        %sRS232.ReadAsyncMode = 'manual';
        sRS232.timeout = 0.5;
        %Open the connection
        try
            if(strcmp(get(sRS232,'status'),'closed'))
                fopen(sRS232);
            end
        catch
            disp('Failed opening COM1...check conflicting programs or
physical connection');
            fclose(sRS232);
            return
        end
        connected = 1;
        %set(handles.pbCapture,'enable','on');
        set(handles.pbSelect,'enable','on');
        set(handles.pbRS232TX,'enable','on');
        set(handles.pbRS232,'string','Disconnect');
        %set(handles.pbCheckAll,'enable','on');
    else
        if(strcmp(get(sRS232,'status'),'open'))
            fclose(sRS232);
        end
        %if(sRS232 ~= 0)
        %    sRS232 = 0;
        %end
        connected = 0;
        set(handles.pbCapture,'enable','off');
        set(handles.pbSelect,'enable','off');
        set(handles.pbRS232TX,'enable','off');
        set(handles.pbRS232,'string','Connect');
```

```
    end
% hObject    handle to pbRS232 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)


% --- Executes on button press in pbSave.
function pbSave_Callback(hObject, eventdata, handles)  %DATA STORAGE VIA
MSEXCEL
cb = zeros(1,12);
cb(1) = get(handles.cb1,'value');
cb(2) = get(handles.cb2,'value');
cb(3) = get(handles.cb3,'value');
cb(4) = get(handles.cb4,'value');
cb(5) = get(handles.cb5,'value');
cb(6) = get(handles.cb6,'value');
cb(7) = get(handles.cb7,'value');
cb(8) = get(handles.cb8,'value');
cb(9) = get(handles.cb9,'value');
cb(10) = get(handles.cb10,'value');
cb(11) = get(handles.cb11,'value');
cb(12) = get(handles.cb12,'value');
global TX_DATA1; global TX_DATA2; global TX_DATA3; global TX_DATA4; global
TX_DATA5; global TX_DATA6;
global TX_SELECT; global ACCDATA;
global xlsFileName;
switch TX_SELECT
    case 1
        TX_DATA = TX_DATA1;
    case 2
        TX_DATA = TX_DATA2;
    case 3
        TX_DATA = TX_DATA3;
    case 4
        TX_DATA = TX_DATA4;
    case 5
        TX_DATA = TX_DATA5;
    case 6
        TX_DATA = TX_DATA6;
    otherwise
        TX_DATA = zeros(1,12);
end
i_x = 1; xcell =
{'A1','B1','C1','D1','E1','F1','G1','H1','I1','J1','K1','L1'};
for i=1:12
    if(cb(i) == 1)

xlswrite(xlsFileName,TX_DATA(i),get(handles.ePage,'string'),xcell{i_x});
        i_x = i_x + 1;
    end
end
xlswrite(xlsFileName,ACCDATA,get(handles.ePage,'string'),'A2');
% hObject    handle to pbSave (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

```
function ePage_Callback(hObject, eventdata, handles)
% hObject    handle to ePage (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of ePage as text
%        str2double(get(hObject,'String')) returns contents of ePage as a
double


% --- Executes during object creation, after setting all properties.
function ePage_CreateFcn(hObject, eventdata, handles)
% hObject    handle to ePage (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in pbInitialize.
function pbInitialize_Callback(hObject, eventdata, handles)
% hObject    handle to pbInitialize (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

%Global TX Initializations
global TX_LENGTH; TX_LENGTH = [46 18 22 32 26 23]; %Transmission length in
Bytes!
global TX_FLOATS; TX_FLOATS = [11 0 0 6 5 5]; %Transmission Values
global TX_INTS;  TX_INTS =    [0 8 10 3 2 0];
global TX_WORDS; TX_WORDS   = TX_FLOATS + TX_INTS;
global TX_SELECT; TX_SELECT = 1;
global TX_DATA1; TX_DATA1 =
{'Latitude','Longitude','AirSpeed(mph)','GroundSpeed(mph)','Pressure(kPa)','A
ltitude(m)','Pitch(deg)','Roll(deg)','Yaw(deg)','Time','AirSpeedInd(mph)'};
%Transmission Data Matrix
global TX_DATA2; TX_DATA2 =
{'Bench0','Bench1','Bench2','Bench3','IntCount','StatesPerGPS','GPSLOST','GPS
ValidData'};
global TX_DATA3; TX_DATA3 =
{'SupplyOut','XGyro','YGyro','ZGyro','XAccl','YAccl','ZAccl','lAlt','hAlt','A
SI'};
global TX_DATA4; TX_DATA4 =
{'Latitude','Longitude','PressureLA(kPa)','PressureHA(kPa)','AltitudeLA(m)','
AltitudeHA(m)','ADCLA','ADCHA','ADCVCC'};
global TX_DATA5; TX_DATA5 =
{'AirSpeedInd(mph)','GroundSpeed(mph)','Pitch(deg)','Rho','Temp(K)','ADCASI',
'ADCVCC'};
```

```
global TX_DATA6; TX_DATA6 = {'Kp','Ki','WeightYaw','WeightRP','deltaGain'};
%Other Global Intializations
global capturing; capturing = 0;
global ACCDATA;   ACCDATA = 0;
global index;     index = 0;
global sRS232;    sRS232 = 0;
global connected; connected = 0;
global GoogleEarth; GoogleEarth = 0;
global TrackLength; TrackLength = 0;
global cbCheckAllToggle; cbCheckAllToggle = 0;

%Enable Push Buttons
set(handles.pbRS232,'enable','on');
set(handles.pbClearTrack,'enable','on');
set(handles.pbInitialize,'enable','off');

%kNode KML Structure
global kNode;
global kRootNode;
global kDocument;
    global kLookAt;
        global kLong;
        global kLat;
        global kRange;
        global kTilt;
        global kHeading;
    global kPlacemark1;
        global kPName1;
        global kStyle1;
            global kIconStyle1;
            global kIcon1;
                global kHref1;
                global kX1;
                global kY1;
                global kW1;
                global kH1;
        global kPoint1;
            global kAltitudeMode1;
            global kPCoordinates1;
    global kPlacemark2;
        global kPName2;
        global kStyle2;
            global kLineStyle2;
                global kColor2;
                global kWidth2;
        global kMultiGeometry;
            global kLineString;
                global kTessellate;
                global kAltitudeMode;
                global kCoordinates;
global kmlFileName; global xlsFileName;
kmlFileName = 'XINS-data.kml';
xlsFileName = 'xins.xls';
kNode = com.mathworks.xml.XMLUtils.createDocument('kml');
kRootNode = kNode.getDocumentElement;
kRootNode.setAttribute('xlmns','http://earth.google.com/kml/2.0');
```

```
%Initialize Nodes%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
kDocument   = kNode.createElement('Document');
    kLookAt     = kNode.createElement('LookAt');
        kLong       = kNode.createElement('longitude');
kLong.appendChild(kNode.createTextNode('-120.621118333333'));
        kLat        = kNode.createElement('latitude');
kLat.appendChild(kNode.createTextNode('35.1139283333333'));
        kRange      = kNode.createElement('range');
kRange.appendChild(kNode.createTextNode('1000'));
        kTilt       = kNode.createElement('tilt');
kTilt.appendChild(kNode.createTextNode('45'));
        kHeading    = kNode.createElement('heading');
kHeading.appendChild(kNode.createTextNode('0'));
    kLookAt.appendChild(kLat);
    kLookAt.appendChild(kLong);
    kLookAt.appendChild(kRange);
    kLookAt.appendChild(kTilt);
    kLookAt.appendChild(kHeading);

    kPlacemark1  = kNode.createElement('Placemark');
        kPName1      = kNode.createElement('name');
kPName1.appendChild(kNode.createTextNode('XINS'));
        kStyle1      = kNode.createElement('Style');
            kIconStyle1 = kNode.createElement('IconStyle');
                kIcon1      = kNode.createElement('Icon');
                    kHref1      = kNode.createElement('href');
kHref1.appendChild(kNode.createTextNode('root://icons/palette-4.png'));
                    kX1         = kNode.createElement('x');
kX1.appendChild(kNode.createTextNode('224'));
                    kY1         = kNode.createElement('y');
kY1.appendChild(kNode.createTextNode('224'));
                    kW1         = kNode.createElement('w');
kW1.appendChild(kNode.createTextNode('32'));
                    kH1         = kNode.createElement('h');
kH1.appendChild(kNode.createTextNode('32'));
                kIcon1.appendChild(kHref1);
                kIcon1.appendChild(kX1);
                kIcon1.appendChild(kY1);
                kIcon1.appendChild(kW1);
                kIcon1.appendChild(kH1);
            kIconStyle1.appendChild(kIcon1);
        kStyle1.appendChild(kIconStyle1);
        kPoint1      = kNode.createElement('Point');
            kAltitudeMode1   = kNode.createElement('altitudeMode');
kAltitudeMode1.appendChild(kNode.createTextNode('clampedToGround'));
            kPCoordinates1   = kNode.createElement('coordinates');
kPCoordinates1.appendChild(kNode.createTextNode('-
120.621118333333,35.1139283333333,0'));
        kPoint1.appendChild(kAltitudeMode1);
        kPoint1.appendChild(kPCoordinates1);
    kPlacemark1.appendChild(kPName1);
    kPlacemark1.appendChild(kStyle1);
    kPlacemark1.appendChild(kPoint1);

    kPlacemark2  = kNode.createElement('Placemark');
```

```
        kPName2      = kNode.createElement('name');
kPName2.appendChild(kNode.createTextNode('Track'));
        kStyle2      = kNode.createElement('Style');
            kLineStyle2 = kNode.createElement('LineStyle');
                kColor2     = kNode.createElement('color');
kColor2.appendChild(kNode.createTextNode('7F00FF00'));
                kWidth2     = kNode.createElement('width');
kWidth2.appendChild(kNode.createTextNode('7'));
            kLineStyle2.appendChild(kColor2);
            kLineStyle2.appendChild(kWidth2);
        kStyle2.appendChild(kLineStyle2);
        kMultiGeometry = kNode.createElement('MultiGeometry');
            kLineString = kNode.createElement('LineString');
                kTessellate      = kNode.createElement('tessellate');
kTessellate.appendChild(kNode.createTextNode('0'));
                kAltitudeMode    = kNode.createElement('altitudeMode');
kAltitudeMode.appendChild(kNode.createTextNode('clampedToGround'));
                kCoordinates     = kNode.createElement('coordinates');
            kLineString.appendChild(kTessellate);
            kLineString.appendChild(kAltitudeMode);
            kLineString.appendChild(kCoordinates);
        kMultiGeometry.appendChild(kLineString);
    kPlacemark2.appendChild(kPName2);
    kPlacemark2.appendChild(kStyle2);
    kPlacemark2.appendChild(kMultiGeometry);

kDocument.appendChild(kLookAt);
kDocument.appendChild(kPlacemark1);
kDocument.appendChild(kPlacemark2);

kRootNode.appendChild(kDocument);

xmlwrite(kmlFileName,kNode);
%Initialization Complete


% --- Executes on selection change in lbTRX.
function lbTRX_Callback(hObject, eventdata, handles)
% hObject    handle to lbTRX (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = get(hObject,'String') returns lbTRX contents as cell
array
%        contents{get(hObject,'Value')} returns selected item from lbTRX


% --- Executes during object creation, after setting all properties.
function lbTRX_CreateFcn(hObject, eventdata, handles)
% hObject    handle to lbTRX (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: listbox controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
```

```
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in pbSelect.
function pbSelect_Callback(hObject, eventdata, handles)
% hObject    handle to pbSelect (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
global sRS232; global TX_SELECT;
global TX_DATA1; global TX_DATA2; global TX_DATA3; global TX_DATA4; global
TX_DATA5; global TX_DATA6;
global GoogleEarth;
lbSelect = get(handles.lbTRX,'value');

set(handles.pbSave,'enable','off');
set(handles.pbCapture,'enable','on');
set(handles.pbCheckAll,'enable','on');

set(handles.tLabel1,'visible','on');
set(handles.tGauge1,'visible','on');
set(handles.cb1,'enable','on');
set(handles.tLabel2,'visible','on');
set(handles.tGauge2,'visible','on');
set(handles.cb2,'enable','on');
set(handles.tLabel3,'visible','on');
set(handles.tGauge3,'visible','on');
set(handles.cb3,'enable','on');
set(handles.tLabel4,'visible','on');
set(handles.tGauge4,'visible','on');
set(handles.cb4,'enable','on');
set(handles.tLabel5,'visible','on');
set(handles.tGauge5,'visible','on');
set(handles.cb5,'enable','on');
set(handles.tLabel6,'visible','on');
set(handles.tGauge6,'visible','on');
set(handles.cb6,'enable','on');
set(handles.tLabel7,'visible','on');
set(handles.tGauge7,'visible','on');
set(handles.cb7,'enable','on');
set(handles.tLabel8,'visible','on');
set(handles.tGauge8,'visible','on');
set(handles.cb8,'enable','on');
set(handles.tLabel9,'visible','on');
set(handles.tGauge9,'visible','on');
set(handles.cb9,'enable','on');
set(handles.tLabel10,'visible','on');
set(handles.tGauge10,'visible','on');
set(handles.cb10,'enable','on');
set(handles.tLabel11,'visible','on');
set(handles.tGauge11,'visible','on');
set(handles.cb11,'enable','on');
set(handles.tLabel12,'visible','on');
set(handles.tGauge12,'visible','on');
```

```
set(handles.cb12,'enable','on');
if(lbSelect == 1)
    TX_SELECT = 1;
    set(handles.pbGoogleEarth,'enable','on');
    try
        fprintf(sRS232,'$NAV');
    catch
        disp('Data Set Selection Failed!');
        return
    end
    set(handles.tLabel1,'string',TX_DATA1(1));
    set(handles.tLabel2,'string',TX_DATA1(2));
    set(handles.tLabel3,'string',TX_DATA1(3));
    set(handles.tLabel4,'string',TX_DATA1(4));
    set(handles.tLabel5,'string',TX_DATA1(5));
    set(handles.tLabel6,'string',TX_DATA1(6));
    set(handles.tLabel7,'string',TX_DATA1(7));
    set(handles.tLabel8,'string',TX_DATA1(8));
    set(handles.tLabel9,'string',TX_DATA1(9));
    set(handles.tLabel10,'string',TX_DATA1(10));
    set(handles.tLabel11,'string',TX_DATA1(11));

    set(handles.tLabel12,'visible','off');
set(handles.tGauge12,'visible','off'); set(handles.cb12,'enable','off');
set(handles.cb12,'value',0);
else
    set(handles.pbGoogleEarth,'enable','off');
    GoogleEarth = 0;
    set(handles.pbGoogleEarth,'string','Enable');
    set(handles.tGoogleEarth,'string','Off');
    set(handles.tGoogleEarth,'ForegroundColor','blue');
    set(handles.eTrackLength,'Enable','on');
end
if(lbSelect == 2)
    TX_SELECT = 2;
    try
        fprintf(sRS232,'$STA');
    catch
        disp('Data Set Selection Failed!');
        return
    end

    set(handles.tLabel1,'string',TX_DATA2(1));
    set(handles.tLabel2,'string',TX_DATA2(2));
    set(handles.tLabel3,'string',TX_DATA2(3));
    set(handles.tLabel4,'string',TX_DATA2(4));
    set(handles.tLabel5,'string',TX_DATA2(5));
    set(handles.tLabel6,'string',TX_DATA2(6));
    set(handles.tLabel7,'string',TX_DATA2(7));
    set(handles.tLabel8,'string',TX_DATA2(8));


set(handles.tLabel9,'visible','off');
set(handles.tGauge9,'visible','off');
set(handles.cb9,'enable','off');
set(handles.cb9,'value',0);
```

```
set(handles.tLabel10,'visible','off');
set(handles.tGauge10,'visible','off');
set(handles.cb10,'enable','off');
set(handles.cb10,'value',0);

set(handles.tLabel11,'visible','off');
set(handles.tGauge11,'visible','off');
set(handles.cb11,'enable','off');
set(handles.cb11,'value',0);

set(handles.tLabel12,'visible','off');
set(handles.tGauge12,'visible','off');
set(handles.cb12,'enable','off');
set(handles.cb12,'value',0);
end
if(lbSelect == 3)
    TX_SELECT = 3;
    try
        fprintf(sRS232,'$IMU');
    catch
        disp('Data Set Selection Failed!');
        return
    end


    set(handles.tLabel1,'string',TX_DATA3(1));
    set(handles.tLabel2,'string',TX_DATA3(2));
    set(handles.tLabel3,'string',TX_DATA3(3));
    set(handles.tLabel4,'string',TX_DATA3(4));
    set(handles.tLabel5,'string',TX_DATA3(5));
    set(handles.tLabel6,'string',TX_DATA3(6));
    set(handles.tLabel7,'string',TX_DATA3(7));
    set(handles.tLabel8,'string',TX_DATA3(8));
    set(handles.tLabel9,'string',TX_DATA3(9));
    set(handles.tLabel10,'string',TX_DATA3(10));

    set(handles.tLabel11,'visible','off');
set(handles.tGauge11,'visible','off'); set(handles.cb11,'enable','off');
set(handles.cb11,'value',0);
    set(handles.tLabel12,'visible','off');
set(handles.tGauge12,'visible','off'); set(handles.cb12,'enable','off');
set(handles.cb12,'value',0);
end
if(lbSelect == 4)
    TX_SELECT = 4;
    try
        fprintf(sRS232,'$ACL');
    catch
        disp('Data Set Selection Failed!');
        return
    end

    set(handles.tLabel1,'string',TX_DATA4(1));
    set(handles.tLabel2,'string',TX_DATA4(2));
    set(handles.tLabel3,'string',TX_DATA4(3));
```

```
    set(handles.tLabel4,'string',TX_DATA4(4));
    set(handles.tLabel5,'string',TX_DATA4(5));
    set(handles.tLabel6,'string',TX_DATA4(6));
    set(handles.tLabel7,'string',TX_DATA4(7));
    set(handles.tLabel8,'string',TX_DATA4(8));
    set(handles.tLabel9,'string',TX_DATA4(9));

    set(handles.tLabel10,'visible','off');
set(handles.tGauge10,'visible','off'); set(handles.cb10,'enable','off');
set(handles.cb10,'value',0);
    set(handles.tLabel11,'visible','off');
set(handles.tGauge11,'visible','off'); set(handles.cb11,'enable','off');
set(handles.cb11,'value',0);
    set(handles.tLabel12,'visible','off');
set(handles.tGauge12,'visible','off'); set(handles.cb12,'enable','off');
set(handles.cb12,'value',0);
end
if(lbSelect == 5)
    TX_SELECT = 5;
    try
        fprintf(sRS232,'$VCL');
    catch
        disp('Data Set Selection Failed!');
        return
    end

    set(handles.tLabel1,'string',TX_DATA5(1));
    set(handles.tLabel2,'string',TX_DATA5(2));
    set(handles.tLabel3,'string',TX_DATA5(3));
    set(handles.tLabel4,'string',TX_DATA5(4));
    set(handles.tLabel5,'string',TX_DATA5(5));
    set(handles.tLabel6,'string',TX_DATA5(6));
    set(handles.tLabel7,'string',TX_DATA5(7));


set(handles.tLabel8,'visible','off');
set(handles.tGauge8,'visible','off');
set(handles.cb8,'enable','off');
set(handles.cb8,'value',0);

set(handles.tLabel9,'visible','off');
set(handles.tGauge9,'visible','off');
set(handles.cb9,'enable','off');
set(handles.cb9,'value',0);

set(handles.tLabel10,'visible','off');
set(handles.tGauge10,'visible','off');
set(handles.cb10,'enable','off');
set(handles.cb10,'value',0);

set(handles.tLabel11,'visible','off');
set(handles.tGauge11,'visible','off');
set(handles.cb11,'enable','off');
set(handles.cb11,'value',0);

set(handles.tLabel12,'visible','off');
```

```
set(handles.tGauge12,'visible','off');
set(handles.cb12,'enable','off');
set(handles.cb12,'value',0);
end
if(lbSelect == 6)
    TX_SELECT = 6;
    try
        fprintf(sRS232,'$GNS');
    catch
        disp('Data Set Selection Failed!');
        return
    end
    set(handles.tGainInfo,'visible','on');

    set(handles.tLabel1,'string',TX_DATA6(1));
    set(handles.tLabel2,'string',TX_DATA6(2));
    set(handles.tLabel3,'string',TX_DATA6(3));
    set(handles.tLabel4,'string',TX_DATA6(4));
    set(handles.tLabel5,'string',TX_DATA6(5));

    set(handles.tLabel6,'visible','off');
set(handles.tGauge6,'visible','off');
set(handles.cb6,'enable','off');
set(handles.cb6,'value',0);
    set(handles.tLabel7,'visible','off');
set(handles.tGauge7,'visible','off');
set(handles.cb7,'enable','off');
set(handles.cb7,'value',0);

set(handles.tLabel8,'visible','off');
set(handles.tGauge8,'visible','off');
set(handles.cb8,'enable','off');
set(handles.cb8,'value',0);

set(handles.tLabel9,'visible','off');
set(handles.tGauge9,'visible','off');
set(handles.cb9,'enable','off');
set(handles.cb9,'value',0);

set(handles.tLabel10,'visible','off');
set(handles.tGauge10,'visible','off');
set(handles.cb10,'enable','off');
set(handles.cb10,'value',0);

set(handles.tLabel11,'visible','off');
set(handles.tGauge11,'visible','off');
set(handles.cb11,'enable','off');
set(handles.cb11,'value',0);

set(handles.tLabel12,'visible','off');
set(handles.tGauge12,'visible','off');
set(handles.cb12,'enable','off');
set(handles.cb12,'value',0);
else
    set(handles.tGainInfo,'visible','off');
end
```

```
% --- Executes on button press in pbRS232TX.
function pbRS232TX_Callback(hObject, eventdata, handles)
% hObject    handle to pbRS232TX (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
global sRS232;
try
    fprintf(sRS232,get(handles.eRS232TX,'String'));
catch
    disp('RS232 Transmit Failed!');
end


function eRS232TX_Callback(hObject, eventdata, handles)
% hObject    handle to eRS232TX (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of eRS232TX as text
%        str2double(get(hObject,'String')) returns contents of eRS232TX as a
double


% --- Executes during object creation, after setting all properties.
function eRS232TX_CreateFcn(hObject, eventdata, handles)
% hObject    handle to eRS232TX (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in pbGoogleEarth.
function pbGoogleEarth_Callback(hObject, eventdata, handles)
global GoogleEarth; global TrackLength;
if(GoogleEarth == 0)
    TrackLength = str2double(get(handles.eTrackLength,'string'));

    GoogleEarth = 1;
    set(handles.pbGoogleEarth,'string','Disable');
    set(handles.tGoogleEarth,'string','On');
    set(handles.tGoogleEarth,'ForegroundColor','red');
    set(handles.eTrackLength,'Enable','off');
else
    set(handles.eTrackLength,'Enable','on');
    GoogleEarth = 0;
    set(handles.pbGoogleEarth,'string','Enable');
    set(handles.tGoogleEarth,'string','Off');
    set(handles.tGoogleEarth,'ForegroundColor','blue');
```

```
end
% hObject    handle to pbGoogleEarth (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)


% --- Executes on button press in pbClearTrack.
function pbClearTrack_Callback(hObject, eventdata, handles)
% hObject    handle to pbClearTrack (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
%Delete Track
global kCoordinates;
try
    kCoordChildren = kCoordinates.getChildNodes;
    i0 = kCoordChildren.getLength-1;
    for i = 0:i0
       kCoordinates.removeChild(kCoordChildren.item(0));
    end
catch
    disp('Error clearing track coordinates');
end


% --- Executes during object creation, after setting all properties.
function eTrackLength_CreateFcn(hObject, eventdata, handles)
% hObject    handle to eTrackLength (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

 --- Executes on button press in pbCheckAll.
function pbCheckAll_Callback(hObject, eventdata, handles)
% hObject    handle to pbCheckAll (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
global cbCheckAllToggle;
if(cbCheckAllToggle == 0)
   cbCheckAllToggle = 1;
   set(handles.pbCheckAll,'string','Clear');
   if(strcmp(get(handles.cb1,'enable'),'on'))
        set(handles.cb1,'value',1); end

   if(strcmp(get(handles.cb2,'enable'),'on'))
    set(handles.cb2,'value',1); end

   if(strcmp(get(handles.cb3,'enable'),'on'))
    set(handles.cb3,'value',1); end
```

```
    if(strcmp(get(handles.cb4,'enable'),'on'))
     set(handles.cb4,'value',1); end

    if(strcmp(get(handles.cb5,'enable'),'on'))
     set(handles.cb5,'value',1); end

    if(strcmp(get(handles.cb6,'enable'),'on'))
     set(handles.cb6,'value',1); end

    if(strcmp(get(handles.cb7,'enable'),'on'))
     set(handles.cb7,'value',1); end

    if(strcmp(get(handles.cb8,'enable'),'on'))
     set(handles.cb8,'value',1); end

    if(strcmp(get(handles.cb9,'enable'),'on'))
     set(handles.cb9,'value',1); end

    if(strcmp(get(handles.cb10,'enable'),'on'))
     set(handles.cb10,'value',1); end

    if(strcmp(get(handles.cb11,'enable'),'on'))
     set(handles.cb11,'value',1); end

    if(strcmp(get(handles.cb12,'enable'),'on'))
     set(handles.cb12,'value',1); end

else
    cbCheckAllToggle = 0;
    set(handles.pbCheckAll,'string','All');

    set(handles.cb1,'value',0);
    set(handles.cb2,'value',0);
    set(handles.cb3,'value',0);
    set(handles.cb4,'value',0);
    set(handles.cb5,'value',0);
    set(handles.cb6,'value',0);
    set(handles.cb7,'value',0);
    set(handles.cb8,'value',0);
    set(handles.cb9,'value',0);
    set(handles.cb10,'value',0);
    set(handles.cb11,'value',0);
    set(handles.cb12,'value',0);
end
```