IMPLEMENTATION AND EVALUATION OF A GENERAL AVIATION
SYNTHETIC VISION DISPLAY SYSTEM

A Thesis presented to

The Faculty of the
Fritz J. and Dolores H. Russ
College of Engineering and Technology
Ohio University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

By
Douglas Paul Burch
March 2004
ACKNOWLEDGEMENTS

The work presented in this thesis was funded by the NASA/FAA Joint University Program for Air Transportation Research. Len Tobias and Mike Paglione are thanked for their continuing efforts and contributions to the program and their valuable comments and suggestions throughout the course of this research.

Sincere appreciation goes to Jay Clark and Harry McCutcheon, of the Ohio University Avionics Engineering Center, for the installation and air safety certification performed during the two years of development and flight-testing of this project. Curtis Cutright is thanked for his involvement and support with this research and the Joint University Program. Sai Kalyanaraman is thanked for his contribution to this research.

Richard P. Kornfeld, R. John Hansman and John J. Deyst of the Massachusetts Institute of Technology are acknowledged for developing their method of Velocity Vector Based Attitude Determination. A special thank you to Myron Kayton for the afternoon spent discussing alternative methods for synthesizing aircraft attitude and to Sean Calhoun for helping me to understand some of the finer points of flight dynamics.

Thank you to the staff, engineers, and scientist of the Ohio University Avionics Engineering Center for their constant support and words of wisdom. All of you have contributed significantly to both my personal and professional growth during my four years with the center.
A special thank you to James Rankin and Bryan Branham for the hours spent flying and evaluating the display system presented in this thesis. Without their advice and patience the work presented here would not be as thorough or complete.

To the members of my thesis committee, thank you for your time and thoughtful input to the work presented within this thesis. To Professor Maarten Uijt de Haag, thank you for your friendship and advice along the way. To Professor Gene Kaufman, thank you for your involvement in both my undergraduate and graduate education. Your insight into mathematics and your ability to convey your knowledge and passion to your class is much appreciated. To James Rankin, thank you for your friendship and words of wisdom. “Don’t let your education get in the way of your learning”, valuable words for any student. The last four years with the Avionics Engineering Center has been a privilege.

A very special thank you goes to my advisor and friend Dr. Michael Braasch. Your guidance, support, and friendship during my time at Ohio University have been paramount to both my personal and professional growth. Thank you for providing me with a challenging and exciting project, the opportunity to travel, and your commitment to your students.
# TABLE OF CONTENTS

1 INTRODUCTION ................................................................................................................. 1

2 GENERAL AVIATION AND SYNTHETIC VISION ................................................................. 4

3 BACKGROUND ..................................................................................................................... 7
   3.1 SPATIAL DISORIENTATION ......................................................................................... 7
   3.2 COORDINATE FRAMES .............................................................................................. 9
   3.3 RADIO POSITIONING ................................................................................................. 11
   3.4 VELOCITY ESTIMATION ............................................................................................ 19
   3.5 ATTITUDE ESTIMATION ............................................................................................. 21
   3.6 AIRCRAFT INSTRUMENTATION .................................................................................. 22

4 SYNTHESIZED ATTITUDE ................................................................................................. 24
   4.1 SYNTHESIZED ATTITUDE METHOD ONE ................................................................. 24
   4.2 SYNTHESIZED ATTITUDE METHOD TWO .................................................................. 32

5 DISPLAY SYSTEM ............................................................................................................... 39
   5.1 SYSTEM OVERVIEW .................................................................................................. 40
      5.1.1 NovAtel GPS Receiver ........................................................................................ 41
      5.1.2 Attitude and Heading Reference System ............................................................ 42
      5.1.3 Data Processing Computer .................................................................................. 43
      5.1.4 Synthetic Vision and Terrain Database ............................................................... 43
      5.1.5 Display Unit ........................................................................................................ 44
   5.2 DATA COLLECTION AND PROCESSING .................................................................... 44
   5.3 SYNTHETIC VISION & GRAPHICS ENGINE ............................................................... 47
      5.3.1 Synthetic Vision Local Reference Frame .............................................................. 49
      5.3.2 Synthetic Vision Object Placement .................................................................... 50
      5.3.3 Image Rendering .................................................................................................. 53
   5.4 HEAD-UP FORWARD LOOKING DISPLAY ............................................................... 55
   5.5 PANEL MOUNTED FORWARD LOOKING DISPLAY .................................................. 58

6 INSTALLATION .................................................................................................................... 60
   6.1 PLACEMENT AND MOUNTING OF SYSTEM COMPONENTS ...................................... 60
   6.2 POWER REQUIREMENTS ............................................................................................ 64
   6.3 EGRESS ISSUES AND PILOT CONCERNS ................................................................. 65

7 FLIGHT TESTS ...................................................................................................................... 66
   7.1 FLIGHT TEST AIRCRAFT ............................................................................................ 66
   7.2 FLIGHT PROFILES ...................................................................................................... 68
   7.3 HEAD-UP DISPLAY EVALUATION ............................................................................. 68
   7.4 PANEL-MOUNTED DISPLAY EVALUATION .............................................................. 72
   7.5 PILOT EVALUATION ................................................................................................... 74
# Index

8 HEAD-UP VS. HEAD-DOWN .............................................................................. 76
9 CERTIFICATION .................................................................................................. 78
10 CONCLUSIONS AND RECOMMENDATIONS ................................................. 80
REFERENCES ........................................................................................................... 84

APPENDIX A DATA COLLECTION CODE ................................................................ 86

A.1 MAIN AND INTERRUPT SERVICE ROUTINES ................................................. 87
A.2 COORDINATE CONVERSIONS ...................................................................... 110
A.3 AHRS SCALING FUNCTIONS ........................................................................ 115
A.4 SYNTHESIZED ATTITUDE FUNCTIONS ......................................................... 116
A.5 DATA COLLECTION VARIABLES ................................................................. 135

APPENDIX B SYNTHETIC VISION CODE ............................................................... 140

B.1 MAIN FORM .................................................................................................. 141
B.2 INTERRUPT SERVICE ROUTINE FORM ......................................................... 156
B.3 DATA ABSTRACTION FUNCTIONS ................................................................. 158
B.4 SYNTHETIC VISION VARIABLES ................................................................. 164
LIST OF FIGURES

Figure 3.1 Example of Aircraft Body-frame used to describe the orientation of the Aircraft against the Local Horizontal Reference Plane .................................................. 11

Figure 3.2 Graphical Explanation of the Ordinary Least Squares Iterative Solution ................................................................................................................................... 13

Figure 4.1 Treatment of the Velocity Vector.......................................................................................................................... 25

Figure 4.2 Free-body Diagram Showing the Formation of the Lift Vector............ 27

Figure 4.3 Free-body Diagram Showing the Relationship between the Lift Vector and the Local-level Reference Plane ........................................................................... 30

Figure 4.4 Synthesized Roll, Pitch, and Ground track Angle Produced by Method One ................................................................................................................................ 32

Figure 4.5 Free-body Diagram used to sum the Forces along the y-body Axis........ 34

Figure 4.6 Synthesized Roll, Pitch, and Ground track Angle Produced by Method Two......................................................................................................................... 36

Figure 4.7 Comparison of the Synthesized Roll produced by Method One (Top) and Two (Middle), and the Difference (Bottom) ................................................................. 37

Figure 5.1 Real World Compared to Virtual World ................................................................. 40

Figure 5.2 System Schematic of General Aviation Synthetic Vision Display System 41

Figure 5.3 ENU Reference Frame used by the Graphics Engine ......................... 49

Figure 5.4 Runway Placement within Synthetic Vision Local Reference Frame .... 51

Figure 5.5 Explanation of Precision Approach Path Indicator (PAPI) ...................... 52

Figure 5.6 Portion of Gray Scale Bit Map Terrain Mesh and Color Plate .............. 53

Figure 5.7 Panoramic Sky Photograph used within the Rendered Image ............. 54

Figure 5.8 Image Rendered by the Graphics Engine ...................................................... 54
Figure 6.1 Equipment Rack used to House the GPS Receiver, AHRS, Data Processing Computer, Power Supply, and User Interface ........................................ 61

Figure 6.2 LCD Projector Mounting ............................................................................. 63

Figure 6.3 Combiner Glass Placed Perpendicular to the Subject Pilot's Field of View ........................................................................................................ 63

Figure 6.4 Panel-mounted Forward-looking Display Strategically Mounted within Range of the Pilot's Peripheral Field of View .................................................. 64

Figure 7.1 Piper Saratoga (PA-32-301) used for Flight Testing of Synthetic Vision Display ........................................................................................................ 67

Figure 7.2 Display System During Climb Out .............................................................. 70

Figure 7.3 Initial Approach on Runway 25 ................................................................. 70

Figure 7.4 Display System during Final Approach on Runway 25 .......................... 71

Figure 7.5 Panel-mounted Display during Final Approach on Runway 7 ............ 73

Figure A. 1 Pseudo-code for the Main Routine used for Data Collection ......... 86

Figure B. 1 Pseudo-code for the Main SVS Routine ........................................... 140
1 Introduction

On July 16\textsuperscript{th}, 1999, at approximately 2038 hrs, John F. Kennedy Jr. departed from Essex County Airport in Fairfield, New Jersey in his Piper Saratoga. Conditions were reported as hazy with poor visibility but still within the limits for flight under Visual Flight Rules (VFR). At 2130 hours, roughly one hour into the flight, the Piper Saratoga disappeared from radar. The plane crashed into the Atlantic Ocean off the coast of Massachusetts and Rhode Island killing Mr. Kennedy and his two passengers. It is believed that John F. Kennedy Jr. was suffering from spatial disorientation. He was no longer able to control the aircraft’s flight path due to his visual deprivation resulting in the loss of the aircraft and all lives on board.

Spatial disorientation related accidents are often a direct result of visual flight into Instrument Meteorological Conditions (IMC). In the opening remarks of the 1999 Nall Report, the Executive Director of the AOPA Air Safety Foundation states that “Visual flight into Instrument Meteorological Conditions (IMC) and low-level maneuvering flight continue to be the two areas of flight producing the largest number of fatal accidents” [2]. In the Executive Overview for the 2001 Nall Report he states “In general terms, little has changed significantly since last year. Low-level maneuvering flight and weather remain the two largest fatal accident producers. [1]” These remarks bring to light the importance of synthetic vision systems (SVS), for not only commercial aircraft, but for general aviation cockpits.
Due to the fact that visual flight into IMC is one of the most frequent causes of accidents resulting in fatalities, one question keeps coming up. The question that GA authorities are asking is “what is it about the fact that they can no longer see the ground that pilots don’t understand?” [2]. The question is not “what does a pilot not understand?” but “what is the pilot not able to sense?” This lack of spatial orientation and inability to understand and control the aircraft’s trajectory is a direct result of not being able to visually reference the local horizon and the surrounding terrain.

Precise attitude determination is not usually performed in general aviation aircraft due to the cost of attitude determination sensors. This is rapidly changing with the introduction of Microelectromechanical System (MEMS) devices that are capable of providing precise, real-time attitude information [3]. This attitude information, coupled with GPS-derived position information, is capable of driving a low-cost synthetic vision display. It has already been determined that this type of display dramatically increases a pilot’s understanding of his or her environment [4]. The goal of this research is to develop an equipment sensor suite specifically designed for a general aviation aircraft and to evaluate two possible display configurations; a see-through forward-looking head-up display and a forward-looking head-down display mounted within the instrument panel. It is believed that a see-through forward-looking head-up display would reduce the pilot’s workload by providing a virtual “out the window view”, thus allowing the pilot to focus on an intuitive visual approach. The same holds true for the forward-looking head-down display if mounted strategically within the instrument panel. These two configurations will be implemented and evaluated based on size and placement, power
requirements, system costs, technical implementation, and initial pilot evaluation. The cost of certifying this technology is of great importance to the general aviation community. A cost analysis for the certification of this system was not performed due to the fact that the focus of this research was to design and implement a GA specific system in an experimental aircraft.
2 General Aviation and Synthetic Vision

Synthetic Vision has been of great interest to the aviation community since its conception. The idea of providing a pilot with a virtual view of the outside world has the potential to save countless lives if implemented correctly. The majority of research in this area has focused on commercial and military applications whose sponsors and end users can afford the resulting system cost. Unfortunately, the pilots who need this technology the most (general aviation) are left using “trickle-down” applications that are still priced well over $30,000 (U.S.). One of the main reasons for the high cost associated with synthetic vision displays is not the computing power needed to render the display, but the sensors used to drive the display and the detailed certification process of the entire equipment suite.

The aircraft state-vector information, measured by the sensor suite, is needed to drive the computer-generated rendering of the outside world. The state-vector consists of, but is not limited to, the attitude and position information of the aircraft. The position information allows the computer to place the aircraft at the correct location in the computer-generated rendering and the attitude information configures the image in the correct orientation to the locally level horizontal reference plane.

Positioning information is usually estimated using a Global Positioning System (GPS) receiver capable of meeting the demands of the aircraft dynamics under consideration. The attitude information can be synthesized from the GPS velocity vector if the aircraft
state is sampled often enough and coordinated flight is assumed or it can be estimated from a combination of gyroscopes and accelerometers.

Both of the aforementioned methods are costly, but for different reasons. The synthesized attitude is an approximation of the aircraft orientation used under the assumption of coordinated flight. When the aircraft is not in coordinated flight, this information can be misleading and dangerous to the pilot. It is conceivable that this misinformation could result in the loss of the aircraft. The traditional implementation of gyroscopes and accelerometers used to form a navigation-grade Inertial Navigation System (INS) can cost more than $100,000. General aviation applications require that the system provide robust and dependable information at an affordable cost.

What is stopping the development of synthetic vision systems for general aviation applications? Nothing. There are a multitude of companies and universities involved in the research and development of synthetic vision. Unfortunately, synthetic vision research involves more than just developing the display software. It involves developing the system from the ground up for general aviation applications. This means using sensors that are geared toward GA applications both in flight dynamics and cost.

Regardless of the system cost, it is important to establish guidelines for the intended use and role synthetic vision will have in the general aviation community. The intended use of the display system, and the pilots for whom it was designed, are important issues that must be addressed. Currently, there are several forward-looking panel-mounted synthetic vision displays targeted for the low-end general aviation market. It is very tempting for the low-time non-instrument rated pilot to purchase one of these displays
and use it to push the envelope when racing against weather fronts. It is possible for a non-instrument rated pilot to plan a VFR flight and then use this type of technology in an attempt to safeguard his or her travel plans against poor weather. This is a very dangerous practice. This technology was developed with the low-time instrument rated pilot in mind to enhance spatial and situational awareness in IMC. The display system was designed to complement current navigation aids, not replace them. This device is presented as an aid to help the low-time instrument-rated pilot conduct precision and non-precision approaches using the current Instrument Landing System augmented by the latest in display technology and positioning and attitude sensors.
3 Background

Synthetic vision research for general aviation applications involves an understanding of a wide variety of engineering topics. This research also requires a basic understanding of general aviation operations and the many human factors associated with the pilot and the mechanical interface of the aircraft. Chapter three provides the necessary background to follow the research and development of the synthetic vision display and supporting architecture designed, developed, and implemented by the Avionics Engineering Center at Ohio University.

3.1 Spatial Disorientation

Spatial disorientation is defined as “any conditions, which deprive the pilot of natural visual references to maintain orientation, such as clouds, fog, haze, darkness, or terrain/sky backgrounds with indistinct contrast (such as arctic whiteout or clear moonless skies over water [at night])” [2]. A pilot’s sense of orientation is based on information originating from the inner ear. The information relayed from the inner ear is very accurate if the body is experiencing little or no motion. If the body is moving at a higher velocity, visual cues are required to augment the information from the inner ear so that the brain is able to keep track of the body’s orientation. If the pilot is deprived of this visual information, as when entering a cloudbank, he or she will experience mild to strong impressions of spatial disorientation. The 2001 Nall Report states, “Because these
false impressions are based on physics and a basic aspect of human physiology, they cannot be avoided by training and can, therefore, affect pilots of all experience levels [1].” A study conducted from 1987 to 1996 concluded that one fatal spatial disorientation accident occurred every eleven days over this ten-year period [2].

The introduction to this research is a tragic reminder of the price general aviation pilots are paying by not having advanced technologies available to them. The average low-time instrument-rated pilot relies heavily on visual cues from the real world to help fly the aircraft. When this low-time instrument-rated pilot enters a weather front or any other conditions that may deprive them of this visual reference they must rely on instrumentation that can be confusing and easily misinterpreted. This misunderstanding of the instrumentation in a general aviation aircraft is due to the fact that the pilot is suffering from spatial disorientation and does not trust the instruments over his or her own instincts. Once an aircraft settles into a coordinated turn, the body does not feel acceleration and therefore cannot accurately determine the attitude and trajectory of the aircraft without visual cues. These visual cues can be interpreted from the current instrumentation, but if the pilot does not have the necessary experience to build a visual image of what the aircraft is doing, then the results can be fatal. This adverse effect of spatial disorientation is why IMC continues to be one of the leading killers of general aviation pilots.
3.2 Coordinate Frames

The position of an object is always defined relative to some other larger or more distinguishable object. For instance if one were to define the location of a book on his or her desk, the location might be 12 inches to the front and another 12 inches to the left of that person. The position of the book could also be described as approximately 17 inches from the person at a 45° angle relative to the body axis of the individual. Both descriptions of the book’s position imply that a frame of reference is necessary to describe the position of an object. By stating that the book is 12 inches to the front and 12 inches to the left of the person sitting at the desk we make the assumption that the person sitting at the desk is the point of reference or origin. It is not convenient to have people describe the position of an object relative to them, so a central reference frame is necessary.

The notion of a central reference frame is even more crucial when describing the position of a celestial body relative to the earth or when defining the location of an object near the earth’s surface. To locate an object on or near the earth all parties must not only use a central reference frame, but also agree on the definition of that reference frame. The reference frame will be of little use if it is not common to all parties involved.

For determining the position of an object on or near the earth’s surface a Cartesian reference frame can be chosen with its origin located at the center of the earth and whose axis rotate with the earth. This coordinate system is conveniently called an Earth-Centered-Earth-Fixed (ECEF) coordinate frame [5]. The ECEF coordinate frame allows
for the position of an object to be described relative to the earth and to other objects whose positions are known.

The position and velocity of an object can be calculated using the ECEF coordinate frame, but the position really does not mean much to the average person. For instance, if a person were told to start at a given ECEF position, meaning an (X, Y, Z) coordinate, and travel to another ECEF position, the directions would be meaningless and the person would be lost even though they knew their precise location and the desired destination. On the other hand, if a person were given the directions relative to an object they knew, they could make sense of the information provided. This leads to the notion of an East, North, and Up (ENU) coordinate frame. An ENU coordinate frame is a right-hand coordinate frame with its origin located at a specific point within the local area. This origin could be a flagpole for children playing in a schoolyard, or the end of a runway for a pilot attempting to land an aircraft on final approach. The ENU coordinate frame simply allows for one to navigate relative to a fixed location. Even if they cannot see the reference point, the measurements relative to that reference point can be easily interpreted.

Another type of coordinate frame used to describe the position of a body in three-dimensional space is called the body-frame and typically has the convention of nose, right wing, and down. The body-frame is used to describe the orientation of the body using the frame relative to its Center of Gravity (CG). This can be seen in Figure 3.1 with the $X_{body}$ axis oriented through the nose of the aircraft and the $Y_{body}$ axis oriented through the right side of the aircraft perpendicular to the $X_{body}$ axis. The $Z_{body}$ axis is formed by
taking the vector cross product of the $X_{body}$ axis into the $Y_{body}$ axis yielding a vector oriented out the bottom of the aircraft.

![Diagram of aircraft body-frame](image)

**Figure 3.1 Example of Aircraft Body-frame used to describe the orientation of the Aircraft against the Local Horizontal Reference Plane**

### 3.3 Radio Positioning

The primary purpose of a radio positioning system is to approximate the current position of a user within a well-defined coordinate frame [5]. Several types of radio positioning systems exist today including Loran, the Russian Global Navigation Satellite System (GLONASS), and the Global Positioning System (GPS) developed by the United States Department of Defense. GPS is one of the most utilized radio positioning systems in existence today due to its global coverage, and low-cost receivers. GPS receivers are accessible to the general public for a variety of applications and are also used for more complex applications such as geodetic surveying and aircraft navigation.

It is important to remember that a GPS receiver is not a navigation device; it merely provides position, velocity, and time. However, a GPS receiver can be incorporated into a system to provide relative positioning with respect to a known point within a well-
defined coordinate frame. The relative position of the GPS user can then be used with respect to the known reference point to guide the GPS user to a desired location or make precise measurements with respect to the GPS user’s location. Keep in mind that the location of the GPS user is with respect to the phase center of the GPS receiver’s antenna.

The GPS satellite constellation is well documented and will not be discussed in the following text. However, the ordinary-least-squares iterative solution used to approximate the position of a near-earth GPS user will be examined in detail. This should provide the reader with a better understanding of the accuracy limitations of the Global Positioning System and why certain limitations are set for the sensor suite and display technology discussed in this research.

**GPS Ordinary-Least-Squares Iterative Solution**

The GPS receiver solves for its position solution by using the observed ranging measurements and the known GPS satellite positions. This sounds like a simple geometric problem known as trilateration but due to the non-linearity of the range equation (3.1) the calculation soon becomes very intensive [5].

\[
PR = \sqrt{(X_{user} - X_{SV})^2 + (Y_{user} - Y_{SV})^2 + (Z_{user} - Z_{SV})^2} + B \quad (3.1)
\]

The range from the GPS satellite, or space vehicle (SV), to the user is called the pseudorange (PR) because it is the sum of the true range and the range equivalent of the user’s clock bias (B). An additional term is added to the standard range equation to
account for the range equivalent of the user’s clock bias. The clock bias is multiplied by the speed of light to transform the offset from the time domain to the range domain.

Equation (3.1) has four unknown variables: $X_{user}$, $Y_{user}$, $Z_{user}$, and the range equivalent of the user’s clock bias ($B$). Clearly, four measurements are needed to solve this set of non-linear equations. In order for equation (3.1) to be solved it is usually linearized about a given point and the higher-order-terms disregarded. By keeping only the 0th and the 1st order terms the equation is now a linear approximation that allows the set of equations to be solved for the $X$, $Y$, and $Z$ position of the user as well as the range equivalent of the user’s clock bias. This set of equations is solved using an iterative process due to the fact that the least squares solution provides the difference between the initial estimate of the user position and the true user position (Figure 3.2).

Figure 3.2 Graphical Explanation of the Ordinary Least Squares Iterative Solution
The simple explanation of the least-squares solution is described in equation (3.2) where $\mathbf{R}_m$ is the measured user position vector and $\mathbf{R}_{est}$ is the estimated user position vector [5].

$$PR_i = PR_{io} + \partial PR_i \left|_{X_0Y_0Z_0B_0} \left( \mathbf{R}_{est} - \mathbf{R}_m \right) \right.$$ (3.2)

The measured pseudorange ($PR_i$) from the GPS user to a given SV in view is taken from the receiver measurements. The calculated pseudorange ($PR_{io}$) is a vector from the GPS satellite to the initial user-position-estimate (for convenience located at the center of the earth $[0,0,0]$ with the range equivalent of the user’s clock bias set to zero, Figure 3.2). The actual expression for this linearization is found in equation (3.3).

$$PR_i = PR_{io} + \left( X - X_o \right) \left( \frac{\partial PR_i}{\partial X} \right) + \\
\left( Y - Y_o \right) \left( \frac{\partial PR_i}{\partial Y} \right) + \\
\left( Z - Z_o \right) \left( \frac{\partial PR_i}{\partial Z} \right) + \left( B - B_o \right) \left( \frac{\partial PR_i}{\partial B} \right)$$ (3.3)

The ordinary least squares term we are solving for is a 4x1 column vector containing the elements $\partial X$, $\partial Y$, $\partial Z$, and $\partial B$:

$$\partial X = X - X_o$$

$$\partial Y = Y - Y_o$$

$$\partial Z = Z - Z_o$$

$$\partial B = B - B_o$$
The pseudorange is equal to the range equation plus the user clock bias scaled by the speed of light. The partial derivatives of the terms in equation (3.3) are as follows [5]:

\[
\frac{\partial PR_i}{\partial x} \bigg|_{X_o Y_o Z_o B_o} = \frac{1}{2} \left[ (X_o - X_i)^2 + (Y_o - Y_i)^2 + (Z_o - Z_i)^2 \right]^{1/2} \frac{1}{2} (X_o - X_i)
\]

\[
\frac{\partial PR_i}{\partial y} \bigg|_{X_o Y_o Z_o B_o} = \frac{1}{2} \left[ (X_o - X_i)^2 + (Y_o - Y_i)^2 + (Z_o - Z_i)^2 \right]^{1/2} \frac{1}{2} (Y_o - Y_i)
\]

\[
\frac{\partial PR_i}{\partial z} \bigg|_{X_o Y_o Z_o B_o} = \frac{1}{2} \left[ (X_o - X_i)^2 + (Y_o - Y_i)^2 + (Z_o - Z_i)^2 \right]^{1/2} \frac{1}{2} (Z_o - Z_i)
\]

\[
\frac{\partial PR_i}{\partial B} \bigg|_{X_o Y_o Z_o B_o} = 1
\]
Using equation (3.1) we can simplify the partial derivatives:

\[
\left. \frac{\partial PR_i}{\partial x} \right|_{X_{0Y_{0Z_{0}}} - cB_{0}} = \frac{X_o - X_i}{PR_{io} - cB_o} \equiv \alpha_{i1} \quad \left. \frac{\partial PR_i}{\partial y} \right|_{X_{0Y_{0Z_{0}}} - cB_{0}} = \frac{Y_o - Y_i}{PR_{io} - cB_o} \equiv \alpha_{i2} \\
\left. \frac{\partial PR_i}{\partial z} \right|_{X_{0Y_{0Z_{0}}} - cB_{0}} = \frac{Z_o - Z_i}{PR_{io} - cB_o} \equiv \alpha_{i3} \quad \left. \frac{\partial PR_i}{\partial B} \right|_{X_{0Y_{0Z_{0}}} - cB_{0}} = 1 \equiv \alpha_{i4}
\]

Each of the partial derivatives has been set equal to an \( \alpha \) term in order to simplify the linearized form of the range equation.

The least squares expression has now been linearized about the initial estimate of the user’s position. A set of simultaneous equations can be set up using the minimum coverage of four satellites to solve for the user \( X, Y, Z, \) and clock bias. Ideally the linear expression is shown in equation (3.4) with \( \epsilon \) representing the measurement error.

\[
\bar{Y} = H\bar{\beta} + \bar{\epsilon} \quad (3.4)
\]

Equation (3.4) has the following form assuming the minimum number of satellites is used when solving for the user’s position [5].

\[
\begin{bmatrix}
\partial PR_1 \\
\partial PR_2 \\
\partial PR_3 \\
\partial PR_4
\end{bmatrix} =
\begin{bmatrix}
\alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{14} \\
\alpha_{21} & \alpha_{22} & \alpha_{23} & \alpha_{24} \\
\alpha_{31} & \alpha_{32} & \alpha_{33} & \alpha_{34} \\
\alpha_{41} & \alpha_{42} & \alpha_{43} & \alpha_{44}
\end{bmatrix}
\begin{bmatrix}
\partial X \\
\partial Y \\
\partial Z \\
\partial B
\end{bmatrix} +
\begin{bmatrix}
\epsilon_1 \\
\epsilon_2 \\
\epsilon_3 \\
\epsilon_4
\end{bmatrix}
\]
It is important to note in the geometry matrix (H) the variables denoted by $\alpha_{i1}$ through $\alpha_{i3}$ form a unit vector pointing from the GPS user to a visible GPS satellite. The final approximation of this set of unit vectors is valid after the last iteration of the GPS ordinary least-squares position solution has been completed.

The ordinary least squares solution of (3.4) is shown in equation (3.5).

$$\overline{\beta} = \left( H^T H \right)^{-1} H^T \overline{Y}$$ (3.5)

This is not the user position; this is the offset that is added to the initial estimate of the user position. This provides an updated user position estimate and the entire process can then be iterated. The only real change from one iteration to the next is the linearization about the new user position estimate, which changes the geometry matrix (H). The value of X, Y, Z, and the range equivalent of the user’s clock bias are updated by adding the position difference onto the user estimate until the absolute difference between the current and previous estimated position is below a given threshold (e.g. 1.0 cm).

If this algorithm were to be implemented the user would receive an erroneous answer without accounting for the correct time of transmission. The signal time of transmission (TOT) is estimated by subtracting the signal transit time from the time of reception (TOR), equation (3.6). As stated previously, the GPS user position solution is dependent on the range measurements and the precise location of the GPS satellites. If the GPS receiver were to use the time of reception to calculate the satellite positions the resulting answer would describe where the satellite is at that time and not where it was when the
signal was transmitted. It is important to know exactly where the satellite was when it transmitted the information the user is currently receiving.

\[ TOT = TOR - (SignalTransitTime) \]

\[ TOT = TOR - \left( \frac{PR}{c} - User\text{ClockBias} + SV\text{ClockBias} \right) \]  \hspace{1cm} (3.6)

The approximated time of transmission is then used to calculate the position of each GPS satellite in view. Once the correct time of transmission has been calculated, each satellite position must be adjusted for the earth’s rotation during the time it takes the GPS signal to travel from the satellite to the user near the earth’s surface. The X, Y, and Z coordinates in Earth Centered Earth Fixed (ECEF) reference frame are rotated or transformed by a Direction Cosine Matrix (DCM) about the Earth’s Z-axis, equation (3.7). The corrected satellite position in ECEF \((X', Y', Z')\) is then used to solve for the user position.

\[
\begin{bmatrix}
X' \\
Y' \\
Z'
\end{bmatrix} =
\begin{bmatrix}
\cos \phi & \sin \phi & 0 \\
-\sin \phi & \cos \phi & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
X \\
Y \\
Z
\end{bmatrix}
\]  \hspace{1cm} (3.7)

The angle used in the DCM is calculated using the earth’s rotational rate \((\Omega)\) multiplied by the corrected GPS signal transit time, equation (3.8).

\[ \phi = (Signal\text{TransitTime}) (\Omega_E) \]

\[ \phi = \left( \frac{PR}{c} - User\text{ClockBias} + SV\text{ClockBias} \right) (\Omega_E) \]  \hspace{1cm} (3.8)
If the earth’s rotation during the GPS signal transit time is not accounted for, the GPS user will have moderate errors in their position solution. The amount of error experienced by the user is dependent on the latitude of the user relative to the earth’s equator. A GPS user at the earth’s equator will experience approximately 40 meters of error while a user at either pole will experience little error due to the earth’s rotation.

The measured Pseudo-Range (PR) observed by the receiver is adjusted to account for the satellite clock bias. The satellite clock bias must be added to the measured Pseudo-Range before it can be used to solve for the user position, equation (3.9).

\[ PR' = PR + SV_{ClockBias} \]  

### 3.4 Velocity Estimation

To estimate the velocity of a GPS user, the relative change in frequency of the GPS signal is used [5]. This change in frequency is known as the Doppler shift and is caused by the motion of the GPS satellite and potentially the motion of the GPS user.

The Doppler shift is the difference between the true frequency of a signal in space (transmitted frequency) and the perceived frequency at a given point (received frequency). This relationship between the transmitted frequency and the apparent received frequency is expressed in equation (3.10):

\[ (f_R - f_T) = -\frac{\dot{r}}{\lambda} \]  

(3.10)
Where $f_R$ is the received frequency, $f_T$ is the transmitted frequency, $r$ is the range between the transmitter and the receiver and $\lambda$ is the wave length associated with the frequency and speed ($v_s$) of the transmitted waveform at the source [5]. Equation (3.10) can be rewritten as follows to better show this relationship:

$$
f_R = f_T \left(1 - \frac{r}{v_s}\right)$$

(3.11)

The Doppler shift is measured in the GPS receiver by the carrier-tracking loop [5]. The velocity of the GPS satellite can be computed from the broadcast orbital parameters and is used with the pseudorange rate to solve for the user velocity vector. This relationship can be expressed as a linear equation in matrix notation as:

$$
\dot{PR} = H \begin{bmatrix} \dot{\vec{v}} \\ \dot{\vec{b}} \end{bmatrix} + \vec{\epsilon}.
$$

(3.12)

The pseudorange rate is represented by the time derivative of pseudorange. The user geometry matrix ($H$) is the user to satellite unit vectors described in section 3.3. The rate of the receiver clock bias and the 3-dimensional user velocity vector can then be solved using an ordinary least squares solution by assuming that the error in the pseudorange rate is zero.
3.5 Attitude Estimation

Three primary parameters are necessary to describe the attitude of an aircraft; roll, pitch, and yaw. These parameters can be estimated in a number of ways including gyroscopes and accelerometers, or synthesized from the velocity vector.

Through the course of this research, two methods of attitude determination were used: synthesized attitude from the velocity vector and the attitude provided from an attitude and heading reference system (AHRS). The remainder of this section will focus on the fundamental operations of the AHRS that was ultimately used to drive the synthetic vision display. The formulation and flight-testing results of the synthesized attitude will be presented in chapter four.

The AHRS used to provide the attitude of the aircraft to the synthetic vision display is the Crossbow© AHRS400. This AHRS unit is capable of measuring stabilized roll, pitch, and yaw angles at rates up to 60 Hz. This AHRS unit is a 9-axis measurement system that uses 3-axis linear accelerometers, 3-axis rotational rate sensors, and magnetometers with 3-axis accelerometers and 3-axis rate sensors to measure the dynamics of the aircraft. The angular rate sensors are micro-machined devices that consist of vibrating ceramic plates that use Coriolis force to measure the angular rate independent of acceleration [3]. The acceleration is measured by three micro electrical mechanical system (MEMS) accelerometers that use differential capacitance. Magnetic fluxgate sensors measure the heading. These measurements are then placed into a Kalman filter that provides estimates of the aircraft attitude through dynamic maneuvers [3].
3.6 Aircraft Instrumentation

Current general aviation instrumentation utilizes a combination of gauges and dials to provide the pilot with the aircraft state-vector information. The pilot then uses the information from this set of gauges and dials to build a visual image of the aircraft velocity, attitude, and altitude. During VFR conditions the pilot is primarily concerned with the altitude and heading of the aircraft. However, during IFR the pilot must be able to use the information from the attitude indicator, turn coordinator, heading indicator, airspeed indicator, altimeter, magnetic compass, and vertical speed indicator to control and safely navigate the aircraft [6]. Additional instrumentation that is crucial to the pilot during the landing phase of flight is the localizer and glide slope indicators that provide information from the instrument landing system (ILS).

The instrument landing system is an integrated radio frequency system that transmits signals from fixed points at a given runway for the purpose of guiding aircraft through category I, II, and III landings, depending on pilot certification. The instrument landing system was made commercially available in 1939 and is still the primary method for landing aircraft today [7].

The ILS uses two primary transmitters for aircraft guidance: the localizer and the glide slope. The localizer antenna is centered on the stop end of the runway providing the aircraft with lateral guidance information with respect to the center of the runway. The glide slope antenna is placed beside the runway near the touchdown point. The glide slope signal provides the aircraft with vertical guidance along a three-degree flight path extending from the touchdown point [7].
When conducting a landing during VMC the pilot relies on visual cues from the outside world to land the aircraft and uses information from the aircraft instrumentation, including the localizer and glide slope indicators, as a cross check. During an instrument approach, the pilot relies entirely on the aircraft instrumentation to build a visual image of the aircraft velocity, attitude, altitude, and position of the aircraft with respect to the runway.
4 Synthesized Attitude

The attitude of an aircraft can be approximated from its velocity vector assuming the aircraft is in coordinated flight. This method of attitude determination can be very cost-effective and computationally inexpensive for general aviation due to the low-dynamic flight and operational uses of GA aircraft.

In the beginning stages of this research the attitude information was approximated from the velocity vector using a technique developed at the Massachusetts Institute of Technology International Center for Air Transportation, Department of Aeronautics and Astronautics. Several flight tests with successful results were conducted using this method of attitude approximation. Later a second method of attitude approximation, suggested by Myron Kayton, was used in the research. The algorithms for attitude approximation will be referred to as methods one and two respectively. Due to the fact that velocity vector based attitude determination is only valid in the assumption of coordinated flight, the algorithm was eventually replaced with an AHRS unit. However, velocity vector based attitude determination could be utilized in the future to augment or provide back up to the AHRS.

4.1 Synthesized Attitude Method One

In 1999, Dr. Kornfeld, Dr. Hansman, and Dr. Deyst developed a method for velocity vector based attitude determination [8]. This method of attitude determination provides a
cost-effective and computationally inexpensive means of approximating an aircraft’s attitude.

To use the velocity vector to derive the attitude, it is assumed the aircraft is in coordinated flight with the velocity vector \( \mathbf{v}_g \) in very close proximity to the x-body axis, which extends from the center of gravity (CG) through the nose of the aircraft. It is required that the GPS antenna be placed near the aircraft’s CG in order to provide the velocity of the aircraft as if it were a particle traveling through free space (Figure 4.1).

![Diagram of aircraft with GPS antenna and coordinate axes](image)

**Figure 4.1 Treatment of the Velocity Vector**

While an aircraft is in a coordinated turn the lift force is acting against two forces: the gravitational acceleration and the normal acceleration multiplied by the mass of the aircraft. The lift force is perpendicular to the y-body axis of the aircraft but the orientation of the body frame is unknown. The normal gravitational acceleration vector can be calculated using an assumed model of gravity and the normal acceleration of the aircraft can be derived from the velocity vector. The vector difference between the normal gravitational acceleration vector and the normal acceleration vector of the aircraft
provides an approximation of the magnitude and direction of the normalized lift vector. This approximation of the normalized lift vector can be referenced against the local horizontal plane to synthesize the roll (φ) of the aircraft [8].

The flight path angle (γ) is used to approximate the pitch of the aircraft. The GPS receiver provides the horizontal speed and vertical speed of the aircraft in meters per second. The flight path angle is the inverse tangent of the vertical speed divided by the magnitude of the horizontal speed.

A free-body diagram is created to derive the equations needed in the algorithm for solving for the instantaneous attitude of the aircraft. This diagram assumes that we are viewing the aircraft CG from the tail of the aircraft (Figure 4.2). Thus the velocity vector with respect to the ground is pointing into the page. From this free-body diagram it can be shown that the normalized lift-vector, represented by the lower-case letter ‘l’, is the vector difference between the normal acceleration vector (\(a_n\)) and the normal gravitational acceleration vector (\(g_n\)).

To find the value of the normal acceleration vector of the aircraft the total acceleration vector (\(a_g\)) with respect to the ground is found by differentiating the velocity vector (\(v_g\)).

\[
\vec{a}_g = \frac{d\vec{v}_g(t)}{dt} \tag{4.1}
\]
The tangential acceleration vector \( \mathbf{a}_t \) is found by projecting the total acceleration vector \( \mathbf{a}_g \) onto the velocity vector (which is assumed to be tangential to the flight path of the aircraft in the coordinated turn) (equation 4.2).

\[
\mathbf{a}_t = \frac{\mathbf{a}_g \cdot \mathbf{v}_g}{|\mathbf{v}_g|^2} \mathbf{v}_g
\]  

The normal acceleration vector \( \mathbf{a}_n \) is then found by the vector difference between the total acceleration vector \( \mathbf{a}_g \) and the tangential acceleration vector \( \mathbf{a}_t \) (equation 4.3).

\[
\mathbf{a}_n = \mathbf{a}_g - \mathbf{a}_t
\]
The normal gravitational acceleration vector \( \mathbf{g}_n \) is found similarly by projecting the assumed gravitational vector \( \mathbf{g} \) onto the tangential velocity vector (equation 4.4) and then subtracting the tangential gravitational acceleration \( \mathbf{g}_t \) from the assumed gravitational vector to estimate the normal gravitational acceleration vector \( \mathbf{g}_n \) (equation 4.5).

\[
\mathbf{g}_n = \mathbf{g} - \mathbf{g}_t
\]

Now that the values of \( \mathbf{g}_n \) and \( \mathbf{a}_n \) have been approximated their vector difference can be taken to form the normalized lift vector that is perpendicular to the y-body axis (equation 4.6, Figure 4.2).

\[
\mathbf{l} = \mathbf{a}_n - \mathbf{g}_n
\]
(\(P\)) to be calculated (Figure 4.3). The local-level reference plane is approximated by taking the cross product of the estimated normal gravitational acceleration vector \((\vec{g}_n)\) and the tangential velocity vector \((\vec{v}_p)\) (equation 4.7).

\[
\vec{P} = \vec{g}_n \times \vec{v}_g
\]  
(4.7)

The synthesized roll-angle (\(\varphi\)) is then calculated by taking the arcsine of the projection of the normalized lift vector onto the vector representing the local-level reference plane (equation 4.8).

\[
\varphi = \sin^{-1}\left( \frac{\vec{l} \cdot \vec{P}}{\|\vec{P}\|} \right)
\]  
(4.8)

The pitch of the aircraft is approximated by the flight path angle (\(\gamma\)). The calculation of the flight path angle from the velocity vector is shown in equation 4.9. The velocity vector is described in the North, East, and Down (NED) reference frame.

\[
\gamma = \tan^{-1}\left( \frac{-\vec{v}_{gd}}{\sqrt{\vec{v}_{gN}^2 + \vec{v}_{gE}^2}} \right)
\]  
(4.9)
Figure 4.3 Free-body Diagram Showing the Relationship between the Lift Vector and the Local-level Reference Plane

The ground track angle is provided in the GPS velocity string from the NovAtel GPS receiver. To complete the derivation of the velocity vector based attitude determination, equation 4.10 shows the formation of the ground track angle. The velocity vector component in the East direction is given by $v_{gE}$ and the velocity vector component in the North direction is given by $v_{gN}$. The sign associated with the North and East components of the velocity vector are used to remove quadrant ambiguity in order to determine the correct ground track angle with respect to North.

$$
\Psi_r = \tan^{-1}\left(\frac{v_{gE}}{v_{gN}}\right) \quad (4.10)
$$
The synthesized roll ($\phi$), flight path angle ($\gamma$), and ground track ($\Psi_T$) make up the approximation of the aircraft attitude (Figure 4.4). The primary concern regarding this method of attitude approximation is due to the dependency on the velocity vector being in very close proximity to the x-body axis of the aircraft (i.e. coordinated flight). Due to the low-dynamic environment that GA aircraft tend to fly, the major deviation from coordinated flight occurs during sideslip. When the aircraft is experiencing sideslip, the velocity vector is no longer pointing out the nose of the aircraft and the ground track is no longer representative of the yaw angle. This misrepresentation can be beneficial in a head-down implementation as it automatically steers the synthetic vision display in the direction of travel. A head-up implementation will suffer display discrepancies during severe sideslip if the ground track is used to approximate yaw.

The true limitation of this method of attitude determination occurs when the aircraft enters a stall condition or high angles of attack. When the velocity vector deviates from the x-body axis of the aircraft the pseudo-roll and flight path angle no longer provide accurate approximations of the true roll and pitch.

Figure 4.4 shows the roll, pitch (flight path angle), and ground track angle from a flight test conducted on the 18th of November 2001. The noise introduced by the differentiation of the velocity vector is apparent on the roll angle. The pitch (flight path angle), which depends on the velocity vector without differentiation, is relatively smoother than the roll. The ground track angle, measured directly by the GPS receiver, is very smooth providing a stable approximation of heading.
4.2 Synthesized Attitude Method Two

The second method for velocity vector based attitude determination does not rely on the differentiation of the velocity vector and is therefore less noisy. This derivation needs only the ground track angle, which is the assumed heading, and the velocity vector. The ground track angle is calculated from the velocity vector as shown in equation 4.10. The flight path angle is calculated as shown in equation 4.9. The fundamental difference between method one and method two lies in the calculation of the synthesized roll angle.
Looking at the free-body diagram shown in Figure 4.5, the sum of the forces along the y-body axis is expressed as follows:

\[
\sum F_{y-body} = |\vec{a}_n| m \cos(\phi) - |g| m \sin(\phi) = 0 \tag{4.11}
\]

Manipulating equation 4.11 yields the following:

\[
|\vec{a}_n| \cos(\phi) = |g| \sin(\phi) \tag{4.12}
\]

\[
|\vec{a}_n| = \frac{|g| \sin(\phi)}{\cos(\phi)} = |g| \tan(\phi) \tag{4.13}
\]

From the basic equations of plane circular motion:

\[
|\vec{a}_n| = \frac{\vec{v}^2}{R} \tag{4.14}
\]

The variable R represents the radius of the coordinated turn at any given instant in time. Substituting equation 4.14 into equation 4.13 yields:

\[
\frac{\vec{v}^2}{R} = |g| \tan(\phi) \tag{4.15}
\]
The differentiation of the ground track ($\psi_T$) is equal to the tangential velocity divided by the radius of the turn (equation 4.16). Equation 4.15 is then rearranged, as shown in equation 4.17, to take advantage of equation 4.16.

$$\frac{|\vec{v}|}{R} = \frac{d\psi_T}{dt} = \psi_T$$  \hspace{1cm} (4.16)

$$\frac{|\vec{v}|}{R} = \left| \frac{g}{|\vec{v}|} \tan(\phi) \right|$$  \hspace{1cm} (4.17)
Equation 4.18 is produced by substituting equation 4.16 into equation 4.17, which can then be solved for the synthesized roll angle of the aircraft.

\[ \psi_\tau = \frac{\bar{g}}{|v|} \tan(\phi) \]  

(4.18)

Equation 4.18 represents the fundamental difference between method one and method two for synthesizing the aircraft attitude, as provided by Myron Kayton. The gravitational acceleration is assumed to be constant, or can be calculated as in section 4.1. The magnitude of the velocity vector is calculated directly from the root sum square of the velocity vector. The only step of the algorithm that contributes noise to this approximation is the differentiation of the ground track angle \(\psi_\tau\). Equation 4.18 is solved for the synthesized roll angle \(\phi\) as follows:

\[ \phi(t_n) = \tan^{-1}\left( \frac{\bar{v}(t_n)(\psi(t_n) - \psi(t_{n-1}))}{|\bar{g}| \Delta t} \right) \]  

(4.19)

The synthesized roll, pitch (flight path angle), and ground track produced by method two are shown in Figure 4.6. The same data used to produce the roll, pitch, and ground track shown in Figure 4.4 was used to calculate the roll, pitch, and ground track shown in Figure 4.6.
It is important to note that for low-dynamic flight that method two (equation 4.19) produces raw results that are less noisy than method one (Figure 4.7). During maneuvers where the pitch of the aircraft is drastically removed from the horizontal reference plane, equation 4.19 fails to correctly model the gravitational acceleration in the direction of the z-body axis. Further flight-testing is needed to fully understand the limitations of synthesizing the aircraft roll using equation 4.19.

The pitch and ground track angles shown in Figure 4.7 were produced using the same techniques for methods one and two. Looking closely at the synthesized roll angle over
time the aircraft performed two left (negative angles = roll left, positive angles = roll right) banking turns during the first five minutes of the flight. The First banking maneuver took place approximately one minute into the flight. Looking closely at the bottom trace of Figure 4.7, the difference between the roll angles (method 1 minus method 2), the difference increases during banking maneuvers. This is due to the fact that when the aircraft is in level flight, the roll calculated by method one contains measurement noise indicative of the aircraft vibrations. When the aircraft enters a banking maneuver, the algorithm noise for method one increases, while the algorithm noise for method two remains nearly constant.

![Comparison of Roll Synthesized by Methods One & Two (Low Pass Filter)](image)

**Figure 4.7 Comparison of the Synthesized Roll produced by Method One (Top) and Two (Middle), and the Difference (Bottom)**
Method one relies heavily on the assumption of coordinated flight. The majority of the error is introduced when the velocity vector is calculated and assumed to be entirely tangential to the flight path about the coordinated turn. This error is then propagated into the calculation of the normal acceleration vector and then into the formation of the normalized lift vector, which shows up in the synthesized roll angle (Figure 4.7). Method two still assumes that the velocity vector is tangential to the flight path during the coordinated turn. However, this method does not propagate the errors from this assumption into the synthesized roll-angle.

To better understand the error occurring during the banking maneuvers a truth reference is needed, which was not available at the time of this flight test. A properly mounted AHRS unit could be used as a general truth reference.

The two methods for synthesizing the approximate attitude of the aircraft are both computationally inexpensive. While the method two provides a less noisy synthesized roll, due to a single differentiation of the ground track angle, both methods lack the required robustness to track extreme flight dynamics. During large angles of attack the velocity vector deviates from the x-body axis of the aircraft inducing error into the synthesized attitude. For this reason an AHRS was added to the system.
5 Display System

The display system consists of an entire suite of sensors, computers, graphical renderings and viewable displays that are used to help the pilot navigate and control the aircraft. The display system utilizes an Attitude and Heading Reference System (AHRS) to measure the roll and pitch of the aircraft. A GPS receiver is used to estimate the position of the aircraft in ECEF and provide the ground track and velocity of the aircraft. This information is collected by a computer, which takes the measurements and forms a state-vector that is an approximation of the aircraft’s attitude and position. This state-vector is then passed to the display computer that compares the position of the aircraft to a database. A “camera” is used to view the virtual world created by the computer. The camera is then placed at the correct position and altitude within the display and the attitude information from the state-vector is used to set the orientation of the camera. The GPS ground track is used to set the heading of the aircraft. The entire database and all measurements are calculated within a predetermined ENU coordinate system to make the measurements meaningful with respect to the runway of interest.

Once the camera has been set in the database and orientated correctly the image is rendered and sent to the display. This display provides the state-vector to the pilot as an easy to interpret visual image. This visual image, representing the aircraft state-vector, is a close approximation of the real world that the pilot would normally reference during VFR conditions. Figure 5.1 shows a comparison between a real-world approach and the approximated state-vector provided to the pilot through the synthetic vision display. This
piece of compelling evidence shows how important this technology could be to general aviation pilots if implemented correctly and used responsibly.

Figure 5.1 Real World Compared to Virtual World

5.1 System Overview

The forward-looking display system is comprised of six primary components. These components are the GPS receiver, AHRS, data processing computer, terrain database, synthetic vision system, and the display unit. These six components and their interconnection are shown in Figure 5.2.
5.1.1 NovAtel GPS Receiver

The NovAtel OEM-4 3151R Power Pack was chosen because it is capable of providing independent position and velocity measurements twenty times per second (20 Hz). This GPS receiver calculates user velocity directly from Doppler measurements and not from position derivatives. In this configuration the unit sends forty position and velocity measurements to the data processing computer every second at a data rate of 115200 baud. This data rate is needed to convey the ASCII strings in real-time. The unit is capable of transmitting binary strings, but due to the complex nature of the receiver message strings, the development of the initial message parsing and processing was performed using ASCII messages. The ASCII messages and the subsequent algorithms...
used to parse and process them are in place and working efficiently. Therefore, there was
not a compelling argument to use binary communications between the receiver and data
collection computer.

The GPS receiver sends a position string and a velocity string to the data processing
computer. The position string contains the latitude, longitude and mean sea-level (MSL)
height of the phase center of the antenna. The velocity string contains the vertical speed,
the horizontal speed, and the ground track. In an ideal scenario only the position string
would be needed, but the ground track is used to augment the information from the
AHRS, which is explained below.

5.1.2 Attitude and Heading Reference System

The AHRS is capable of providing stabilized roll, pitch, and yaw measurements sixty
times per second (60 Hz). Due to the start-up procedure required to initialize the sensors
for measuring the yaw of the aircraft, it was decided to approximate the yaw using the
velocity based ground track. This eliminates unnecessary overhead in the preflight
checklist performed by the pilot prior to take-off. One problem associated with using the
ground track with the head-up implementation is that it renders the display in the
direction of travel and not in the direction of the x-body axis. Rendering the display in the
direction of the velocity vector can cause display discrepancies for the head-up
implementation during severe sideslip.
5.1.3 Data Processing Computer

The data processing computer (DPC) is a 300 MHz Pentium desktop computer running the QNX real-time operating system. The software for the data processing was written using the C programming language. The purpose of the data processing computer is to collect the GPS and AHRS measurement strings via two DB-9 pin/RS-232 links and convert them into a concise state-vector. This state-vector is then sent to the synthetic vision computer via a third DB-9 pin/RS-232 link. The collection and parsing of the GPS and AHRS data will be discussed more thoroughly in section 5.2.

5.1.4 Synthetic Vision and Terrain Database

The synthetic vision and terrain database exist on a separate computer running the Microsoft Windows® operating system. The Revolution-3D® graphics engine [9] is manipulated using Microsoft Visual Basic® and is used to render the synthetic realization of the forward-looking view from the aircraft based on a static database. The static database is produced from a gray scale bit map with white representing the highest terrain elevations and black representing the lowest terrain elevations. The gray scale bit map is rendered into a three-dimensional mesh grid that is covered with a uniform grid square system that provides the pilot with a sense of depth and motion when viewing the rendered image. Hard features, such as the runway and landmarks, are then added to provide the pilot with a sense of situational awareness when viewing the display.
5.1.5 Display Unit

The display unit was implemented as a head-up forward-looking display and a panel-mounted forward-looking display within the instrument panel. The head-up forward-looking display was comprised of an LCD projector and a section of Lexan® acting as the combiner material. The computer rendered display was sent to the LCD projector via a VGA cable and displayed on the Lexan® combiner material as a see-through head-up forward-looking display. The panel-mounted display used a flat panel high intensity display located at the top of the instrument panel. The location and installation of both of these displays will be discussed in sections 5.4 and 5.5.

5.2 Data Collection and Processing

The data collection, processing and parsing is performed using the C programming language while taking advantage of the interrupt capabilities of the QNX operating system. The data collection software is an interrupt driven program that initializes the hardware and then performs the necessary collection and manipulation of the data strings sent from the GPS receiver and AHRS unit.

The GPS receiver is set up to transmit its strings to the data collection computer at 115200 baud, no parity, 8 data bits, 1 stop bit, no handshaking, and echo off. The GPS strings are then received on communication port one of the data collection computer. The
AHRS unit is set up with a data rate of 38400 baud, no parity, 8 data bits, 1 stop bit, no handshaking, and echo off. The AHRS data is collected on communication port two of the data collection computer. The third communication port is then configured to send the state vector to the synthetic vision computer with a data rate of 115200 baud, no parity, 8 data bits, 1 stop bit, no handshaking, and echo off. Additional output ports can be configured for secondary displays.

Once the communication ports and hardware sensors have been configured the software uses two different interrupt routines to collect measurements from the GPS receiver and AHRS unit. The GPS receiver sends its measurements at 20 Hz and the AHRS sends its measurements at 60 Hz. The GPS receiver is given interrupt priority because of it has the slowest data rate. The AHRS is placed on an internal clock driven interrupt that poles the AHRS for measurements 60 times per second. The GPS receiver is placed on a communication port interrupt. If there is data at the GPS communication port, reading of this buffer is given priority. Once the GPS string is placed into a circular buffer, the program continues to poll the AHRS unit and place its measurements into a circular buffer.

Both circular buffers are monitored for start and end of a complete message based on prior knowledge of the specified message format. Each time a complete AHRS message is detected in the buffer, the buffer is read and the message is parsed for the roll and pitch measurements. The measurements received from the AHRS unit are in manufacturer standard angular format and must be converted to degrees using equation 5.1.
Each converted measurement is then placed into the state-vector string (Table 5.1). The state-vector is sent to the synthetic vision computer each time a complete AHRS message has been processed. Therefore, the state-vector information is sent approximately sixty times per second (60 Hz).

### Table 5.1 – State-Vector Fields, Parameters, and Units

<table>
<thead>
<tr>
<th>Field</th>
<th>Parameter</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>#FD</td>
<td>None</td>
</tr>
<tr>
<td>2</td>
<td>Time, GPS Seconds into the Week</td>
<td>Seconds</td>
</tr>
<tr>
<td>3</td>
<td>Local East with respect to ENU origin</td>
<td>Meters</td>
</tr>
<tr>
<td>4</td>
<td>Local North with respect to ENU origin</td>
<td>Meters</td>
</tr>
<tr>
<td>5</td>
<td>MSL Altitude</td>
<td>Meters</td>
</tr>
<tr>
<td>6</td>
<td>Ground Speed</td>
<td>Meters/second</td>
</tr>
<tr>
<td>7</td>
<td>Ground Track in degrees from North</td>
<td>Degrees</td>
</tr>
<tr>
<td>8</td>
<td>Pitch</td>
<td>Degrees</td>
</tr>
<tr>
<td>9</td>
<td>Roll</td>
<td>Degrees</td>
</tr>
<tr>
<td>10</td>
<td>*</td>
<td>None</td>
</tr>
</tbody>
</table>

The GPS data is collected and parsed upon the detection of a communication port interrupt. When a communication port interrupt is received and a complete GPS message is read from the circular buffer the string is identified as a position or velocity string and

\[
\text{Angle} = \frac{(\text{measurement})(180^\circ)}{2^{15}}
\] (5.1)
parsed accordingly. The latitude and longitude are converted to a local East and North measurement with respect to a surveyed point located at the threshold of Ohio University runway 25. The East, North, and MSL position measurements, along with the magnitude of the velocity vector and ground track are stored in global variables.

Each time the state-vector is sent it is loaded with the position and attitude measurements stored in a set of global variables. The AHRS measurements are considered new each time a state-vector is sent while the GPS measurements will be up to 50 milliseconds old. Once a new GPS position or velocity string is received the global variables are updated and the state-vector loaded accordingly.

5.3 Synthetic Vision & Graphics Engine

Synthetic vision is not a new concept to the field of aviation. The goal of this research is not to “reinvent the wheel” and create a unique synthetic vision display for general aviation but to incorporate an existing system into the display architecture once the system architecture is established. For the duration of this research the synthetic vision was implemented using the Revolution-3D® graphics engine and a simplified version of the surrounding airport terrain as a static database [9].

The synthetic vision software is an object-oriented program that processes the state-vector information and renders the appropriate display based on the static terrain database. The program is responsible for setting up the communication port for receiving
the state-vector message from the data processing computer. After the communication port is set up, the graphics engine, pipeline, sky features, terrain features, and runway are created. Upon start up, a set of global variables is initialized including the necessary parameters for the attitude and position of the aircraft.

The state-vector is received by the synthetic vision computer via a DB-9/RS-232 link. The Visual Basic program monitors the communication port for an interrupt event. When an event is received the state-vector message is checked to ensure that all of the necessary data fields are present. If the message is intact, it is sent to the parsing algorithm, if not then it is ignored. Once the state-vector is parsed, a set of global variables is loaded with the state-vector information and the main routine of the program is notified that a valid state-vector has been received and processed.

The main loop of the program contains a Boolean statement whose condition is set by the communication port interrupt. If the Boolean variable is true, then the main routine updates the global variables containing the attitude and position data, if it is false it assumes that the current state-vector information is valid. The position, roll, pitch, and heading of the aircraft are used to place the camera in the display and the appropriate image is rendered and sent to the VGA port for output to the display device.
5.3.1 Synthetic Vision Local Reference Frame

There are three primary reference frames used by the data processing computer and the synthetic vision software: Earth Centered Earth Fixed (ECEF), ENU with respect to the airport, and the ENU within the graphics engine. The ECEF reference frame is used for the original measurements made by the GPS receiver. The data processing computer then converts the ECEF position into an ENU measurement with respect to the threshold of runway 25 at the Ohio University Airport in Albany. This ENU measurement is then converted to the local ENU of the graphics engine that is a 1024 pixel by 1024 pixel grid with each pixel set to 10 meters so that the pixel to meter ratio is 0.1 [pixels/meter]. The local ENU system used by the graphics engine uses the x-axis as the local East direction, z-axis as the local North direction, and the y-axis as the local up direction (Figure 5.3) [10, 11, 12].

![Figure 5.3 ENU Reference Frame used by the Graphics Engine](image-url)
The state-vector ENU position of the aircraft is then shifted by 512 pixels in the East and North direction to convert to the graphics engine local ENU system [10, 11, 12].

\[
East_{\text{Local}} = East_{\text{ENU}} + 512[\text{pixels}]
\]

\[
North_{\text{Local}} = North_{\text{ENU}} + 512[\text{pixels}]
\]

### 5.3.2 Synthetic Vision Object Placement

The ENU coordinate system representing the ECEF GPS measurements are given with respect to the threshold of the runway and is the anchor point for each object rendered within the display. Each object is placed within the display by its center point and then created about that X and Z coordinate. To place the runway as an object within the rendered display the length and width of the runway were converted to a pixel unit [10, 11, 12].

\[
\text{Length} = 560[\text{feet}] \left( \frac{0.3048[\text{m}]}{[\text{ft}]} \right) \left( \frac{0.1[\text{pixels}]}{[\text{m}]} \right) = 170.718[\text{pixels}]
\]

\[
\text{Width} = 100[\text{feet}] \left( \frac{0.3048[\text{m}]}{[\text{ft}]} \right) \left( \frac{0.1[\text{pixels}]}{[\text{m}]} \right) = 3.048[\text{pixels}]
\]
The touchdown point of runway 25 is placed at the center of the 1024 by 1024 pixel area to provide the maximum usable area for pattern work and approach. The GPS heading of the runway is 240 degrees from North. This heading was used in the placement of the runway within the rendered display. Figure 5.4 shows the relative placement of the runway within the synthetic vision local reference frame. The geometry established in Figure 5.4 was used to calculate the center of the runway in pixels with respect to the center of the 1024 by 1024 pixel grid system. The center of the rectangle representing the runway is located at (475.052, 490.668) [pixels] [10, 11, 12].

Figure 5.4 Runway Placement within Synthetic Vision Local Reference Frame

A Precision Approach Path Indicator (PAPI) was also installed to the left of runway 25. The PAPI is used to give the pilot a visual cue for the approximate angle of the
approach path. Figure 5.5 provides an explanation of the visual information the PAPI system provides to the pilot.

The PAPI system uses the local level coordinate system to calculate the glide path angle. If the GPS receiver measurements were perfect, this portion of the display would be very useful to the pilot. The vertical GPS measurements contain the largest portion of the position error. Therefore, this portion of the display is still under development and is only intended to be used with a Wide Area Augmentation System (WAAS) enabled GPS receiver.
5.3.3 Image Rendering

The final rendered image is the result of layering a three-dimensional mesh with a color plate to create the local level system. After the mesh has been layered with the color plate, it is wrapped with a panoramic photograph representing the sky features the pilot will view when using the display. During ideal flying conditions one might think the sky to be clear blue and therefore, a clear blue panoramic photograph would make the most sense in this type of display. After testing several types of panoramic photographs it was determined that the presence of clouds provides a better sense of roll and pitch [10, 11, 12].

![Figure 5.6 Portion of Gray Scale Bit Map Terrain Mesh and Color Plate](image)

The left portion of Figure 5.6 represents a portion of the gray scale bit map used to represent the local terrain surrounding the airport. The right portion of Figure 5.6 is the color plate used to cover the three-dimensional mesh. Figure 5.7 shows the panoramic
photograph used to create the sky within the virtual world. The lower portion of the panoramic photograph was set to a solid earth tone color to provide a solid horizon line when viewing the display from high altitudes [10, 11, 12].

Figure 5.7 Panoramic Sky Photograph used within the Rendered Image

Figure 5.8 shows an approach on virtual runway 25 created by the graphics engine using the information provided by the bitmaps in Figures 5.6 and 5.7.

Figure 5.8 Image Rendered by the Graphics Engine
5.4 Head-up Forward Looking Display

The forward-looking head-up display was designed specifically for the low-end general aviation market. This technology was developed, with the low-time instrument rated pilot in mind, to enhance situational awareness in IMC.

The head-up forward-looking display was initially envisioned to support only the landing phase of flight. During its development it has become apparent that the display is applicable to all three phases of flight, departure, en-route flight, and landing.

The landing of an aircraft in IMC is the most critical time when a technology such as this is needed. This display technology is not a replacement for the current Instrument Landing System (ILS). This device is presented as an aid to help the pilot safely navigate the aircraft to a decision height of 200 feet AGL where the pilot will either be able to land the aircraft visually or perform a missed approach due to the fact that the runway could not be identified. In the latter of the two cases, the pilot would not even have attempted to land the aircraft, and would have diverted to another airport [10].

A good illustration of a real-world application would be an Instrument Flight Rules (IFR) scenario where visibility might be two to three statute miles, but the ceiling is only 800 feet. In this simplified scenario the ceiling is composed of cumulus clouds producing moderate turbulence effectively increasing the single pilot workload. The pilot would perform a standard instrument approach crossing the seven-mile beacon perpendicular to the runway, banking right and eventually coming about so the aircraft is lined up on the
runway. The aircraft should be at a height of approximately 2200 feet AGL and seven nautical miles from the runway threshold. At this time the aircraft is still well within the clouds and the pilot is flying “blind” [10].

Today, the only choice the pilot has is the ILS and watching the “T” on the instrument panel of the aircraft. With the head-up forward-looking display installed in the aircraft, the pilot would flip the combiner screen down and hit a button that would initialize the display to the correct position, altitude, and attitude. Within seconds the pilot could be looking at the runway with the synthetic vision provided by this display technology. The pilot would then use the synthetic vision as an aid to navigate the aircraft on a glide slope of 3° to within about a nautical mile from the runway threshold. The aircraft would be at a height 300 feet AGL, which should be below the clouds, and the pilot would simply flip the display out of his or her field of view and land visually. The beauty and practicality of the head-up forward-looking display is that it places the synthetic vision in the pilot’s direct field of view that overlaps the visual world with the real world. Because of this overlap, the pilot would know the exact moment when he or she breaks through the clouds and could fly visually from that point on. This appears to be a definite advantage over a head-down panel-mounted display, which is troubled by some of the same distractions as flying a standard instrument approach using the “T” on the instrument panel [10].

The head-up forward-looking display is an excellent tool for en-route flight during IMC due to the fact that it provides the pilot with an “out the window view”, increasing
his or her situational and spatial awareness. This type of technology could have averted the accident that tragically took the life of Mr. Kennedy and his passengers [10].

The third scenario in which a head-up forward-looking display could provide the pilot with assistance is during instrument climb-out. During VMC the pilot would monitor the view from the front windshield and occasionally check the aircraft’s position and attitude out the side window. During IMC, the pilot no longer has the luxury of visual flight and once again must rely on the instrument panel. For an aircraft equipped with a synthetic vision display, instrument climb-out would be much safer due to the fact the pilot would have the visual cues afforded him or her under clear conditions [10].

If the aforementioned scenario is examined during a climb-out, with a ceiling of 800 feet AGL, and visibility between three and four statute miles, the preflight check list would be modified only slightly. Before take off the pilot would initialize the display and then run through the normal preflight checklist. The pilot could then fly visually until he or she reaches the cloud ceiling. Once the pilot climbs into the clouds the pilot would lower the forward-looking display panel manually providing the visual cues need to maintain attitude [10].

During visual climb-out, a pilot will look out the left window for visual cues regarding the attitude and rate of climb of the aircraft. During instrument climb-out the pilot is deprived of all visual cues. The instrument climb-out scenario has brought to light the idea of adding multi-view displays to the cockpit in order to increase the amount of visual information provided to the pilot. Multi-view displays were not flight tested during this research but the software was modified to simulate left-looking and right-looking
displays based on pre-recorded flight data. The results of the simulation were very promising and should be investigated further with actual flight tests [12].

5.5 Panel Mounted Forward Looking Display

Panel mounted forward-looking displays have been under investigation for a number of years now including general aviation applications. In order to compare the performance of the head-up forward-looking display it was necessary to implement the head-down display and study possible mounting configurations within the instrument panel.

A head-down forward-looking display offers many of the same benefits as the head-up display. The main problem with a head-down panel mounted display is its placement. With the typical glass cockpit displays, the synthetic vision display is often mounted in the instrument panel causing it to suffer from some of the same limitations as mechanical instrumentation currently in use.

The two main limitations that panel-mounted synthetic vision displays tend to suffer from are realism and location. With the graphics in use today, it is possible to create a synthetic or virtual view of the outside world that is near photographic quality. This type of display is very compelling and the novice user has a tendency to forget to look beyond the instrument panel. This leads directly into the second problem, which is the location of the display. If the display were moved to a strategic location within the peripheral range
of the pilot’s field of view, he or she would more effectively be able to monitor the display and the real worldview at the same time.

A photograph of the installed panel-mounted display is provided in chapter six showing the placement of the display just above the standard instrumentation within the range of the pilot’s peripheral field of view. This display was installed in such a way that it would fold down to minimize the amount of window real estate it occupied when not in use.
6 Installation

The synthetic vision display discussed in chapter five went through several phases of development. Each stage of development was flight-tested using a Piper Saratoga. The installation and power requirements for each subsequent flight test changed as the display evolved and new pieces of equipment were added into the display architecture. This chapter will focus on the current display system and its installation into the flight test aircraft.

6.1 Placement and Mounting of System Components

The GPS receiver, AHRS, data processing computer, and power supply were mounted into a specially designed rack developed for this project. A flat panel LCD was attached to the rack in order for the flight test engineer to operate and monitor the system (Figure 6.1). The synthetic vision software was loaded onto a laptop computer that was held by the flight test engineer during flight operations and stowed for take-off and landings.
The only two pieces of equipment whose location was critical to the system measurements were the AHRS and the GPS receiver antenna. The AHRS was mounted very near the aircraft’s CG in line with the local body axis. The GPS receiver antenna was mounted just above the aircraft’s CG on the top of the aircraft fuselage.

The display was implemented using a Liquid Crystal Display (LCD) projector and a 14” by 10” section of ¼” gray Lexan 9034 acting as the combiner material. The LCD projector chosen to perform the task of the projection system was the InFocus® Digital Multimedia Projector LP280. The projector was chosen for its size, keystone correction capability, range of inputs, resolution, and its front or rear projection capability.
The projector is mounted on a rack placed in the center of the aircraft between the pilot and safety pilot (Figure 6.2). The safety pilot occupies the left seat and the display is mounted in front of the right seat. The projector is set back about 3 inches from the front side of the pilot seat headrest at a slight angle to project the synthetic image of the terrain onto the combiner glass in the pilot’s field of view. The keystone correction is used to remove any nonsymmetrical properties of the projected image.

The combiner glass is placed directly in the field of view nearly perpendicular to the pilot’s line of sight (Figure 6.3). The display provides the pilot with a 70° field of view, ±35° of the vertical center. The display focal distance is set to project the image at the exact distance of the combiner glass resulting in a non-collimated display.

The glass is mounted to a hinged brace that positions the display against the dash to allow for maximum pilot movement and minimal effect on egress issues (Figure 6.3). The combiner material was placed in front of the right seat with the copilot acting as the observer for the flight tests.
Figure 6.2 LCD Projector Mounting

Figure 6.3 Combiner Glass Placed Perpendicular to the Subject Pilot's Field of View
The flat panel display used with the head-down panel-mounted configuration was placed within range of the pilot's peripheral field of view as shown in Figure 6.4. The display was mounted on a hinge that allowed for the display to be stowed upon the instrument dash when not in use.

![Panel-mounted Forward-looking Display Strategically Mounted within Range of the Pilot's Peripheral Field of View](image)

**Figure 6.4 Panel-mounted Forward-looking Display Strategically Mounted within Range of the Pilot's Peripheral Field of View**

### 6.2 Power Requirements

The power requirement of the synthetic vision system is one of the major hurdles that must be overcome before this technology can become common in general aviation aircraft. The GPS receiver and AHRS unit are of little concern, drawing less than one ampere each. The data processing computer and the synthetic vision computer can
eventually be combined into one computer with a common DC power source supplied by the aircraft. The flat panel display has a power requirement similar to that of the GPS receiver drawing less than one ampere. For flight-testing using the head-up forward-looking display an uninterruptible power supply (UPC) was used to provide power to the LCD projector.

6.3 Egress Issues and Pilot Concerns

When a system is placed in an aircraft for flight-testing it must not interfere with the normal operation. The system or device must be installed so that it does not interfere with the crew’s ability to exit the aircraft in case of emergency or break loose during aircraft dynamics. The primary concern with the display system was the placement of the actual display devices. The placement of the head-up and panel-mounted displays had the potential to interfere with the pilot’s view through the forward window of the aircraft.
7 Flight Tests

Numerous flight tests were performed over the course of this research. The initial flight tests were performed for data collection and to evaluate the performance of the aircraft position and attitude sensors and the data collection computer. After the initial system evaluation the remainder of the flight testing focused on the evaluation of the head-up forward looking display and the head-down panel-mounted display. Each display configuration was installed and evaluated separately.

7.1 Flight Test Aircraft

The Piper Saratoga is a single-engine, low-wing aircraft capable of carrying six occupants (Figure 7.1). The Saratoga is configured with a 3-blade variable pitch/constant speed propeller that is driven by a Lycoming IO-540-K1G5 (300 Hp) engine. The aircraft has an empty weight of 2216 lbs with a useful load weight of 1384 lbs and a cruising speed of 135 knots at 65 % power [10, 11, 12, 13].
The Saratoga has a great deal of cargo and passenger space, but has little room in the cockpit for additional equipment. This was another motivating factor for investigating a head-up configuration as well as a panel-mounted head-down display. The windshield area on either the left or right side of the cockpit provides an area of approximately 18 inches by 18 inches in which a piece of combiner material can be mounted in the pilots field of view. This is roughly the size needed to correctly render the displayed image to the pilot.

For the flight tests required for the head-up configuration and the panel-mounted configuration the Saratoga provides all of the basic needs for size and power requirements. It is capable of safely carrying a pilot, safety pilot and flight engineer as well as the equipment and sensors needed to drive the display.
7.2 Flight Profiles

General aviation flight profiles are typically comprised of level flight and gentle banking maneuvers. Even the most extreme maneuvers are typically considered low dynamic in comparison to the capabilities of modern sensor equipment used to measure the aircraft attitude and position. To evaluate the system’s performance, standard traffic patterns were flown at the Ohio University Airport in Albany on runway 25. Using this type of flight profile allowed the system to be evaluated for latency and accuracy by comparing the synthetic horizon to the observed horizon. Another advantage of this type of profile was that the runway was in view more frequently allowing for a direct comparison of object placement and size within the virtual world created by the synthetic vision software.

7.3 Head-up Display Evaluation

A flight test was conducted at 21:30 hours on the 27th of May 2003 at UNI to evaluate the performance of the head-up forward-looking display. The weather was clear (VFR) with a storm approaching from the West. Several approaches were made on UNI runway 25. Video and photographs were taken to document the results of the flight test.

It was decided that the subject pilot would sit in the right seat to act as an observer and critique the performance of the display system. The LCD projector was placed
between the pilot and copilot with the image displayed on the combiner glass in the subject pilot’s field of view (Figures 6.2 and 6.3).

Figure 7.2 shows the display system during climb out as the aircraft was banking left into the down wind leg of the traffic pattern. This photograph was taken from the perspective just behind the left seat due to the fact that the video recorder was placed behind the right seat.

Figure 7.3 shows the display system during the initial approach on runway 25. Unfortunately it is hard to make out the runway from this photograph. The aircraft roll indicated by the display system matches that of the horizon. The pitch of the aircraft looks to be too low, but this is due to the perspective of the camera taking the photograph. During the flight test, the only real problem was the object size presented by the display was slightly smaller than objects in the real world.
Figure 7.2 Display System During Climb Out

Figure 7.3 Initial Approach on Runway 25
Figure 7.4 shows the display system during the final approach on runway 25 at the Ohio University Airport. The camera perspective does not capture the overlapping of the synthetic vision with the real world. From this perspective it is still apparent that the virtual runway is at the correct placement and orientation with the real runway. The virtual horizon matches that of the real horizon indicating that the pitch and roll of the aircraft were being provided to the pilot accurately and in real-time.

During the evaluation of the head-up forward looking display two flaws were noticed in the system performance. The image rendered suffered from distortion (i.e. synthesized objects rendered on the head-up display did not match size and/or position of real world
objects), but improved as the aircraft approached the threshold of the runway. The second performance issue was that rapid changes in vertical positioning caused the synthesized horizon line to vary from the true horizon. During this flight test a non-augmented GPS receiver was used. It is believed that the use of a WAAS enabled receiver would help mitigate most of the vertical error propagated into the rendered display.

7.4 Panel-mounted Display Evaluation

A flight test was conducted at 19:45 hours on the 17th of September 2003 at UNI to evaluate the performance of the head-down panel-mounted forward-looking display. The weather was clear (VFR). Several approaches were made on runway seven. Video and photographs were taken to document the performance of the display.

During this flight test the panel-mounted display was placed near the top of the instrument panel as shown in Figure 6.4. The pilot sitting in the left seat was the pilot in command and also the primary observer of the display. A safety pilot was sitting in the right seat and also made observations and comments about the display.

The pilot in command was instructed to fly pattern work on runway 7 while casually observing the display to become comfortable with visual cues being provided. Once the pilot became comfortable with the display he was instructed to use the display for primary guidance and use the real-world cues as a check against the aircraft’s position
and attitude while flying the traffic pattern. During this operation the safety pilot monitored the aircraft’s progress and watched for traffic.

The synthetic vision image rendered to the pilot is generated from the same display system that provided the image to the head-up display. During the flight test, the attitude and position of the display was not as much of a concern as the usability of the image provided to the pilot. Figure 7.5 shows the display during final approach on runway seven at Ohio University airport. From this photograph it can be seen that the display is providing real-time attitude and position information to the pilot. It is important to note the view of the video recording camera is not the same as the pilot’s view.

Figure 7.5 Panel-mounted Display during Final Approach on Runway 7
The only major concern with the display is that the pilot had a difficult time maintaining altitude using the display for primary guidance. Many synthetic vision systems already in existence today use pavers or a tunnel in the sky to help the pilot navigate the aircraft to the desired location. This display does not use pavers or a tunnel system for several reasons. The primary reason this display uses only simulated visual cues for guidance has to do with the accuracy of the sensors used to estimate the position of the aircraft. In order to keep the system affordable the only positioning device is a GPS receiver. This system was not developed for precision landings or precision flying.

7.5 Pilot Evaluation

The head-up and panel-mounted display systems were both evaluated by commercially rated pilots. Due to safety issues surrounding the head-up display the most valuable comments and suggestions came from the flight-testing of the panel-mounted display.

The display provided position and attitude information in real-time. There were not any latency issues or major measurement error issues concerning the use of the panel-mounted display. The size of the panel-mounted display was adequate, though the intensity of the display was of some concern. It is important for the pilot to be able to adjust the contrast and the brightness of the display while in use.
The test pilot’s had two main issues with the panel-mounted display. The first issue was that it was hard to maintain altitude using the display for primary guidance. To help the pilot have a better feel for the pitch of the aircraft in order to maintain altitude it was suggested that a portion of the nose be placed in the display. This would give a visual reference of the aircraft pitch with respect to the local horizon. Horizon indicators could also be placed at the left and right sides of the display along with altitude bars providing the pilot with not only a visual cue but also a numeric read-out of altitude. This display was designed with a minimalist approach; the horizon indicator and altitude bars can easily be added to the display to fix this problem. The second was that during traffic patterns the runway was not always visible in the display especially during the crosswind and base legs of the traffic pattern. The display was intended to provide the pilot with situational awareness and reduce the workload during low-visibility situations. In order for the system to meet this requirement a bird’s-eye or God’s-eye view will need to be added. This will increase the amount of real-estate necessary to house the display but is absolutely essential if the display is to quickly provide the pilot with situational and spatial awareness.
8 Head-up vs. Head-down

The goal of this research was to implement and evaluate a head-up forward-looking display and a panel-mounted forward-looking display for use with general aviation aircraft. The performance benefits of both displays have already been discussed. In this section the head-up and head-down displays will be compared based on pilot comments, installation, and possible certification issues.

Both displays are driven by the same sensor suite and are limited by the same measurement errors and performance criteria. In order for a system to provide spatial and situational awareness to a pilot while reducing his or her work load it must, at a minimum, use a forward-looking display and a bird’s-eye display. Based on pilot evaluation, these two displays are critical. With this in mind, the head-down panel-mounted display lends itself to general aviation applications more readily than the head-up display. With the head-up forward-looking display a large portion of the window space is required to mount the combiner material (Figure 6.3). It also requires a projection device to materialize the display on the combiner material (Figure 6.2). The projection device also requires considerable space and power. Along with these two pieces of equipment, a secondary display is needed to provide a bird’s-eye view for situational awareness of the aircraft’s location relative to the runway.

The head-down panel-mounted display requires only the real estate necessary to mount the flat panel display (approximately 7 inches by 9 inches), which is considerably
smaller than the combiner material. A second equivalent flat panel display could then be used to implement the bird’s-eye display in the vicinity of the forward-looking display.

Another issue with using these displays is the reduction of pilot workload. When the displays were used at dusk, the light emitted by the displays either through reflection off of the combiner material or directly from the flat-panel display, was minimal and not a distraction to the pilot. When dark fell, the light emitted from both displays was too intense and caused eyestrain. The pilot’s also had a difficult time transitioning from the bright display and then back to the outside world. These illumination issues were mostly noticed during the flight-testing of the flat-panel display due to the fact that the pilot in command was attempting to use the display for primary guidance. The location of the flat-panel display allows for it to be accessed easily by the pilot. The addition of a light sensitive dimming sensor would allow the intensity of the light being emitted from the flat-panel display to be automatically adjusted. The head-up display would require a much more complicated method of attenuating the amount of light provided by the projector. This could be done, but very few commercial off-the-shelf (COTS) projectors come with this feature.

The head-up display and the panel-mounted display implemented in this research should not represent head-up and head-down displays as a whole. This research does suggest that head-down panel-mounted forward-looking displays lend themselves more readily to general aviation applications.
9 Certification

The Federal Aviation Administration (FAA) is the governing body that sets the standards of safety and is the final authority on the certification of aviation aircraft and systems. In an ideal world engineers would build systems that work in every scenario. Certain aspects of aviation research such as human factors studies, safety, and the subsequent certification of proposed systems would never be an issue. This display system was designed to reduce single pilot workload and to increase safety in IMC conditions, but the reality is that any technology that is used incorrectly or not tested thoroughly can potentially have tragic results.

The display system proposed in this research was implemented as a head-up and panel-mounted display for several reasons. One of the primary reasons was to see which display implementation helped to reduce the single-pilot workload. In theory both implementations should reduce the pilot workload for the reasons discussed in chapter five. However, theory does not always match the physical world. For this reason numerous flight tests were performed to compare and contrast the performance of the displays with safety concerns expressed by the FAA.

The primary concerns expressed by the FAA revolved around pilot workload and the potential of obscuring the pilot’s view of the real world. Does a head-up display reduce single pilot workload by providing an out-the-window view during low visibility conditions or does it cause unnecessary eyestrain and obscure the pilot’s view of possible
real-world threats not shown within the synthesized display? Another concern involving the head-up configured display had to do with the amount of excess light given off by the display’s LCD projector. Other concerns dealt with collimation issues and different types of distortions that deteriorate the quality of the information provided by the synthetic vision software.

The head-down panel-mounted display automatically reduces the list of concerns the FAA has with this type of technology. For starters, collimation and image distortion are no longer an issue. A simple dimming sensor could be used to reduce the amount of light in the cockpit. The windows are free of equipment clutter and the pilot has full view of the real-world out-the-window view while still being able to use the synthetic vision display.

Other issues that concern both systems have to do with the certification of the GPS receiver, AHRS unit, the synthetic vision software, and terrain database. This display system relies heavily on commercial of-the-shelf (COTS) technology and it is not the intent of this research to certify each component. The installation and use of the display are the primary focus of the certification issues outlined above.
10 Conclusions and Recommendations

Synthetic vision systems, providing either forward-looking or multi-view displays, are a necessary step in the evolution of general aviation aircraft instrumentation. Many pilots feel that this is not true and that current instrumentation is sufficient. On average three general aviation aircraft are lost each month, many of these due to spatial disorientation. This fact alone suggests that this type of research must be continued with the ultimate goal of providing this technology to the general aviation community.

General aviation aircraft may be years away from head-up displays that fill each window of the cockpit providing the pilot with an exact virtual image of the outside world, complete with the most recent topographic features and real-time traffic awareness. This does not mean that intermediate steps cannot be taken, such as panel-mounted displays placed at strategic locations within the cockpit.

This research has shown that this type of technology can be made available to the GA pilot at an affordable cost. The commercial off-the-shelf (COTS) and miscellaneous equipment used in this research was purchased for under $15,000 (U.S.). Certification and the man-hours were not included in the development cost for this proof of concept prototype. There is much work to be done regarding human factors studies and the continued development of GA specific synthetic vision software.
This research also explored alternative methods of providing attitude information to the synthetic vision system via velocity vector based attitude approximation. Two methods of approximating the aircraft roll were explored and tested against real flight data from a general aviation aircraft that was assumed to be in coordinated flight. This research highlighted the advantages of both methods and ultimately showed that for the assumption of coordinated flight and small angles of attack, method two (described in chapter 4) produced cleaner and more usable results without the aid of a Kalman Filter. It is recommended that both of these methods be explored further and compared to a truth reference, such as an inertial unit, for the purpose of evaluating both algorithms beyond the flight dynamics of coordinated flight. The main reason that an AHRS unit replaced the velocity vector based attitude was that the algorithm is only valid under the assumption of coordinated flight. This algorithm provides a very cost-effective means of providing the aircraft attitude to a synthetic vision system or other horizon indicator. It would be worthwhile to study the constraints of this algorithm more thoroughly and its possible use as a back up or augmentation to an AHRS unit.

The goal of this research was to develop an equipment sensor suite specifically designed for general aviation aircraft, and to evaluate two possible display configurations. Both a head-up forward-looking display and a panel-mounted head-down forward-looking display were implemented. While in theory, the head-up display was expected to reduce pilot workload, it cluttered the forward-looking window of the aircraft and was very difficult to implement. The head-up installation required more room and
power than most GA aircraft have to offer. The synthetic vision software used to render the virtual image for the head-up display was much more complicated due to image distortion (i.e. proper alignment of the real world and the rendered virtual world). The head-down display required a small amount of room on the instrument panel and could be used at the pilot’s discretion. The flat-panel display was installed left of center, directly in front of the pilot. Optimally this display should be installed in the center of the instrument panel with approximately one third of the display rising above the top of the console. This recommended installation is very similar to the flight-tested installation shown in Figure 6.4 with the exception that the flat panel display should be centered on the console.

Flight-testing of the flat-panel head-down display involved some pilot evaluation. From an engineering perspective the display worked flawlessly providing real-time attitude and position information to the pilot with respect to the runway threshold at the Ohio University Airport. From a human factors perspective, the display was too bright during low-light conditions and a bird’s-eye-view was also needed for a complete understanding of the aircraft’s position relative to the runway. Several other suggested improvements were noted by the test pilot and safety pilot and have been taken into consideration for future flight-testing of this display system.

It is recommended that this research be continued as a head-down panel-mounted multi-function display. This would allow the pilot to have multiple views for increased situational awareness, and to toggle to a bird’s-eye-view when needed. It is envisioned
that this system should be expanded to three flat-panel displays providing left-looking, forward-looking, and right-looking guidance information. The right-looking display would then alternate as the bird’s-eye-view at the pilot’s discretion.

It is obvious that this system is still in its infancy and needs further flight-testing and development. This research has demonstrated the viability of synthetic vision for general aviation aircraft and has also shown the need to design and tailor such systems specifically for the general aviation fleet. The concept of this research is not new, and the intent of the research was not to reinvent synthetic vision, but to place it into the hands of the pilots who need it the most, general aviation.
References


Appendix A  Data Collection Code

The code presented in appendix A was written using the C-programming language. The code provided in appendix A.1, A.2, and A.3, is used to collect and parse the GPS and AHRS data. These four sections of code reside on the data collection computer. The main body of the code (Figure A.1) is the most complex due to the interrupt service routine (ISR) used to handle the GPS strings and AHRS packets. The rest of the data collection software is very modular and should be easy for an experienced programmer to follow. The algorithms for the synthesized aircraft attitude using method one is provided in appendix A.4. The global variables used are provided in appendix A.5.

Figure A. 1 Pseudo-code for the Main Routine used for Data Collection
# A.1 Main and Interrupt Service Routines

```c
/
* Author:  Douglas Burch
  * Sai Kalyanaraman
* Date:    13 Dec 2002
* Purpose:
  * This program is set up to monitor the AHRS communication port based on a clock
  * interrupt ISR. The GPS communication port is monitored based on a communication
  * port driven ISR. The AHRS data is collected and placed into a serial buffer, when a
  * comm. Interrupt is received from the GPS port, it is given priority, once the entire GPS
  * string has been collected and placed in the circular buffer, the AHRS collection resumes.
  * Two circular buffers are used, one for the GPS data and one for the AHRS data. The circular
  * buffers are then searched for the appropriate headers and parsed. The critical data from the
  * strings are then placed into global variables and updated each time a new GPS string or AHRS
  * packet arrives. The global variables are then used to perform the necessary positioning and
  * attituded calculations in the main loop.
  * Once the positioning and attitude calculations have been performed the data is placed into a state
  * vector and sent to communication ports three and four for output to the synthetic vision computer.
  * The state-vector is updated each time new attitude information is sent from the AHRS.
  * The Latitude, Longitude, and Altitude are converted to a local level coordinate system using the
  * threshold of runway 25 as the origin.
  * Runway 25 Threshold: 39.213774 N (degrees) 0 East (meters)
  * -82.224948 W (degrees) 0 North (meters)
  * Runway 7 Threshold: 39.208133 N (degrees) -1116.703657 E (meters)
  * -82.237877 W (degrees) -626.181143 N (meters)
  * This information is used as the Aircraft State Vector which consists of the following fields:
    1. Time, GPS Seconds into Week
    2. Local East in meters
    3. Local North in meters
    4. MSL Altitude in meters
    5. Ground Speed in m/s
    6. Ground Track in degrees from North
    7. Pitch Angle in degrees
    8. Roll Angle in degrees
  * This Aircraft State Vector is then packetized into an ASCII string with the following fields
    where the numbers 1-8 represent the members of the Aircraft State Vector listed above:
    #FD,1,2,3,4,5,6,7,8,*
  * Interrupt Request Lines:
  * Name    Address  Interrupt
  * COM1     3F8      IRQ4
  * COM2     2F8      IRQ3
```
/* COM3  3E8   VARIES
  * COM4  2E8   VARIES
  */

                                  ***********************************************/

/* INCLUDED LIBRARIES
                        ***********************************************/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include <string.h>
#include <unistd.h>
#include <termios.h>
#include <fcntl.h>
#include <sys/neutrino.h>
#include <sys/syspage.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>
#include <time.h>
#include <errno.h>
#include <unix.h>
#include <pthread.h>
#include <atomic.h>

// Libraries required for developed routines.

#include "WGSconversion.h"
#include "PseudoAttitude.h"
#include "SVSGlobalvars.h"
#include "AHRS.h"

                                  ***********************************************/

/* IRQ INTERRUPT DEFINITIONS
                        ***********************************************/

/* DEFINE THE INTERRUPT FOR COMM1, COMM2, COMM3, and COMM4 */
#define IRQ3 3   // COMM1
#define IRQ4 4   // COMM2
#define IRQ9 9   // COMM3 and COMM4 share interrupt line 9.

                                  ***********************************************/

/* BUFFER DEFINITIONS
                        ***********************************************/

/* Define the size of the holding buffers, the processing buffers and the resultant output buffers */
#define PROCBUFSIZE   4096
#define INPUTBUFSIZE1 4096
#define INPUTBUFSIZE2 4096
#define OUTPUTBUFSIZE  512
```c
#define PACKET_SIZE_VG_MODE 30
#define NUMDATABYTES_IN_VGMODE 28

/***** INTERRUPT HANDLING FUNCTIONS *******/

#include <unistd.h>

volatile int COMM1_IntrFlag;

// DEFINE PSEUDO BOOLEAN VALUES FOR TRUE AND FALSE
#define TRUE 1
#define FALSE 0

const struct sigevent *clock_handler(void *

FUNCTION PROTOTYPEING
int ProcessAHRSData(void);
int ObtainGPSData(void);
int Chksum(char Data[], int size);
int Ptr_Diff(int scanner_ptr, int current_ptr, int bufsize);
unsigned int SearchGPSHeader(unsigned char[]);
void GetBytes(unsigned char *strScannerptr, unsigned char *strStartptr, unsigned char *strEndptr, int bufsize, int numbytes);

/* INTERRUPT HANDLER FUNCTION 'COMM1_HANDLER' */
#pragma off(check_stack):
struct sigevent event;
struct sigevent clock-event;
volatile unsigned counter;

/* HARDWARE AND DEVICE POINTERS */
// initialize the hardware, etc.
int intCOMM1 = 0; /* File Pointer for /dev/ser1 */
int intCOMM2 = 0; /* File Pointer for /dev/ser2 */
int intCOMM3 = 0; /* File Pointer for /dev/ser3 */
int intCOMM4 = 0; /* File Pointer for /dev/ser4 */
int intCOMM1_ID = 0; /* Return Value for COMM1 Interrupt Handler. */
int intCOMM2_ID = 0; /* Return Value for COMM2 Interrupt Handler. */

/* PROCESSING BUFFERS */
/* This is used as a generic counter inside the code to initialize the AHRS */
unsigned char strProcBuffer1[PROCBUFSIZE]; /* Processing Buffer for incoming data */
/* Pointer to the first element of beginning of the buffer "strProcBuffer1" */
```
unsigned char * strProcbufstartptr1;
/* Pointer to the last element of beginning of the buffer "strProcBuffer1" */
unsigned char * strProcbufendptr1;

/* Pointer to the next element that needs to be scanned in the buffer "strProcBuffer1" */
unsigned char * strProcscanptr1;
unsigned char * strInputbufptr1;

/* Pointer to the current element that needs to be loaded with new data */
unsigned char * strProccurrentptr1;

/* Array is a temporary holding area for the bytes from the SIO */
unsigned char strComparisonbuf[OUTPUTBUFSIZE];
unsigned char * strComparisonbufptr; /* Pointer to Array strComparisonbuf[] */
unsigned char strGPSdata[OUTPUTBUFSIZE];
unsigned char strAHRSdata[OUTPUTBUFSIZE];

unsigned char strProcBuffer2[PROCBUFSIZE]; /* Processing Buffer for incoming data */

/* Pointer to the first element of beginning of the buffer "strProcBuffer2" */
unsigned char * strProcbufstartptr2;

/* Pointer to the last element of beginning of the buffer "strProcBuffer2" */
unsigned char * strProcbufendptr2;

/* Pointer to the next element that needs to be scanned in the buffer "strProcBuffer2" */
unsigned char * strProcscanptr2;

/* Pointer to the current element that needs to be loaded with new data */
unsigned char * strProccurrentptr2;
unsigned char * strInputbufptr2;

volatile int intRolloverflag1 = 0; /* Flag indicates Processing buffer Rollover on SIO 1 */
volatile int intRolloverflag2 = 0; /* Flag indicates Processing buffer Rollover on SIO 2 */
volatile time_t Start_Time = 0; /* Used for "time-out" starting point. */
volatile time_t End_Time = 0; /* Used for "time-out" ending point. */
int temp_counter = 0; /* Counter is used inside the clock interrupt and gets incremented every millisecond */

/* This is used as a generic counter inside the code to initialize the AHRS */
int intSendCounter = 0;


#pragma on( check_stack );

const struct sigevent *isr_handler( void *arg, int id )

/**** INTERRUPT SERVICE ROUTINE /****
#pragma on( check_stack );
/* ERROR TRAPPING FOR THE INTERRUPT ATTACH */
extern int errno;
const struct sigevent *isr_handler( void *arg, int id )
/* A FLAG IS SET TO TRUE TO INDICATE THAT AN INTERRUPT HAS BEEN RECEIVED */
COMM1_InterruptFlag = TRUE;

/* A NULL VALUE IS RETURNED FROM THE FUNCTION INSTEAD OF A SIGNAL TYPE 
BECAUSE ONLY ONE INTERRUPT IS BEING MONITORED AND ONLY ONE SIMPLE 
EVENT WILL BE PERFORMED EACH TIME THE INPUT BUFFER IS READ */

return(&event);

/******************+*****txm:i:*******x*************************************
** INTERRUPT CLOCK ROUTINE
const struct sigevent *clock_handler( void *area, int id )
{
static int clock_counter = 0;
temp_counter++;

// Counter used to monitor how fast the State Vector is sent.
intSendCounter++;

// Wake up the thread every 5'th interrupt
if ( ++clock_counter == 5 )
{
    clock_counter= 0;
    return(&clock_event);
}
else
{
    return( NULL );
}
}

/******************+*****txm:i:*******x*************************************
** INTERRUPT SERVICE ROUTINE
// This thread is dedicated to handling and managing Serial interrupts
void * int_thread ( void *arg)
/*...........................................................................
Input: argc: Optional argument
Output: None
This function creates a thread for serial port hardware interrupt processing. The thread is waiting for the 
Serial Port ISR to wake it up. When the ISR wakes it up, it checks to see if there is any data waiting to be 
read from the Serial Port (in this case, it is SIO1). The routine reads the data and saves it in a circular 
buffer: "strProcBuffer1". Pointers are incremented in accordance with the location of the next slot into 
which data is to be read into the buffer. Buffer rollovers are also kept track of. The routine also invokes a 
timer function which works in conjunction with a call to the timer function inside the main routine to keep 
a software watchdog timer active.*/
void * int_thread (void *arg) {

  // INITIALIZATIONS AND DECLARATIONS
  unsigned char strInputBuffer1[INPUTBUFSIZE];  /* Input Buffer for serial input on COM1 */
  int intSize_Read1 = 0;  /* Return Value for Device Read */
  int intSize_Waiting1 = 0;  /* Return value for number of characters in the UART waiting to be read */
  int intreadcount1 = 0;  /* Temporary variable used for computation */
  strInputbufptr1 = &strInputBuffer1[0];  /* Points to the first element of the Input Buffer */

  // enable I/O privilege
  ThreadCtl (_NTO_TCTL_IO, NULL);

  // Event Structure initialization
  event.sigev_notify = SIGEV_INTR;

  // attach the ISR to IRQ 4
  InterruptAttach(IRQ4, isr_handler, NULL, 0, 0);

  // perhaps boost this thread's priority here

  while (1) {
    InterruptWait (NULL, NULL);
    // at this point, when InterruptWait unblocks,
    // the ISR has returned a SIGEV_INTR, indicating
    // that some form of work needs to be done.

    // do the work
    intSize_Read1 = 1;

    /* READ THE H/W INPUT BUFFER */
    while (intSize_Read1 != 0) {
      intSize_Read1 = 0;

      /* Check the Input Buffer to see if any characters are waiting to be read */
      intSize_Waiting1 = rdchk (intCOMM1);
      if (intSize_Waiting1 > 0) {
        /* READ THE INPUT BUFFER */
        intSize_Read1 = read(intCOMM1, strInputBuffer1, intSize_Waiting1);
      }
    }

    /* LOADING THE PROCESSING BUFFER WITH THE Bytes FROM THE SERIAL PORT */
    intreadcount1 = intSize_Read1;
    while (intreadcount1 != 0) {
      *strProccurrentptr1 = (*strInputbufptr1) & 0xff;
      strProccurrentptr1++;
      strInputbufptr1++;
    }
}

if (strProccurrentptr1 == strProcbufendptr1) {
    strProccurrentptr1 = strProcbufstartptr1;
    intRolloverflag1++;
}
    intreadcount1--;
}
strInputbufptr1 = &strInputBuffer1[0];

// IF THE ISR_HANDLER DID AN INTERRUPTMASK, THEN THIS THREAD SHOULD
// DO AN INTERRUPTUNMASK TO ALLOW INTERRUPTS FROM THE HARDWARE
Start_Time = time (NULL);
End_Time = Start_Time;

void *clock_int_thread (void *arg)
Input: argc: Optional argument
Output: None

This function creates a thread for clock interrupt processing. The thread is waiting for the Clock ISR to
wake it up. When the Clock ISR wakes it up, it checks to see if there is any data waiting to be read from the
Serial Port (in this case, it is SIO2). If there is any data to be read from the Serial port, the routine reads the
data and saves it in a circular buffer: "strProcBuffer2". Pointers are incremented in accordance with the
location of the next slot into which data is to be read into the buffer. Buffer rollovers are also kept track of.

void *clock_int_thread (void *arg)

// this thread is dedicated to handling and managing Clock based interrupts
void *clock_int_thread (void *arg) {

// INITIALIZATIONS AND DECLARATIONS
unsigned char strInputBuffer2[INPUTBUFSIZE2]; /* Input Buffer for serial input on COM2 */
int intSize_Read2 = 0; /* Return Value for Device Read. */
int intSize_Waiting2 = 0; /* Return value for number of characters in the UART waiting to be read */
int intreadcount2 = 0; /* Temporary variable used for computation */
strInputbufptr2 = &strInputBuffer2[0]; /* Points to the first element of the Input Buffer */

// Create thread for Clock interrupt processing
    // Request I/O privity
    ThreadCtl (_NTO_TCTL_IO, 0);

    // Event Structure initialization
clock_event.sigev_notify = SIGEV_INTR;

// Attach ISR vector
InterruptAttach( SYSPAGE_ENTRY(qtime)->intr, &clock_handler, NULL, 0, 0 );

while (TRUE) {
// Wait for ISR to wake us up
    InterruptWait( 0, NULL );

        // do the work
        intSize_Read2 = 1;

/* READ THE H/W INPUT BUFFER */
    while (intSize_Read2 != 0) {
        intSize_Read2 = 0;

/* Check the Input Buffer to see if any characters are waiting to be read */
        intSize_Waiting2 = rdchk (intCOMM2);

            if (intSize_Waiting2 > 0) {
/* READ THE INPUT BUFFER */
                intSize_Read2 = read(intCOMM2, strInputBuffer2, intSize_Waiting2);
            }

/* LOADING THE PROCESSING BUFFER WITH THE BYTES FROM THE SERIAL PORT 2 */
        intreadcount2 = intSize_Read2;

            while (intreadcount2 != 0) {
                *strProcurrentptr2++ = *strInputbufptr2++;

                if (strProcurrentptr2 == strProbufendptr2) {
                    strProcurrentptr2 = strProbufstartptr2;
                    intRolloverflag2++;   
                }   
                intreadcount2--;
            }
        strInputbufptr2 = &strInputBuffer2[0];

} // END WHILE (intSize_Read2 != 0)

} // END While (TRUE)

} // END clock_int_thread()

 Rolloverflag2++;   

} // END While (TRUE)

} // END clock_int_thread()

/********************Nguồn liên quan đến chương trình主程序*************************************/

*  MAIN PROGRAM
  **************************************************/

/****************************------------------
int main(int argc, char *argv[])
Input:  argc: Optional argument count
     argv[]: Optional array of input line arguments

Output:  None

This function initializes all the required program parameters. It opens the Serial Ports, sets up all the
required multithreading and instates the ISR capability in the system. It also performs the requisite
background processing of the GPS data in the program. In addition, it runs a timer algorithm that is
supported by the Serial Port ISR to check for the availability of the GPS data and would exit the process if
there is no GPS data in any 5 second interval

void InitSerialPort(void);
void SetRCVRPortRate(int);
void SetSerialRate(void);
void UNLOG_ALLRequest( int );
void SendPositionRequest( int );
void SendVelocityRequest( int );

int main(int argc, char *argv[]) {
    volatile time_t Time_Critical = 5;  /* Number of seconds before while loop times out. */
    int intTimeFlag = TRUE;  /* Time Flag is set to true to prime while loop. */

    unsigned char test_byte[2];
    /* Test Byte */

    /* INITIALIZING THE CHARACTER POINTER VARIABLES FOR SERIAL I/O 1 */
    strProcbufstartptr1 = &strProcBuffer1[0];
    strProcscanptr1 = &strProcBuffer1[0];
    strProccurrentptr1 = &strProcBuffer1[0];
    strProcbufendptr1 = &strProcBuffer1[0] + PROCBUFSIZE;
    strComparisonbufptr = &strComparisonbuf[0];

    /* INITIALIZING THE CHARACTER POINTER VARIABLES FOR SERIAL I/O 2 */
    strProcbufstartptr2 = strProcBuffer2;
    strProcscanptr2 = strProcBuffer2;
    strProccurrentptr2 = strProcBuffer2;
    strProcbufendptr2 = strProcBuffer2 + PROCBUFSIZE;

    //********** ADD THE COMM PORTS TO THE SERIAL DEVICE DEVC-SER8250 (DPB) **********

    system("/sbin/devc-ser8250 3f8,3 2f8,3 10b0,9 10b8,9 &");
    delay(1000);

    //**** NOW INITIALIZE THE COMM PORTS *****

    /* Initialize the COM1 serial port. */
    InitSerialPort();
/* Open the COM1 serial port */
intCOMM1 = open("/dev/ser1", O_RDWR);

if( intCOMM1 < 0 )
{
    printf("\n\nCOM1 NOT opened!\n\n");
    exit(1);
}

/* Set the RCVR rate to 115200 bps. */
SetRCVRPortRate(intCOMM1);

/* Set COM1 rate to 115200 bps. */
SetSerialRate();

***** REQUEST POSITION AND VELOCITY LOGS FOR RCVR *****/

printf("Requesting Position and Velocity strings.\n");
UNLOG_ALLRequest( intCOMM1 );
SendPositionRequest(intCOMM1 );
SendVelocityRequest( intCOMM1 );

/* INITIALIZE THE COMM2 SERIAL PORT. */
printf("Initializing COMM2 Serial Port. \n\n");
system("stty baud = 38400 < /dev/ser2");
system("stty -echo -echoe -echok -echonl < /dev/ser2");
system("stty -ihflow -ohflow < /dev/ser2");
system("stty -ohpaged < /dev/ser2");
system("stty +raw < /dev/ser2");
printf("COMM2 Initialized, Baud Rate = 38400 \n\n");

/* OPEN COMM2 SERIAL PORT */
intCOMM2 = open("/dev/ser2", O_RDWR);

if( intCOMM2 < 0 )
{
    printf(" Unable to open Serial I/O Port 2 !!!!! \n");
    exit(1);
}
else {
    printf("Serial Port 2 Opened\n\n");
    fflush(stdin);
}

/*****************************/

*** SET UP THE COMM PORT FOR OUTPUT TO SVS COMM 3 ***
/*****************************/

/* Initialize the COM3 serial port. */
system("stty baud=115200 < /dev/ser3");
system("stty -echo -echoe -echok -echonl < /dev/ser3");
system("stty -ihflow -ohflow < /dev/ser3");
system("stty -ohpaged < /dev/ser3");
system("stty +raw < /dev/ser3");

intCOMM3 = open("/dev/ser3", O_WRONLY);

/* Insure serial port is opened. If not exit. */
if (intCOMM3 < 0)
{
    printf("Serial Port Not Opened. Check Connections and try again.\n");
    exit(1);
}
else
{
    printf("Commport 3 ready.\n");
}
delay(2000);

/* Initialize the COM4 serial port. */
system("stty baud=115200 < /dev/ser4");
system("stty -echo -echoe -echok -echonl < /dev/ser4");
system("stty -ihflow -ohflow < /dev/ser4");
system("stty -ohpaged < /dev/ser4");
system("stty +raw < /dev/ser4");

intCOMM4 = open("/dev/ser4", O_WRONLY);

/* Insure serial port is opened. If not exit. */
if (intCOMM4 < 0)
{
    printf("Serial Port Not Opened. Check Connections and try again.\n");
    exit(1);
}
else
{
    printf("Commport 4 ready.\n");
}
delay(2000);
The following code tries to initialize the AHRS. If the AHRS does not initialize it continues
// but lets the user know that the AHRS did not initialize

```c
    test_byte[0] = 0;
    write(intCOMM2, "R", 1);
```

```c
    temp_counter = 0;
```

// Code to reset the AHRS and initialize the Same

```c
    while (temp_counter < 20000 && test_byte[0] != 'H'){
        read(intCOMM2, &test_byte[0], 1);
        temp_counter++;
    }
```

```c
    printf(" %c\n",test_byte[0]);
```

```c
    if (test_byte[0] != 'H') {
        printf(" Enter a character to continue. AHRS did not respond to the request for system reset \n ");
        getchar();
    }
```

```c
    test_byte[0] = 0;
    write(intCOMM2, "a", 1);
```

```c
    temp_counter = 0;
```

// Request the AHRS to send data in the Angle or VG Mode

```c
    while (temp_counter < 20000 && test_byte[0] != 'A') {
        read(intCOMM2, &test_byte[0], 1);
        temp_counter++;
    }
```

```c
    printf(" %c\n",test_byte[0]);
```

```c
    if (test_byte[0] != 'A') {
        printf(" Enter a character to continue. AHRS did not respond to the request to change to Angle
Data Packet Mode \n ");
        getchar();
    }
```

// Request the AHRS to send continuous data

```c
    test_byte[0] = 0;
    write(intCOMM2, "C", 1);
```

```c
    temp_counter = 0;
```

// Create thread for Serial Interrupt processing
```
    pthread_create(NULL, NULL, int_thread, NULL);
```

// Create thread for Clock Interrupt processing
```c
pthread_create(NULL, NULL, clock_int_thread, NULL);

// INITIALIZING THE TIMING VARIABLE
Start_Time = time(NULL);

while (intTimeFlag == TRUE) {
    // CODE TO PROCESS THE DATA FROM SERIAL PORT 1 (GPS)
    if (ObtainGPSData()) {
        strcpy(strGPSdata, strComparisonbuf);
        ProcessGPSData(strGPSdata, sizeof(strGPSdata));
    }

    // CODE TO PROCESS THE DATA FROM SERIAL PORT 2 (AHRS)
    if (ProcessAHRSData()) {
        strcpy(strAHRSdata, strComparisonbuf);
        dblAHRS_RollRaw = (strAHRSdata[1] & 0xff) << 8 | (strAHRSdata[2] & 0xff);
        dblAHRS_PitchRaw = (strAHRSdata[3] & 0xff) << 8 | (strAHRSdata[4] & 0xff);
        dblAHRS_YawRaw = (strAHRSdata[5] & 0xff) << 8 | (strAHRSdata[6] & 0xff);

        dblAHRS_Roll = ScaleAHRSData(dblAHRS_RollRaw);
        dblAHRS_Pitch = ScaleAHRSData(dblAHRS_PitchRaw);

        //*** ROLL AND PITCH RANGE FROM 0 TO 360, CONVERT TO 0 TO -180 AND 0 TO +180. ***
        if (dblAHRS_Roll > 180 && dblAHRS_Roll < 360) {
            dblAHRS_Roll = dblAHRS_Roll - 360;
        }

        if (dblAHRS_Pitch > 180 && dblAHRS_Pitch < 360) {
            dblAHRS_Pitch = dblAHRS_Pitch - 360;
        }

        sprintf(strStateRoll,"%7.2f",dblAHRS_Roll);
        sprintf(strStatePitch,"%7.2f",dblAHRS_Pitch);
    }

    if (intSendCounter >= 70) // 30 millisecond wait time.
    {
        // send State Vector Packet then reset the count to zero.
        StateVectorDataToStateVectorString();

        /* Count data characters in the state vector string. */
Kcount = strlen(strStateVector);

/**********************
* CREATE THE DATA PACKET *
***************************/

/* Build the Header to the buffer string for the output data packet */
outputbuffer[0] = Header[0];
outputbuffer[1] = Header[1];
outputbuffer[3] = Header[3];

/* Add the buffer string to the outputbuffer string. */
for (Xcount = 4; Xcount < Kcount + 6; Xcount++)
{
    outputbuffer[Xcount] = strStateVector[Xcount-4];
}

/* Add the tail of the data packet. */
outputbuffer[strlen(outputbuffer) - 2] = Tail[0];
outputbuffer[strlen(outputbuffer) - 1] = Tail[1];

intSendCounter = 0;

/******************************
* SEND DATA TO COMM PORT 3 *
******************************/

intSizeWrite = write( intCOMM3, &outputbuffer, Kcount + 6 );

if (intSizeWrite < 0)
{
    perror("Commport 3 Error, no data sent!n\n");
}
else
{
    //Clear the output buffer.
    //strcpy(outputbuffer, ");
}

/******************************
* SEND DATA TO COMM PORT 4 *
******************************/

intSizeWrite4 = write( intCOMM4, &outputbuffer, Kcount + 6 );

if (intSizeWrite4 < 0)
{
    perror("Commport 4 Error, no data sent!
");
}
else
{
    //Clear the output buffer.
    strcpy(outputbuffer, "");
}

/* CALCULATE THE TIME BETWEEN INTERRUPTS */
End_Time = time( NULL );

/* TIME-OUT WHILE LOOP IF CRITICAL TIME IS REACHED */
if( difftime( End_Time, Start_Time ) > Time_Critical ){
    intTimeFlag = FALSE;
    printf("NO INTERRUPT RECEIVED FOR PAST \%d SECONDS FROM THE GPS_UNIT \n", Time_Critical);
    printf("PROGRAM HAS TIMED OUT!\n\nNumber of interrupts received: %d\n\nStart Time \%u \n", Start_Time);
    printf("End Time \%u \n", End_Time);
}

} // END WHILE (intTimeFlag == TRUE)

printf("Exiting the System\n");
fflush(stdin);
fcloseall();
return;

} // END main()

#ifndef End of Main #####################################

//ObtainGPSData(void)

int ObtainGPSData(void)
{
    static int HDRretval = 0; /* return flag of the GPS Header Scan function */
    int CRCretval = 0; /* return flag to indicate the result of the CRC */
    int ptrdiff = 0; /* Pointer difference counter */
    int tempptrdiff = 0; /* Temporary pointer difference counter */

    Input: None
    Output: retval: The number of bytes in the successfully decoded GPS ASCII data string
            (No CRC or Checksums, searching for character termination)
    This function parses the GPS Data string obtained from the Serial port and return a value that is equal to the
    length of the GPS string(along with the CR and LF appended to the end). The contents of the GPS string
    are in the variable "strComparisonbuf"
    *----------------------------------------------------------------------------------------------------

    int ObtainGPSData(void)
    {
        //...
int retval = 0; /* Function return value */

static int LostGPSPacketcount = 0; /* Keeps track of lost GPS Data packets */
static int RcvdGPSPacketcount = 0; /* Keeps track of lost GPS Data packets */
static int loopentryflag = 1; /* Initial value of the flag when we enter the loop to scan the buffer */
static unsigned char * strtempProcscanptr1; /* Temporary scanner pointer used inside the loop */

if (!HDRretval) {
    if (ptrdiff = Pointer_Diff((int)strProcscanptr1,(int)strProccurrentptr1,PROCBUFSIZE) ) >= 10 )
    {
        // Do the Sync Byte comparison here
        Get_Bytes(strProcscanptr1, strProcbufstartptr1, strProcbufendptr1, PROCBUFSIZE, 1);
        HDRretval = SearchGPSHeader (strComparisonbuf);

        if (HDRretval != 1) {
            strProcscanptr1++;
            if (strProcscanptr1 == strProcbufendptr1 & intRolloverflag1 == 1) {
                strProcscanptr1 = strProcbufstartptr1;
                intRolloverflag1--;
            }
        } // end (HDRretval != 1)

    } // end Pointer_Diff

    } // end (!HDRretval)

else if (HDRretval) {
    if (loopentryflag == 1) {
        loopentryflag = 0;
        strtempProcscanptr1 = strProcscanptr1 + 1;
    }

    if (strtempProcscanptr1 == strProcbufendptr1) {
        strtempProcscanptr1 = strProcbufstartptr1;
    }

    } // Data is ASCII

if ( (ptrdiff = Pointer_Diff((int)strProcscanptr1,(int)strProccurrentptr1,PROCBUFSIZE)) >= 5 ) {
if ((*strProcscanptr1&0xff) == '#') {
    ptrdiff = Pointer_Diff((int)strProcscanptr1,(int)strProcscanptr1,PROCBUFSIZE);
    while (ptrdiff--) {
        strProcscanptr1++;
    }
if (strProcscanptr1 >= strProcbufendptr1 && intRolloverflag1 == 1) {
    strProcscanptr1 = strProcbufstartptr1;
    intRolloverflag1--;
}

HDRretval = 0;
loopentryflag = 1;
LostGPSPacketcount++;
// Begin test code added by Sai 04/30/03 !!!!!!!!!!!!!
printf("BBB ");
// End test code added by Sai 04/30/03 !!!!!!!!!!!!!
// Begin test code added by Sai 04/30/03 !!!!!!!!!!!!!
// printf("%d
",LostGPSPacketcount);
// End test code added by Sai 04/30/03 !!!!!!!!!!!!!
}
else if (*strtempProcscanptr1&0xff) == "+") {

    strComparisonbufptr = &strComparisonbuf[0];

    ptdiff = (Ptr_Diff((int)strProcscanptr1,(int)strtempProcscanptr1,PROCBUFSIZE)) + 1;

    // Has length enough to append the CR and LF at the end of the line
    tempptdiff = ptdiff + 1;

    while (ptdiff--) {
        *strComparisonbufptr++ = *strProcscanptr1++;
        if (strProcscanptr1 >= strProcbufendptr1 && intRolloverflag1 == 1) {
            strProcscanptr1 = strProcbufstartptr1;
            intRolloverflag1--;
        }
    }

    // Append a Carriage return and a line feed to the end of line
    *strComparisonbufptr = 0xd;
    *strComparisonbufptr = 0xa;

    HDRretval = 0;
    loopentryflag = 1;
    RcvdGPSPacketcount++;

    // AT THIS POINT THE DATA IN THE STRING "strComparisonbuf" IS TO BE TAKEN FROM THE
    // FIRST ELEMENT
    // TILL WE SEE A CARRIAGE RETURN AND A LINE FEED

    // Begin test code added by Sai 04/30/03 !!!!!!!!!!!!!
    // printf("%d
",RcvdGPSPacketcount);
    // End test code added by Sai 04/30/03 !!!!!!!!!!!!!
    retval = tempptdiff;
}
else if ( (ptrdiff = StrPtrDiff(int strProcscanptr1,(int)strProcCurrentPtr1.PROCBUFSIZE)) >= 700 ) {
    while ( ptrdiff--) {
        strProcscanptr1++;  
        if ( strProcscanptr1 >= strProcbufendptr1 && int Rolloverflag1 == 1 ) {
            strProcscanptr1 = strProcbufstartptr1;  
            int Rolloverflag1--;  
        }
        HDRretval 0;  
        loopentryflag 1;  
    }
}
else {
    strtempProcscanptr1++;
    if ( strtempProcscanptr1 == strProcbufendptr1 ) {
        strtempProcscanptr1 = strProcbufstartptr1;  
    }
}
} // end (ptr_diff >= 10)
} // end else if (HDRretval)
return retval;
} // end function

/*..............................................................................................................................
int ProcessAHRSData(void)

Input: None
Output: intchksum_status:

The status of the checksum algorithm inside the ProcessAHRSData algorithm. If this is zero, a valid data packet has not been found. If this is one, a valid data packet has been found.

This function parses the AHRS Data string obtained from the Serial port. It checks to see if the AHRS header is found in the Serial Port data. Upon finding the header, it proceeds to obtain the AHRS data packet. It runs a checksum test on it and strips the incoming buffer off to obtain the AHRS data packet. The data packets are stored in the buffer "strComparisonbuf".
..............................................................................................................................*/

int ProcessAHRSData(void) {

    unsigned char strdatabuffer2[NUMDATABLES_IN_VMODE + 2];

    static int hdr_found = 0;  /* Flag to
    indicate the detection of the header in the data packet */
int ii = 0;  // Counter variable for looping */
int intchksum_status = 0;  /* Pass or Fail
Status of the checksum computation routine */

if (!hdr_found) {
    if (Ptr_Diff((int)strProcscanptr2, (int)strProccurrentptr2, PROCBUFSIZE) >= 1) {
        // Do the Sync Byte comparison here
        Get_Bytes(strProcscanptr2, strProcbufstartptr2, strProcbufendptr2, PROCBUFSIZE, 2);
        if (strComparisonbuf[0] && 0xff == 0xff) {
            hdr_found = 1;
        } else {
            strProcscanptr2++;
            if (strProcscanptr2 >= strProcbufendptr2 && intRolloverflag2 == 1) {
                strProcscanptr2 = strProcbufstartptr2;
                intRolloverflag2--;;
            }
        }
    } // End (ptrdiff())
}

if (hdr_found) {
    // Data is Binary
    if ((Ptr_Diff((int)strProcscanptr2, (int)strProccurrentptr2, PROCBUFSIZE)) >= PACKET_SIZE_VG_MODE) {
        Get_Bytes(strProcscanptr2, strProcbufstartptr2, strProcbufendptr2, PROCBUFSIZE, PACKET_SIZE_VG_MODE);
        intchksum_status = Compute_Chksum (strComparisonbuf, NUMDATABYTES_IN_VGMODE);
    } // Copy "strdatabuffer2[""] into array using "memcpy()" if necessary
    ii = PACKET_SIZE_VG_MODE;
} else {
    ii = 1;
}

while (ii--) {
    strProcscanptr2++;
    if (strProcscanptr2 >= strProcbufendptr2 && intRolloverflag2 == 1) {
        strProcscanptr2 = strProcbufstartptr2;
        intRolloverflag2--;;
    }
}
int Compute_CHKsum(char DataArray[], int size) {
    int jj = 0;
    int status = 0;
    unsigned long checksum = 0;

    // Begin test code added by Sai !!!!!
    static int packet_count = 0;
    // End test code added by Sai !!!!!

    while (jj < size) {
        checksum += DataArray[jj+1]&0xff;
        jj++;
    }
    checksum = checksum % 256;

    if (checksum == (DataArray[size+1]&0xff)) {
        packet_count++;
        status = TRUE;
    }
    // Begin test code added by Sai !!!!!
    if (temp_counter > 1000) {
        temp_counter = 0;
        // printf("%d \n",packet_count);
        packet_count = 0;
    }
    // End while (ii--)
    hdr_found = 0;
    // End ptrdiff()
    // End else if (hdr_found)
    return intchksum_status;
} //END ProcessAHRSData()
1

else {
    status = FALSE;
}

return status;

/***** PORT AND RCVR INITIALIZATIONS ******/
void InitSerialPort( void )
{
    /* Initialize the COM1 serial port. */
    system("stty baud=9600 < /dev/ser1");
    system("stty -echo -echoe -echonl < /dev/ser1");
    system("stty -ihflow -ohflow < /dev/ser1");
    system("stty -ohpaged < /dev/ser1");
    system("stty +raw < /dev/ser1");

    printf("COM1 initialized, Baud Rate = 9600 bps.");
}

void SetSerialRate( void )
{
    system("stty baud=115200 < /dev/ser1");
    printf("COM1, Baud Rate = 115200 bps.");
}

void SetRCVRPortRate( int intSerialPort )
{
    int intSize_Write = 0;
    char strRCVRPortInit[] = "com com1 115200 N 8 1 N off on
    ";

    intSize_Write = write( intSerialPort, &strRCVRPortInit, size of (strRCVRPortInit) );

    if( intSize_Write < 0 )
    {
        perror("Could NOT set RCVR rate to 115200 bps!!!\n");
        exit(1);
    }

    else
    {
        printf("RCVR rate set to 115200 bps.\n");
        delay(100);
    }
}

void UNLOG_ALLRequest( int intSerialPort )
{
int intSize_Write = 0;
int XIndex = 0;
char strUNLOG[] = "unlogall\n\r";

for( XIndex = 0 ; XIndex < 2 ; XIndex++ )
{
    intSize_Write = write( intSerialPort, &strUNLOG, sizeof(strUNLOG));

    if( intSize_Write < 0 )
    {
        perror("Could NOT UNLOG previous request!!\n");
        exit(1);
    }

    else
    {
        printf("Unlogging all previous request.\n");
        delay(100);
    }
}

void SendPositionRequest( int intSerialPort )
{
    int intSize_Write = 0;
    char strPositionRequest[] = "log com1 bestposa ontime 0.05\n\r";

    intSize_Write = write( intSerialPort, &strPositionRequest, sizeof(strPositionRequest));

    if( intSize_Write < 0 )
    {
        perror("Could not send Position Request!\n");
        exit(1);
    }

    else
    {
        printf("Position Request Sent.\n");
        delay(100);
    }
}

void SendVelocityRequest( int intSerialPort )
{
    int intSize_Write = 0;
    char strVelocityRequest[] = "log com1 bestvela ontime 0.05\n\r";

    intSize_Write = write( intSerialPort, &strVelocityRequest, sizeof(strVelocityRequest));

    if( intSize_Write < 0 )
    {
        perror("Could not send Velocity Request!\n");
    }
else
{
    printf("Velocity Request Sent \n");
    delay(100);
}
exit(1);
A.2 Coordinate Conversions

Author: Douglas Burch
Date: 11 Feb 2003
File Name: WGSConversion.h

NOTE: M_PI is used, 4*tan(1) = PI

When compiling using the math.h lib use the "-l m" with cc.

*******************************************************************************/
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

*******************************************************************************/

FUNCTION PROTOTYPES
*******************************************************************************/

void WGSlla2xyz( double *, double * );
void WGSxyz2enu( double *, double *, double * );

*******************************************************************************/

FUNCTION DECLARATIONS
*******************************************************************************/

FUNCTION: void WGSlla2xyz( double *, double *)
Purpose:

Convert WGS-84 Latitude, Longitude, and Altitude to WGS-84 XYZ coordinates (meters).
The WGS-84 Latitude and Longitude are expected in degrees and the Altitude above
the WGS-84 ellipsoid is expected in meters.

* Convention:
  * North Latitude (+)
  * South Latitude (-)
  * East Longitude (+)
  * West Longitude (-)

* Input: 1x3 vector containing Latitude (deg), Longitude (deg), Altitude (meters).
  * Output: 1x3 vector containing X, Y, Z in meters.

* Reference:
  * Matlab Function xyz = wgslla2xyz(lat, lon, alt)
  * Ref: Decker, B. L., World Geodetic System 1984,
  * Defense Mapping Agency Aerospace Center.

*******************************************************************************/

void WGSlla2xyz(double *LLA, double *XYZ)
{
  double dblSemiMajorAxis = 6378137;
double dblFlattening = 1/298.257223563;
double dblEccentricitySquared = (2 - dblFlattening) * dblFlattening;
double DEG2RAD = M_PI / 180.0;
double dblLAT = 0;
double dblLON = 0;
double dblALT = 0;
double dblX = 0;
double dblY = 0;
double dblZ = 0;
double dblCosLAT = 0;
double dblSinLAT = 0;
double dblRn = 0;

dblLAT = LLA[0];
dblLON = LLA[1];
dblALT = LLA[2];

dblSinLAT = sin( dblLAT * DEG2RAD );
dblCosLAT = cos( dblLAT * DEG2RAD );

dblRn = dblSemiMajorAxis / sqrt( 1 - dblEccentricitySquared * dblSinLAT * dblSinLAT );

dblX = ( dblRn + dblALT ) * dblCosLAT * cos( dblLON * DEG2RAD );
dblY = ( dblRn + dblALT ) * dblCosLAT * sin( dblLON * DEG2RAD );
dblZ = ( dblRn * ( 1 - dblEccentricitySquared) + dblALT) * dblSinLAT;

/* ERROR CHECK */

XYZ[0] = dblX;
XYZ[1] = dblY;
XYZ[2] = dblZ;

} // END FUNCTION WGSLLA2XYZ

******************************************************************************
* FUNCTION:    void WGSxyz2enu( double *refLLA, double *XYZ, double *ENU )  *
* Purpose:     Converts WGS 84 XYZ coordinates to ENU system with respect to a  *
*               given LLA.                                                *
*               *
* Convention:  *
* North Latitude (+) *
* South Latitude (-) *
* East Longitude (+) *
* West Longitude (-) *
* *
* Input:  1x3 vector containing the reference LLA.                     *
* 1x3 vector containing the XYZ coordinate.                            *
* *
* Output:  1x3 vector containing the ENU coordinates in meters.         *
* *
* Reference:  Matlab Function xyz = wgslla2xyz(lat, lon, alt)
Ref: Decker, B. L., World Geodetic System 1984.
Defense Mapping Agency Aerospace Center.

void WGSxyz2enu(double *LLA, double *XYZ, double *ENU)
{
    double dblSemiMajorAxis = 6378137;
    double dblFlattening = 1/298.257223563;
    double dblEccentricitySquared = (2 - dblFlattening) * dblFlattening;
    double DEG2RAD = M_PI / 180.0;
    double dblrefLAT = 0;
    double dblrefLON = 0;
    double dblrefALT = 0;
    double dblLAT = 0;
    double dblLON = 0;
    double dblALT = 0;
    double dblrefX = 0;
    double dblrefY = 0;
    double dblrefZ = 0;
    double dblX = 0;
    double dblY = 0;
    double dblZ = 0;
    double dblDiff[3];
    double dblCosLAT = 0;
    double dblSinLAT = 0;
    double dblCosRotAngleR1 = 0;
    double dblSinRotAngleR1 = 0;
    double dblCosRotAngleR2 = 0;
    double dblSinRotAngleR2 = 0;
    double dblRn = 0;
    double R[3][3];
    double R1[3][3];
    double R2[3][3];

    /* CONVERT THE REFERENCE LLA TO REFERENCE XYZ */
    dblLAT = LLA[0];
    dblLON = LLA[1];
    dblALT = LLA[2];

    dblSinLAT = sin( dblLAT * DEG2RAD );
    dblCosLAT = cos( dblLAT * DEG2RAD );
    dblRn = dblSemiMajorAxis / sqrt( 1 -dblEccentricitySquared * dblSinLAT * dblSinLAT );
    dblrefX = ( dblRn + dblALT ) * dblCosLAT * cos( dblLON * DEG2RAD );
    dblrefY = ( dblRn + dblALT ) * dblCosLAT * sin( dblLON * DEG2RAD );
    dblrefZ = ( dblRn * ( 1 - dblEccentricitySquared ) + dblALT ) * dblSinLAT;

    /* END REFERENCE CONVERSION TO XYZ */

    /* DIFFERENCE THE REFERENCE POINT FROM THE XYZ LOCATION OF INTEREST */
    dblDiff[0] = XYZ[0] - dblrefX;
I* ROTATE THE DIFFERENCE VECTOR TO THE ENU FRAME */

// Initialize the Identity Matrices R1 and R2.
R1[0][0] = 1; R1[0][1] = 0; R1[0][2] = 0;
R1[1][0] = 0; R1[1][1] = 1; R1[1][2] = 0;
R1[2][0] = 0; R1[2][1] = 0; R1[2][2] = 1;
R2[0][0] = 1; R2[0][1] = 0; R2[0][2] = 0;
R2[1][0] = 0; R2[1][1] = 1; R2[1][2] = 0;
R2[2][0] = 0; R2[2][1] = 0; R2[2][2] = 1;

// Find the cos and sin of the angle to rotate this is 90 plus the latitude or minus the latitude.

dblCosRotAngleR1 = cos((90 + dbILON) * DEG2RAD);
dblSinRotAngleR1 = sin((90 + dbILON) * DEG2RAD);

dblCosRotAngleR2 = cos((90 - dbILAT) * DEG2RAD);
dblSinRotAngleR2 = sin((90 - dbILAT) * DEG2RAD);

// Perform the 3-axis rotation.
R1[0][0] = dblCosRotAngleR1;
R1[1][1] = dblCosRotAngleR1;
R1[0][1] = -dblSinRotAngleR1;
R1[0][0] = dblSinRotAngleR1;
R1[0][1] = dblSinRotAngleR1;

// Perform the 1-axis rotation.
R2[1][1] = dblCosRotAngleR2;
R2[1][2] = dblCosRotAngleR2;
R2[1][2] = dblSinRotAngleR2;
R2[1][1] = -dblSinRotAngleR2;

// Multiply R by R1.

R[0][0] = R2[0][0]*R1[0][0] + R2[0][1]*R1[1][0] + R2[0][2]*R1[2][0]; // Row1 Column1
R[0][1] = R2[0][0]*R1[0][1] + R2[0][1]*R1[1][1] + R2[0][2]*R1[2][1]; // Row1 Column2
R[0][2] = R2[0][0]*R1[0][2] + R2[0][1]*R1[1][2] + R2[0][2]*R1[2][2]; // Row1 Column3

R[1][0] = R2[1][0]*R1[0][0] + R2[1][1]*R1[1][0] + R2[1][2]*R1[2][0]; // Row2 Column1
R[1][1] = R2[1][0]*R1[0][1] + R2[1][1]*R1[1][1] + R2[1][2]*R1[2][1]; // Row2 Column2
R[1][2] = R2[1][0]*R1[0][2] + R2[1][1]*R1[1][2] + R2[1][2]*R1[2][2]; // Row2 Column3

R[2][0] = R2[2][0]*R1[0][0] + R2[2][1]*R1[1][0] + R2[2][2]*R1[2][0]; // Row3 Column1
R[2][1] = R2[2][0]*R1[0][1] + R2[2][1]*R1[1][1] + R2[2][2]*R1[2][1]; // Row3 Column2
R[2][2] = R2[2][0]*R1[0][2] + R2[2][1]*R1[1][2] + R2[2][2]*R1[2][2]; // Row3 Column3

// Multiply R (rotation matrix) by the difference vector to get the ENU coordinates in meters.
ENU[0] = R[0][0]*dblDiff[0] + R[0][1]*dblDiff[1] + R[0][2]*dblDiff[2];  // East in meters.

} // END FUNCTION WGSXYZ2ENU
### A.3 AHRS Scaling Functions

```c
#include <fcntl.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "SVSexternvars.h"

/******** FUNCTION PROTOTYPES **********/
double ScaleAhrsData(double);  // (1)

/******** FUNCTION DECLARATIONS **********/

/*
 * (1)
 * ScaleAhrsData
 * The received AHRS data for the roll, pitch, and yaw is delivered in
 * a digital packet in angle mode. This digital packet must be converted
 * to an angle by multiplying the particular parameter by the relation
 * described below.
 */

double ScaleAhrsData( double DataIn )
{
    double Angle = 0;

    Angle = ( DataIn * 180 ) / pow( 2.15 );

    return Angle;
}
```
A.4 Synthesized Attitude Functions

/* ***********************************************************************/
/* Author: Douglas Burch, all functions contained within file PseudoAttitude.h */
/* Date: 12 Sept 2001 */
/* Modified: 19 Feb 2003 */
/* File Name: PseudoAttitude.h */
/* Purpose: Calculate the pseudo roll, flight path angle and the associated */
/* parameters involved with each calculation. */
/***********************************************************************/

#include <fcntl.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "SVSexternvars.h"

/****** FUNCTION PROTOTYPES *******/
int POS_StringID( char * ); // (1)
int VEL_StringID( char * ); // (2)
int TimeStampCheck(void); // (3)
void POS_DataExtract(char *); // (4)
void VEL_DataExtract( char *); // (5)
void SkipDataField(char *, int); // (6)
void StringDataToFloat(void); // (7)
void StateVectorDataToStateVectorString(void);
void CalculateVelocityVector(void); // (8)
void CalculateFlightPathAngle(void); // (9)
void CalculateAcceleration(void); // (10)
void CalculateGravity(void); // (11)
void CalculateLiftVector(void); // (12)
void CalculateReferenceVector(void); // (13)
void CalculatePseudoRoll(void); // (14)
void DisplayPositionData(void); // (15)
void DisplayVelocityData(void); // (16)
void PrintResults(void); // (17)
void GPSDataToFile( FILE *, char *, int);
void ClearPOSStrings(void);
void ClearVELStrings(void);
void ProcessGPSData( char *strSortedBuffer, int intBufferLength);

/****** FUNCTION DECLARATIONS *******/

/*
  * (1)
  * POS_StringID()
  * Function searches GPS input until a 'P' is located in the text. If
* a 'P' is found then the function searches for an 'O' and then
* likewise for an 'S'. If all three letters are found in sequence then
* it is assumed that a POSA header has been located and the function
* returns a logic 1.
*/
int POS_StringID( char *strBuffer )
{
  int intPOS_FLAG = 0;

  /* Identify POS message for parsing. */
  if( strBuffer[intMark] == 'P')
  {
    intMark++;
    if( strBuffer[intMark] == 'O')
    {
      intMark++;
      if( strBuffer[intMark] == 'S')
      {
        /* POS ident confirmed */
        //printf("Found POSn");
        intPOS_FLAG = 1;
      }
    }
  }
  /* End search for "POS" string. */

  return intPOS_FLAG;
}

/*
* (2)
* VEL_StringID()
* Function searches GPS input until an 'S' is located in the text. If
* an 'S' is found then the function searches for a 'P' and then
* likewise for an 'H'. If all three letters are found in sequence then
* it is assumed that an SPHA header has been located and the function
* returns a logic 1.
*/
int VEL_StringID( char *strBuffer )
{
  int intVEL_FLAG = 0;

  /* Identify VEL string for parsing. */
  if( strBuffer[intMark] == 'V')
  {
    intMark++;
  }
if( strBuffer[intMark] == 'E')
{
    intMark++;

    if( strBuffer[intMark] == 'L')
    {
        /* VEL ident confirmed. */
        //printf("Found VEL\n");
        intVEL_FLAG = 1;
    }
}

} /*End search for "VEL" string. */

return intVEL_FLAG;

/*
 (3)
 TimestampCheck()
 Reads the global time stamp strings for the POSA and the SPHA strings.
 If the strings correspond to the same time stamp then a 1 is returned
 from the function, else a zero is returned. The strings are used instead
 of the actual values because it is easier to compare a string than a
 floating point number.
 */
int TimestampCheck(void)
{
    int intMatch = 0;

    if( strcmp(strPOS_TimeStamp, strVEL_TimeStamp) == 0)
    {
        intMatch = 1;
    }

    return intMatch;
}

/*
 (4)
 POS_DataExtract()
 This function receives the FilePointer and then passes it to
 SkipDataField() so the second field (the Time Stamp) of the POS string
 can be read. The function gets character of the Time Stamp string
 until a comma signals the start of the longitude field. The latitude
 and height are handled in the same way.
 */
void POS_DataExtract(char *strBuffer)
{
char CharIn;
int intNumToSkip;
int intIndex = 0;

/*
 * The Time Stamp starts in the sixth field of the header
 * and is the number of GPS seconds that have elapsed since
 * the start of the week. The longitude information starts in
 * the 14th field of the POSA string. SkipDataField() skips
 * the first five fields of the string so the sixth one can
 * be read in.
 */

intNumToSkip = 5;
SkipDataField( strBuffer, intNumToSkip );

/* Extracting Time Stamp from POSA string. */
while( strBuffer[intMark] != ',')
{
    strPOS_TimeStamp[intIndex] = strBuffer[intMark];
    intIndex++;
    intMark++;
    intMark++;
}

/* Move to beginning of next field. */
intNumToSkip = 4;
SkipDataField( strBuffer, intNumToSkip );

/* Extracting Latitude from POSA string. */
intIndex = 0;
while( strBuffer[intMark] != ',')
{
    strLatitude[intIndex] = strBuffer[intMark];
    intIndex++;
    intMark++;
}

/* Extracting Longitude from POSA string. */
intMark++;
/* Move to beginning of next field. */
intIndex = 0;
while( strBuffer[intMark] != ',')
{
    strLongitude[intIndex] = strBuffer[intMark];
    intIndex++;
    intMark++;
}

/* Extracting Altitude from POSA string. */
intMark++;
/* Move to beginning of next field. */
intIndex = 0;
while( strBuffer[intMark] != ',')
{
This function receives the FilePointer and passes it to SkipDataField() so the second field (Time Stamp) of the VEL string can be read. The function gets characters of the Time Stamp string until a comma signals the start of the vertical speed field. The ground track and altitude are handled in the same way.

```c
/* (5) VEL_DataExtract() This function receives the FilePointer and passes it to SkipDataField() so the second field (Time Stamp) of the VEL string can be read. The function gets characters of the Time Stamp string until a comma signals the start of the vertical speed field. The ground track and altitude are handled in the same way. */

void VEL_DataExtract( char *strBuffer)
{
    char CharnIn;
    int intNumToSkip;
    int intIndex = 0;

    /* The Time Stamp starts in the sixth field of the header and is the number of GPS seconds that have elapsed since the start of the week. The Horizontal Speed starts in the 14th field of the POSA string. SkipDataField() skips the first five fields of the string so the sixth one can be read in. */

    intNumToSkip = 5;
    SkipDataField( strBuffer, intNumToSkip);

    /* Extracting the Time Stamp from the VEL string. */
    while( strBuffer[intMark] != ' ,')
    {
        strVEL_TimeStamp[intIndex] = strBuffer[intMark];
        intIndex++;
        intMark++;
    }

    intMark++; /* Move to beginning of next field. */
    intNumToSkip = 6;
    SkipDataField( strBuffer, intNumToSkip);

    strAltitude[intIndex] = strBuffer[intMark];
    intIndex++;
    intMark++;

    }
/* Extracting horizontal speed from VELA string. */
intIndex = 0;
while( strBuffer[intMark] != ',')
{
    strHorizontalSpeed[intIndex] = strBuffer[intMark];
    intIndex++;
    intMark++;
}

/* Extracting ground track from VELA string. */
intMark++; /* Move to beginning of next field. */
intIndex = 0;
while( strBuffer[intMark] != ',')
{
    strGroundTrack[intIndex] = strBuffer[intMark];
    intIndex++;
    intMark++;
}

/* Extracting vertical speed from VELA string. */
intMark++; /* Move to beginning of next field. */
intIndex = 0;
while( strBuffer[intMark] != ',')
{
    strVerticalSpeed[intIndex] = strBuffer[intMark];
    intIndex++;
    intMark++;
}

/*
* (6)
* SkipDataField()
* Moves FilePointer past the specified number of data fields. This
* is accomplished by counting the commas between fields. A comma is
* used in the GPS string to delineate between data fields.
*/
void SkipDataField(char *Buffer, int intSkip)
{
    int intFieldCount = 0;
    int intMaxCount = 0;
    int CharIn;

    /* Number of fields passed in but actually skipping commas. */
    intMaxCount = intSkip + 1;

    while( intFieldCount < intMaxCount )
intMark++;

if (Buffer[intMark] == ',') || (Buffer[intMark] == ';')
{
    intFieldCount++;
}

/* Increment intMark one more time so index is at the start of the
  next data field and not on the ',' or ';'. */

intMark++;
*/
/* (8)
* CalculateVelocityVector()
* The GPS receiver provides the ground track and the vertical and
* horizontal air speeds. The horizontal air speed is actually the
* sum of the Velocity North and Velocity East vectors. The Velocity
* North and Velocity East vectors are found by decomposing the
* Horizontal air speed by taking the cosine of the ground track
* angle and multiplying it by the Horizontal air speed to yeild the
* Velocity North component of the velocity vector. The Velocity East
* component of the Velocity Vector is found by taking the sine of the
* ground track and multiplying it by the Horizontal air speed. The k
* component of the velocity vector is found by multiplying the Vertical
* air speed by negative one to insure that the Velocity Vector follows
* the right hand rule, i into j yeilds k, which points down towards the
* earth. There for the given vertical air speed must be negated.
*/

void CalculateVelocityVector(void)
{
    int i = 0;
    int j = 1;
    int k = 2;
    double dblHorizontalVelocityMag = 0;
    double dblVN squared = 0;
    double dblVESquared = 0;
    double dblVDsquared = 0;
    double DegToRad = 0;
    double RadToDeg = 0;

    DegToRad = PI/180.0;
    RadToDeg = 180.0 / PI;
dblVelocityVector[i] = dblHorizontalSpeed * cos (dblGroundTrack * DegToRad);
dblVelocityVector[j] = dblHorizontalSpeed * sin (dblGroundTrack * DegToRad);
dblVelocityVector[k] = dblVerticalSpeed * -1.0;

dblVNsquare = dblVelocityVector[i] * dblVelocityVector[i];
dblVEsquare = dblVelocityVector[j] * dblVelocityVector[j];
dblVDsquare = dblVelocityVector[k] * dblVelocityVector[k];

dblVelocityMagnitude = sqrt (dblVNsquare + dblVEsquare + dblVDsquare);

/* Calculate the Unit Velocity Vector. */

dblUnitVelocityVector[i] = (dblVelocityVector[i] / dblVelocityMagnitude);
dblUnitVelocityVector[j] = (dblVelocityVector[j] / dblVelocityMagnitude);
dblUnitVelocityVector[k] = (dblVelocityVector[k] / dblVelocityMagnitude);

void CalculateFlightPathAngle(void)
{
    int i = 0;
    int j = 1;
    int k = 2;
    double dblHorizontalVelocityMag = 0;
    double dblVNsquared = 0;
    double dblVEsquared = 0;
    double dblVelocityUp = 0;
    double DegToRad = 0;
    double RadToDeg = 0;
    double dblTempFPA = 0;

    /* (9)
    * CalculateFlightPathAngle()
    * The flight path angle is calculated by taking first assigning the 
    * Velocity Vector to more explicit names. Once the Velocity North, 
    * Velocity East and the Velocity Down have values the calculations 
    * are performed. The Velocity North and Velocity East terms are 
    * squared. Then the magnitude of the sum of the two vectors is found 
    * by taking the square root of the sum of the squares. This gives the 
    * two sides of the right triangle needed to calculate the flight path 
    * angle. The inverse tangent of the two angles is taken to find the 
    * flight path angle. The Velocity Down from the Velocity Vector is a 
    * positive number when pointing towards the earth. We need to switch this 
    * sign convention so it follows an ENU coordinate system. The Vn value is 
    * multiplied by negative one to insure the flight path angle indicates a 
    * climb or descent. The atan2() function is used in order to keep track 
    * of the correct quadrant. The value calculated is in radians. This is 
    * converted to degrees and assigned as the actual Flight Path Angle. 
    */
}
DegToRad = PI/180.0;
RadToDeg = 180.0 / PI;

dblVelocityNorth = dblVelocityVector[i];
dblVelocityEast = dblVelocityVector[j];
dblVelocityDown = dblVelocityVector[k];

dblVelocityUp = dblVelocityDown * -1.0;

dblVN squared = dblVelocityNorth * dblVelocityNorth;
dblVE squared = dblVelocityEast * dblVelocityEast;
dblHorizontalVelocityMag = sqrt( dblVN squared + dblVE squared );

dblFlightPathAngleRad = atan2(dblVelocityUp, dblHorizontalVelocityMag);
dbTempFPA = dblFlightPathAngleRad * RadToDeg;

/* The Flight Path Angle is now placed into a 3-point moving
* average to act as a low pass filter.
*/

dblFPA FILTER[2] = dblFPA FILTER[1];
dblFPA FILTER[1] = dblFPA FILTER[0];
dblFPA FILTER[0] = dbTempFPA;

dblFlightPathAngleDeg = ( dblFPA FILTER[0] +
                     dblFPA FILTER[1] +
                     dblFPA FILTER[2]) / 3;

} /*
 * (10)
 * CalculateAcceleration()
 * Calculates the acceleration and its normal and tangential components.
 */
void CalculateAcceleration(void)
{
    int i = 0;
    int j = 1;
    int k = 2;
    double dblNewAcceleration[3] = {0};
    double dblAN squared = 0;
    double dblAE squared = 0;
    double dblAD squared = 0;
    double dt = dblTimeRateOfChange;
    double dblAg dot_Vg = 0;

    dblNewAcceleration[i] = ( dblVelocityVector[i] - dblPreviousVelocityVector[i] ) / dt;
    dblNewAcceleration[k] = ( dblVelocityVector[k] - dblPreviousVelocityVector[k] ) / dt;
/* 3-point moving average to filter the acceleration. */

dblAcceleration_FILTER[2][i] = dblAcceleration_FILTER[1][i];
dblAcceleration_FILTER[2][j] = dblAcceleration_FILTER[1][j];
dblAcceleration_FILTER[2][k] = dblAcceleration_FILTER[1][k];

dblAcceleration_FILTER[1][i] = dblAcceleration_FILTER[0][i];
dblAcceleration_FILTER[1][j] = dblAcceleration_FILTER[0][j];
dblAcceleration_FILTER[1][k] = dblAcceleration_FILTER[0][k];

dblAcceleration_FILTER[0][i] = dblNewAcceleration[i];
dblAcceleration_FILTER[0][j] = dblNewAcceleration[j];
dblAcceleration_FILTER[0][k] = dblNewAcceleration[k];

dblAcceleration[i] = ( dblAcceleration_FILTER[0][i] +
                  dblAcceleration_FILTER[1][i] +
                  dblAcceleration_FILTER[2][i] ) / 3;

dblAcceleration[j] = ( dblAcceleration_FILTER[0][j] +
                  dblAcceleration_FILTER[1][j] +
                  dblAcceleration_FILTER[2][j] ) / 3;

dblAcceleration[k] = ( dblAcceleration_FILTER[0][k] +
                  dblAcceleration_FILTER[1][k] +
                  dblAcceleration_FILTER[2][k] ) / 3;

/* End of 3-point moving average. */

dblANsquared = dblAcceleration[i] * dblAcceleration[i];
dblAEsquared = dblAcceleration[j] * dblAcceleration[j];
dblADsquared = dblAcceleration[k] * dblAcceleration[k];

dblAccelerationMagnitude = sqrt( dblANsquared + dblAEsquared + dblADsquared );

/* Calculate the dot product of the Acceleration and Velocity. */

dblAg_dot_Vg = ( dblAcceleration[i] * dblVelocityVector[i] ) +
                     ( dblAcceleration[j] * dblVelocityVector[j] ) +
                     ( dblAcceleration[k] * dblVelocityVector[k] );

/* Calculate the Tangential Acceleration. */

dblTangentialAcceleration[i] = dblUnitVelocityVector[i] *
                              ( dblAg_dot_Vg / dblVelocityMagnitude );
dblTangentialAcceleration[j] = dblUnitVelocityVector[j] *
                              ( dblAg_dot_Vg / dblVelocityMagnitude );
dblTangentialAcceleration[k] = dblUnitVelocityVector[k] *
                              ( dblAg_dot_Vg / dblVelocityMagnitude );
Calculate the Normal Acceleration. */

dblNormalAcceleration[i] = dblAcceleration[i] - dblTangentialAcceleration[i];
dblNormalAcceleration[j] = dblAcceleration[j] - dblTangentialAcceleration[j];
dblNormalAcceleration[k] = dblAcceleration[k] - dblTangentialAcceleration[k];

dblPreviousVelocityVector[i] = dblVelocityVector[i];
dblPreviousVelocityVector[j] = dblVelocityVector[j];
dblPreviousVelocityVector[k] = dblVelocityVector[k];

} /*
 * (11)
 * CalculateGravity()
 * Calculates the normal and tangential compoents of gravity. For this
 * application of the gravity vector, the k component will be in the
 * positive Z direction, or the vertical down direction. The convention
 * for the Gravity Vector will be \( G = (0, 0, 9.81) \) and not the usual
 * \( G = (0, 0, -9.81) \).
 * */
void CalculateGravity(void)
{
    int i = 0;
    int j = 1;
    int k = 2;
    double dblG_dot_Vg = 0;

    /* Calculate the dot product of Gravity and the VelocityVector. */
    dblG_dot_Vg = (dblGravity[i] * dblVelocityVector[i]) +
    (dblGravity[j] * dblVelocityVector[j]) +
    (dblGravity[k] * dblVelocityVector[k]);

    /* Calculate the Tangential Gravitational Acceleration. */
    dblTangentialGravity[i] = dblUnitVelocityVector[i] *
        (dblG_dot_Vg /
        dblVelocityMagnitude);
    dblTangentialGravity[j] = dblUnitVelocityVector[j] *
        (dblG_dot_Vg /
        dblVelocityMagnitude);
    dblTangentialGravity[k] = dblUnitVelocityVector[k] *
        (dblG_dot_Vg /
        dblVelocityMagnitude);

    /* Calculate the Normal Gravitational Acceleration. */
    dblNormalGravity[i] = dblGravity[i] - dblTangentialGravity[i];
    dblNormalGravity[k] = dblGravity[k] - dblTangentialGravity[k];
CalculateLiftVector()

* Takes the Vector Difference of the normal component of gravity and
* the normal component of acceleration. This simple difference is the
* lift vector.
*/
void CalculateLiftVector(void)
{
    int i = 0;
    int j = 1;
    int k = 2;

    dblLiftVector[i] = dblNormalAcceleration[i] - dblNormalGravity[i];
    dblLiftVector[j] = dblNormalAcceleration[j] - dblNormalGravity[j];
    dblLiftVector[k] = dblNormalAcceleration[k] - dblNormalGravity[k];
}

CalculateReferenceVector()

* Gives the cross product of the normal component of gravity and the
* velocity vector. This produces a reference plane which is orthogonal
* to the velocity vector and parallel to the ground. The lift vector is
* compared to this to determine the pseudo-roll.
*/
void CalculateReferenceVector(void)
{
    int i = 0;
    int j = 1;
    int k = 2;

    dblReferenceVector[i] = dblNormalGravity[j]*dblVelocityVector[k] -
                          dblNormalGravity[k]*dblVelocityVector[j];

    dblReferenceVector[j] = - (dblNormalGravity[i]*dblVelocityVector[k] -
                              dblNormalGravity[k]*dblVelocityVector[i]);

    dblReferenceVector[k] = dblNormalGravity[i]*dblVelocityVector[j] -
                          dblNormalGravity[j]*dblVelocityVector[i];
}

CalculatePseudoRoll()
void CalculatePseudoRoll(void)
{
  int i = 0;
  int j = 1;
  int k = 2;
  double dblLmag = 0;
  double dblPmag = 0;
  double dblL_dot_P = 0;
  double dblLiSquare = 0;
  double dblLjSquare = 0;
  double dblLkSquare = 0;
  double dblPiSquare = 0;
  double dblPjSquare = 0;
  double dblPkSquare = 0;
  double dblTemp = 0;
  double dblTempPseudoRollDeg = 0;
  double dblBoundedPseudoRollDeg = 0;

  /* Calculate the dot product of L (LiftVector) and P (ReferenceVector). */
  dblL_dot_P = dblLiftVector[i] * dblReferenceVector[i] +
               dblLiftVector[j] * dblReferenceVector[j] +
               dblLiftVector[k] * dblReferenceVector[k];

  /* Calculate the Magnitude of the Lift Vector (L). */
  dblLmag = sqrt( dblLiftVector[i] * dblLiftVector[i] +
                  dblLiftVector[j] * dblLiftVector[j] +
                  dblLiftVector[k] * dblLiftVector[k] );

  /* Calculate the Magnitude of the Reference Vector (P). */
  dblPmag = sqrt( dblReferenceVector[i] * dblReferenceVector[i] +
                  dblReferenceVector[j] * dblReferenceVector[j] +
                  dblReferenceVector[k] * dblReferenceVector[k] );

  dblTemp = dblL_dot_P / ( dblLmag * dblPmag );
  dblPseudoRollRad = asin( dblTemp );
  dblTempPseudoRollDeg = dblPseudoRollRad * ( 180.0 / PI );

  /* Set bounds for PSR to mitigate noise. */
  if( dblTempPseudoRollDeg > 55 )
  {
    dblBoundedPseudoRollDeg = 55;
  }
  else if( dblTempPseudoRollDeg < -55 )
  {
    dblBoundedPseudoRollDeg = -55;
  }
else
{
    dblBoundedPseudoRollDeg = dblTempPseudoRollDeg;
}

/* This Bounded Pseudo-Roll is now placed into a 3-point moving
* average to act as a low pass filter.
*/
dblPSR_FILTER[2] = dblPSR_FILTER[1];
dblPSR_FILTER[1] = dblPSR_FILTER[0];
dblPSR_FILTER[0] = dblBoundedPseudoRollDeg;

dblPseudoRollDeg = ( dblPSR_FILTER[0] +
                    dblPSR_FILTER[1] +
                    dblPSR_FILTER[2] ) / 3;

*/

/*
* (15)
* DisplayPositionData()
* Displays the current floating point values from the POSA string.
*/
void DisplayPositionData(void)
{
    printf("POS: ");
    printf("%8.2f ", dblPOS_TimeStamp);
    printf("%6.8f ", dblLatitude);
    printf("%6.8f ", dblLongitude);
    printf("%6.8fn", dblAltitude);
}

/*
* (16)
* DisplayVelocityData()
* Displays the current floating point values from the SPHA string.
*/
void DisplayVelocityData(void)
{
    printf("VEL: ");
    printf("%8.2f ", dblVEL_TimeStamp);
    printf("%f ", dblHorizontalSpeed);
    printf("%f ", dblGroundTrack);
    printf("%fn", dblVerticalSpeed);
/*
 * (17)
 * PrintResults
 * Used to simulate the psr calculation at the point at which it would
 * take place. Instead of a calculation being performed, the POS and VEL
 * data is displayed.
 */
void PrintResults(void)
{
    DisplayPositionData();
    DisplayVelocityData();
    printf("Flight Path Angle : \%fn", dblFlightPathAngleDeg);
}

/* Commented out FlightDataToFile 30 April 2003, DPB
 *
 * FlightDataToFile()
 * Outputs the time stamp, latitude, longitude, altitude, FPA and PSR to
 * a file for use with the Bench Test Display written in Visual Basic.
 */
void FlightDataToFile( FILE * FilePointer )
{
    fprintf( FilePointer, "%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%flt%fl

/*
 * (18)
 * GPSDataToFile()
 * Sends the current GPS String contained in the sorted buffer to file.
 */
void GPSDataToFile( FILE * FilePointer, char * strBuffer, int intSize )
{
```c
int intIndex = 0;

for( intIndex = 0 ; intIndex < intSize ; intIndex++ )
{
    fprintf( FilePointer, "%c", strBuffer[intIndex] );
}

fprintf( FilePointer, \n";)

/*
 * ClearPOSStrings() 
 * The strings which hold the timestamp, latitude, longitude 
 * and altitude must be cleared between file sampling. 
 * This is done by placing '0' in each element of the character 
 * string used to represent the lat, long and alt. 
 */
void ClearPOSStrings(void)
{
    strcpy( strPOS_TimeStamp, strInitialize );
    strcpy( strLatitude, strInitialize );
    strcpy( strLongitude, strInitialize );
    strcpy( strAltitude, strInitialize );
}

/*
 * ClearVELStrings() 
 * The strings which hold the timestamp, horizontal speed, vertical 
 * speed and ground track must be cleared between file sampling. 
 * This is done by placing a '0' in each element of the character 
 * string used to represent the Vh,Vv and Gt. 
 */
void ClearVELStrings(void)
{
    strcpy( strVEL_TimeStamp, strInitialize );
    strcpy( strHorizontalSpeed, strInitialize );
    strcpy( strVerticalSpeed, strInitialize );
    strcpy( strGroundTrack, strInitialize );
}

/*
 * (23)
 * ProcessGPSdata( char *, int );
 * 
 * The variables used to build the state vector are:
 * 1. strPOS_TimeStamp 
 * 2. dblEast 
 * 3. dblNorth 
 * 4. dblUp 
 * 5. dblVelocityMagnitude 
 */
void ProcessGPSData( char *strSortedBuffer, int intBufferLength) 
{

double dblLLAREF[3];

dblLLAREF[0] = 39.213774;
dblLLAREF[1] = -82.224948;
dblLLAREF[2] = 231;

/***** PROCESS GPS STRING ACCORDING TO TYPE *****/

intMark = 0; /* Cleared each time before parsing string */

while( ( intMark < intBufferLength ) )
{

    intMark++;

/***** BESTPOSA HEADER AQUISITION *****/

if( POS_StringID( strSortedBuffer ) )
{

    /* Strings must be cleared of old data */
    ClearPOSStrings();

    POS_DataExtract( strSortedBuffer );

    if( TimeStampCheck() )
    {
        StringDataToFloat();
        CalculateVelocityVector();

        // Convert LLA to XYZ and then to ENU.
        dblLLA[0] = dblLatitude;
        dblLLA[1] = dblLongitude;
        dblLLA[2] = dblAltitude;
        WGSlla2xyz( dblLLA, dblXYZ );
        WGSxyz2enu( dblLLAREF, dblXYZ, dblENU );
        dblEast = dblENU[0];
        dblNorth = dblENU[1];
        dblUp = dblENU[2];

    }
}

/***** BESTVELA HEADER AQUISITION *****/

if( VEL_StringID( strSortedBuffer ) )
{
{ /* Strings must be cleared of old data */
    ClearVELStrings();
    VEL_DataExtract( strSortedBuffer );
    if( TimeStampCheck() )
    {
        StringDataToFloat();
        CalculateVelocityVector();
        // Convert LLA to XYZ and then to ENU.
        dblLLA[0] = dblLatitude;
        dblLLA[1] = dblLongitude;
        dblLLA[2] = dblAltitude;
        WGSlla2xyz( dblLLA, dblXYZ );
        WGSxyz2enu( dblLLAREF, dblXYZ, dblENU );
        dblEast = dblENU[0];
        dblNorth = dblENU[1];
        dblUp = dblENU[2];
    }
} /* End while loop for processing POS or VEL string */
A.5 Data Collection Variables

#include <math.h>

/******** Global Variable Declarations ********/

/* Used to Manipulate Sorted Buffer. */
int intMark = 0; /* Marks the index position in sorted buffer. */

/* (GLOBAL VAR) State Vector information taken from GPS strings. */
char strPOS_Timestamp[15] = "000000000000000";
char strLatitude[15] = "0000000000000000";
char strLongitude[15] = "00000000000000000";
char strAltitude[15] = "00000000000000000";
char strVEL_Timestamp[15] = "0000000000000000000";
char strHorizontalSpeed[15] = "0000000000000000000";
char strVerticalSpeed[15] = "0000000000000000000";
char strGroundTrack[15] = "0000000000000000000";
char strInitialize[15] = "0000000000000000000";

double PI = M_PI;
double dblTimeRateOfChange = 0.05; /* Depends on RCVR rate. */

/* (GLOBAL VAR) Converted State Vector information to floating point. */
double dblPOS_Timestamp = 0;
double dblLatitude = 0;
double dblLongitude = 0;
double dblAltitude = 0;
double dblILLA[3] = {0};
double dblXYZ[3] = {0};
double dblENU[3] = {0};
double dblEast = 0;
double dblNorth = 0;
double dblUp = 0;

double dblVEL_Timestamp = 0;
double dblHorizontalSpeed = 0;
double dblVerticalSpeed = 0;
double dblGroundTrack = 0;

/* Holds the i,j,k components of the velocity vector. */
double dblVelocityVector[3] = {0};
double dblPreviousVelocityVector[3] = {0};
double dblVelocityNorth = 0;
double dblVelocityEast = 0;
double dblVelocityDown = 0;
double dblVelocityMagnitude = 0;
double dblUnitVelocityVector[3] = {0};
I* Holds the i,j,k components of the Acceleration vector. */
double dblAcceleration[3] = {0};
double dblNormalAcceleration[3] = {0};
double dblTangentialAcceleration[3] = {0};
double dblAccelerationMagnitude = 0;

/* This Variable is used as a filter for the Acceleration. */
double dblAcceleration_FILTER[3][3] = {0};

/* Holds the i,j,k components of the gravitational acceleration vector. */
double dblGravity[3] = {0.0,0.0,9.80665};
double dblNormalGravity[3] = {0};
double dblTangentialGravity[3] = {0};

/* Holds the Flight Path Angle Gamma. */
double dblFlightPathAngleDeg = 0;
double dblFlightPathAngleRad = 0;

/* Used as a filter for the Flight Path Angle. */
double dblFPA_FILTER[3] = {0};

/* Holds the Lift Vector. */
double dblLiftVector[3] = {0};

/* Holds the Reference vector. */
double dblReferenceVector[3] = {0};

/* Holds the calculated Pseudo-Roll and Filter for the PSR. */
double dblPseudoRollDeg = 0;
double dblPseudoRollRad = 0;
double dblPSR_FILTER[3] = {0};

/* Holds the Roll, Pitch, and Yaw from the AHRS unit. */
double dblAHRS_Roll = 0;
double dblAHRS_Pitch = 0;
double dblAHRS_Yaw = 0;
double dblAHRS_RollRaw = 0;
double dblAHRS_PitchRaw = 0;
double dblAHRS_YawRaw = 0;

/* Holds the State Vector Strings */
char strStateVector[512] = "";
char strStateTimeStamp[15] = "";
char strStateEast[15] = "";
char strStateNorth[15] = "";
char strStateUp[15] = "";
char strStateVelocity[15] = "";
char strStateGroundTrack[15] = "";
char strStatePitch[15] = "";
char strStateRoll[15] = "";

/* Used for sending state vector to SVS */
int intSizeWrite = 0; /* Return value for device write. */
int intSizeWrite4 = 0;
char outputbuffer[250] = {0};
char Header[5] = "#FD."
char Tail[3] = ",*";
int Xcount = 0;
int Kcount = 0;

/************************* GLOBAL VARIABLE DECLARATIONS *************************/
#include <math.h>

/* Used to Manipulate Sorted Buffer. */
int intMark; /* Marks the index position in sorted buffer. */

/* (GLOBAL VAR) State Vector information taken from GPS strings. */
extern char strPOS_TimeStamp[15];
extern char strLatitude[15];
extern char strLongitude[15];
extern char strAltitude[15];
extern char strVEL_TimeStamp[15];
extern char strHorizontalSpeed[15];
extern char strVerticalSpeed[15];
extern char strGroundTrack[15];
extern char strInitialize[15];

extern double PI;
extern double dblTimeRateOfChange; /* Depends on RCVR rate. */

/* (GLOBAL VAR) Converted State Vector information to floating point. */
extern double dblPOS_TimeStamp;
extern double dblLatitude;
extern double dblLongitude;
extern double dblAltitude;
extern double dblLLA[3];
extern double dblXYZ[3];
extern double dblENU[3];
extern double dblEast;
extern double dblNorth;
extern double dblUp;
extern double dblVEL_TimeStamp;
extern double dblHorizontalSpeed;
extern double dblVerticalSpeed;
extern double dblGroundTrack;

/* Holds the i,j,k components of the velocity vector. */
extern double dblVelocityVector[3];
extern double dblPreviousVelocityVector[3];
extern double dblVelocityNorth;
extern double dblVelocityEast;
extern double dblVelocityDown;
extern double dblVelocityMagnitude;
extern double dblUnitVelocityVector[3];

/** Holds the i,j,k components of the Acceleration vector. */
extern double dblAcceleration[3];
extern double dblNormalAcceleration[3];
extern double dblTangentialAcceleration[3];
extern double dblAccelerationMagnitude;

/** This Variable is used as a filter for the Acceleration. */
extern double dblAcceleration_FILTER[3][3];

/** Holds the i,j,k components of the gravitational acceleration vector. */
extern double dblGravity[3];
extern double dblNormalGravity[3];
extern double dblTangentialGravity[3];

/** Holds the Flight Path Angle Gamma. */
extern double dblFlightPathAngleDeg;
extern double dblFlightPathAngleRad;

/** Used as a filter for the Flight Path Angle. */
extern double dblFPA_FILTER[3];

/** Holds the Lift Vector. */
extern double dblLiftVector[3];

/** Holds the Reference vector. */
extern double dblReferenceVector[3];

/** Holds the calculated Pseudo-Roll and Filter for the PSR. */
extern double dblPseudoRollDeg;
extern double dblPseudoRollRad;
extern double dblPSR_FILTER[3];

/** Holds the Roll, Pitch, and Yaw from the AHRS unit. */
extern double dblAHRS_Roll;
extern double dblAHRS_Pitch;
extern double dblAHRS_Yaw;
extern double dblAHRS_RollRaw;
extern double dblAHRS_PitchRaw;
extern double dblAHRS_YawRaw;

/** Holds the State Vector Strings */
extern char strStateVector[512];
extern char strStateTimeStamp[15];
extern char strStateEast[15];
extern char strStateNorth[15];
extern char strStateUp[15];
extern char strStateVelocity[15];
extern char strStateGroundTrack[15];
extern char strStatePitch[15];
extern char strStateRoll[15];

/* Used for sending state vector to SVS */
int intSizeWrite; /* Return value for device write. */
int intSizeWrite4;
char outputbuffer[250];
char Header[5];
char Tail[3];
int Xcount;
int Kcount;
Appendix B  Synthetic Vision Code

The code presented in this section was written using the Visual Basic programming language in conjunction with the Revolution-3D® graphics engine. This code should be viewed from the Microsoft development suite. A programmer familiar with the Visual Basic programming language should be able to follow the comments within this section to make necessary changes to the code. All variables are descriptive and all function names provide the functional nature of the code that lies within. Treat the main body of this code (Figure B.1) as the main body of a C-program. Use a top down design flow and follow through the main and into the necessary function calls.

```
DECLARE Graphics Objects

Main Loop
while(FLAG = TRUE){
    if(Attitude Calculated){
        Set Camera Attitude
        Set Camera Position
    }
    Clear Screen of Previous Data
    Render the Current Scene
    Render Appropriate Text
}end while
```

Figure B. 1 Pseudo-code for the Main SVS Routine
B.1 Main Form

Author: Douglas Burch
Date: 15 Jan 2002
Purpose:
Initializes COMMI port (RS-232) to 9600 bps then sets the NovAtel OEM-4 GPS receiver's baud rate to 115200 bps.

The main function of this program monitors for a comm. I interrupt. When an interrupt is received, the state-vector information is parsed and the parameters are used to update the state-vector internal to the synthetic vision software. Once the internal state-vector is updated the camera used to simulate the forward looking aircraft body view is set up and the correct image is rendered making use of the Revolution-3D graphics engine. The main loop runs continuously rendering the same camera view until an interrupt is received and a new state-vector has been confirmed.

There are a total of eight flight parameters that are needed by this program:
- Time Stamp
- East (meters)
- North (meters)
- Height (meters)
- Ground Speed (meters/sec)
- Ground Track (degrees)
- Pitch (degrees)
- Roll (degrees)

Option Explicit 'Avoid Undifined Variables.

API Call Declarations.

Private Declare Sub Sleep Lib "kernel32" (ByVal dwMilliseconds As Long)

declare references to Revolution3D
Public Engine As New R3D_Engine
Public Pipeline As New R3D_Pipeline
Public Control As New R3D_Control
Public Interface2D As New R3D_Interface2D
Public MeshBuilder As New R3D_MeshBuilder
Public MaterialLib As New R3D_MaterialLib
Public TextureLib As New R3D_TextureLib
Public PowerMonitor As New R3D_PowerMonitor
Public Tools As New R3D_Tools
Public Camera As New R3D_Camera
Public SkyDome As New R3D_SkyDome
Public PolyVox2 As New R3D_PolyVox2
Public Helper As New R3D_Helper
Public BSPSystem As New R3D_BSPSystem

Private Sub cmd_Exit_Click()
    Close (intFile)
    Unload Me
End Sub

'*** Start Main Loop ***
Private Sub Form_Load()

'*** Open the Flight Data File ***
    intFile = FreeFile
    Open "StateVectorData.txt" For Output As #intFile
    strFileHeader = "Header Time(s) East(m) North(m) Up(m) GrndSpd(m/s) GrndTrck(deg) Pitch(deg) Roll(deg)"
    Print #intFile, strFileHeader
    Call SetUpGameEngine
    Call SetUpPipeLine
    Call CreateSky
    Call CreateTerrain
    Call CreateRunway

'*** SET THE STARTING POSITION FOR THE CAMERA ***
    Call SetPosition

'*** Create the PAPI and set it beside the runway ***
    Call CreatePAPI_Light_1
    Call CreatePAPI_Light_2
    Call CreatePAPI_Light_3
    Call CreatePAPI_Light_4

'*** SET THE PIXEL TO METER RATIO ***
    sngIPixel_Per_Meter = 0.1

'*** MAIN LOOP ***

'// While condition set to true.
blnRUN = True

'// Timer
    sngIStartTime = Timer
    sngIDelayTime = dt '0.05 = 20Hz

'*** INITIALIZE THE GPS RECEIVER ***
    Call Initialize_NovAtel

    While (Not blnRUN = False)
'//receive the pressed keys
Control.Keyboard_ReceiveKeys

'//check if the user has pressed and exit if true
If (Control.Keyboard_GetKeyState(R3DKEY_END) = True) Then blnRUN = False

'***** SET POSITION, ALTITUDE, AND ATTITUDE *****
If (blnAttitudeCalculated) Then

    blnAttitudeCalculated = False

'***** Convert Doubles to type Single. *****
Call ConvertFlightDataToSingle

'***** SET CAMERA ATTITUDE. *****

'*** Pseudo-Roll ***
snglCurrentPseudoRoll = snglPseudoRoll - snglPreviousPseudoRoll
snglPreviousPseudoRoll = snglPseudoRoll
Camera.RotateZ (-snglCurrentPseudoRoll) 'Camera Roll is in different coordinate frame.

'*** Flight Path Angle ***
snglCurrentFlightPathAngle = snglFlightPathAngle - snglPreviousFlightPathAngle
snglPreviousFlightPathAngle = snglFlightPathAngle
Camera.RotateX (-snglCurrentFlightPathAngle) 'Camera Pitch is in different coordinate frame.

'***** Altitude and Position Setting. *****

'*** Calculate Altitude Correction of Camera. ***

'//Get the height (y) on the polyvox terrain at the position the camera currently is...
snglTerrainHeight = PolyVox2.Scape_GetAltitude(Helper.R3DPoint2D(snglPosition_Z, snglPosition_Z))
Camera.SetPosition snglAirCraft_X, snglAirCraft_Y, snglAirCraft_Z

'***** Set the Heading. *****
snglCurrentGroundTrack = snglGroundTrack - snglPreviousGroundTrack
snglPreviousGroundTrack = snglGroundTrack
Camera.RotateY (snglCurrentGroundTrack)

End If

DoEvents

'***** GET ATTITUDE AND ALTITUDE INFORMATION FROM THE CAMERA. *****
'// Get Roll, Pitch, and Heading.
Camera.GetRotation snglPitch, snglHeading, snglRoll

'// get the height (y) on the polyvox terrain at the position the camera currently is...
sng1TerrainHeight = PolyVox2.Scape_GetAltitude(Helper.R3DPoint2D(sng1Position_Z, sng1Position_Z))

'**** CALCULATE THE GLIDE SLOPE. ***/
dblGlideSlope = CalculateGlideSlope(sng1Position_X, sng1Position_Y, sng1Position_Z)

'**** PAPI CONTROL ****/
Call SetPAPI(dblGlideSlope)

***** SCREEN CLEAN UP *****

'// Clear the screen of previous data.
Pipeline.Renderer_Clear

'// Render the scene, Display the terrain.
Pipeline.Renderer_Render

'// Display Text and Camera Position.
Pipeline.Renderer_Display

Wend

'//terminate the engine and call vb's end
Engine.TerminateMe
End

End Sub

Public Function CalculateGlideSlope(X As Single, Y As Single, Z As Single) As Double
' X is the Local X
' Y is the Local Height
' Z is the Local Z

Dim PI As Double

PI = 4 * Atn(1)

Dim dblILS_X As Double ' ILS Local X Position
Dim dblILS_Y As Double ' ILS Local Y Position
Dim dblAircraft_X As Double ' Aircraft Local X Position
Dim dblAircraft_Y As Double ' Aircraft Local Altitude
Dim dblAircraft_Z As Double ' Aircraft Local Y Position
Dim dblHeight As Double
Dim dblDelta_X As Double
Dim dblDelta_Z As Double
Dim Range_to_ILS As Double
'*** THE GLIDE SLOPE IS CALCULATED BASED ON THE RANGE FROM THE AIRCRAFT TO ' THE CENTER OF THE RUNWAY, 1000 FEET FROM THE THRESHOLD. THIS IS ' PERFORMED IN THE LOCAL COORDINATE FRAME BASED ON PIXELS.

dblAircraft_X = X
dblHeight = Y
dblAircraft_Z = Z

dblILS_X = 503.669
dblILS_Z = 507.19

dblDelta_X = dblILS_X - dblAircraft_X
dblDelta_Z = dblILS_Z - dblAircraft_Z

Range_to_ILS = Sqr(dblDelta_X * dblDelta_X + dblDelta_Z * dblDelta_Z)
dblGlideSlope = Atn(dblHeight / Range_to_ILS) * (180 / PI)

CalculateGlideSlope = dblGlideSlope

End Function

Public Sub SetPAPI(dblGlideSlope As Double)

' // TRUE = Red
' // FALSE = White

'*** VERY HIGH (W)(W)(W)(W) ***
If (dblGlideSlope > 3.5) Then
    Call SetPAPI_1(False) 'WHITE
    Call SetPAPI_2(False) 'WHITE
    Call SetPAPI_3(False) 'WHITE
    Call SetPAPI_4(False) 'WHITE

'*** HIGH (W)(W)(R) ***
ElseIf (dblGlideSlope > 3.25 And dblGlideSlope <= 3.5) Then
    Call SetPAPI_1(False) 'WHITE
    Call SetPAPI_2(False) 'WHITE
    Call SetPAPI_3(True) 'RED
    Call SetPAPI_4(Truc) 'RED

'*** ON GLIDE SLOPE (W)(W)(R)(R) ***
ElseIf (dblGlideSlope >= 2.75 And dblGlideSlope <= 3.25) Then
    Call SetPAPI_1(False) 'WHITE
    Call SetPAPI_2(False) 'WHITE
    Call SetPAPI_3(True) 'RED
    Call SetPAPI_4(True) 'RED

'*** LOW (W)(R)(R)(R) ***
ElseIf (dblGlideSlope >= 2.5 And dblGlideSlope < 2.75) Then

    Call SetPAPI_1(False) 'WHITE
    Call SetPAPI_2(True) 'RED
    Call SetPAPI_3(True) 'RED
    Call SetPAPI_4(True) 'RED

    '*** VERY LOW (R)(R)(R)(R) ***
ElseIf (dblGlideSlope < 2.5) Then

    Call SetPAPI_1(True) 'RED
    Call SetPAPI_2(True) 'RED
    Call SetPAPI_3(True) 'RED
    Call SetPAPI_4(True) 'RED

End If

End Sub

Private Sub Form_Unload(Cancel As Integer)

    If (frmNovAtel.SerialComm1.PortOpen = True) Then
        frmNovAtel.SerialComm1.PortOpen = False
    End If

End Sub

Property Get AttitudeCalculated() As Boolean

    AttitudeCalculated = blnAttitudeCalculated

End Property

Property Let AttitudeCalculated(blnAttitudeCalculatedIn As Boolean)

    blnAttitudeCalculated = blnAttitudeCalculatedIn

End Property

Public Sub FlightDataToFile()

    Write #intFLTDATAHandler, dblPOS_TimeStamp, _
        dblEast, _
        dblNorth, _
        dblUp, _
        dblVelocityMagnitude, _
        dblGroundTrack, _
        dblFlightPathAngleDeg, _
        dblPseudoRollDeg

End Sub
Public Sub SetUpGameEngine()

'// The first thing that needs to be done is to set up the parameters
'// of the game engine. For the flight display we want a full screen
'// view. The field of view is set to 50 degrees and the view distance
'// is set to 350.

'// By setting the .InitializeMe property to True the start up menu is
'// ignored and the program goes directly to execution. Had the property
'// been set to False then a start up prompt would have appeared.

With Engine
    .Inf_SetFieldOfView 45
    .Inf_SetViewDistance 5000
    .Inf_SetNearClippingPlane 0
    .Inf_SetRenderTarget frmEHUD hWnd, R3DRENDERTARGET_FULLSCREEN
    .Inf_SetRenderTarget frmEHUD hWnd, R3DRENDERTARGET_WINDOW
    .Inf_SetProjectionType R3DPROJECTIONTYPE_PERSPECTIVE
    .Inf_ForceResolution 1024, 1024, 10
    If .InitializeMe(True) = -1 Then End With
End With

'Position the Window and set the size.
Me.Width = (Screen.Width - (Screen.Width / 4))
Me.Height = (Screen.Height - (Screen.Height / 3))

'Place window on screen.
Me.Left = 0
Me.Top = 0

End Sub

Public Sub SetUpPipelineO

'// The pipeline describes how the image will be rendered or executed.
With Pipeline
    .SetAmbientLight 0, 0, 0
    .SetBackColor 30, 30, 140
    .SetDithering False
    .SetSpecular False
    .SetFillMode R3DFILLMODE_SOLID
    .SetShadeMode R3DSHADEMODE_GOURAUD
    .SetTextureFilter R3DTEXTUREFILTER_LINEARFILTER
    .SetColorKeying True
    .SetFog 250, 150, 30.0.500, R3DFOGTYPE_LINEAR
End With

'// Now that we know how the display will be executed we have to create
'// the image to be rendered through loading texture files into strings.

End Sub
Public Sub CreateSky()

    '// Load the texture for the sky.
    TextureLib.Texture_Load "sky_panorama", "skyground_panorama.bmp"

    '//add the sky sphere with a radius of 2500.
    SkyDome.Sphere_Create "sky_panorama", "", 2500

End Sub

Public Sub CreateTerrain()

    '// The Terrain texture or "look" is created by loading in a file that contains grass,
    '// rocks or something of that nature. In this case a rocking terrain is loaded in. The
    '// path to the file is given and then it is bound to a string as a sort of nick name.
    TextureLib.Texture_Load "rockyground", "checkerdetail.bmp"

    '// The terrain is created using two files. One file "height.bmp" is a bitmap
    '// file which contains a grayscale bitmap. The lite color values are high
    '// elevation terrain features and the dark color values are the low elevation
    '// terrain features. The second file contains the detail that is laid ontop of
    '// the elevated terrain, "detail03.jpg".

    '// To generate the terrain we use the PolyVox2 object.
    PolyVox2.Scape_Create "terrain", "largeflatterrain2.bmp", 1500, True,
    POLYVOXDETAIL_AVERANGE
    PolyVox2.Scape_SetTexture 1, "rockyground", R3DTEXTURELAYERCOMMAND_COLORBLEND

    '// The texture scale sets the resolution on the loaded texture. The bitmap is 512x512 for
    '// both the texture and the terrain. The texture(rockyground) is compressed so that 50x50
    '// images exist in the 512x512 bitmap expressing the terrain features.
    PolyVox2.Scape_SetTextureScale 1, 50, 50

    '// The texture stages are added and given priority based on a reaction percentage.
    PolyVox2.Blender_AddTexture "new_2.bmp", 50
    PolyVox2.Blender_AddTexture "new_4.bmp", 20

    '// The dynamic textures are generated and then blended together. The _setblendpower() sets
    '// the percentage of blend between the two dynamic textures added above. This is compiled
    '// and stored as a temporary file in the TerrainData directory.
    PolyVox2.Blender_SetBlendPower 40
    PolyVox2.Blender_Compile "temp.bmp", R3DTEXTURERESOLUTION_32X32

End Sub

Public Sub CreateRunway()

    '// Load the texture for the runway.
    TextureLib.Texture_Load "runway", "runway.bmp"


// Load the numbers for the runway.
TextureLib.Texture_Load "runwaynumber25", "runwaynumber25.bmp"
TextureLib.Texture_Load "runwaynumber7", "runwaynumber7.bmp"

// Load the texture for the main taxi and minor taxi ways.
TextureLib.Texture_Load "maintaxi", "maintaxi.bmp"
TextureLib.Texture_Load "minortaxi", "minortaxi.bmp"

// Load the Material for the Hanger.
TextureLib.Texture_Load "hangersiding", "metalsiding.bmp"

// The runway was created using a flattened box. This gives the illusion of the runway
// having some depth.
MeshBuilder.Mesh_Create "runway"
"MeshBuilder.Mesh_AddBox "runway", "matsundown", 3, 0.01, 64.5, R3DBLENDMODE_NONE, R3DCULLMODE.DOUBLESIDED
'MeshBuilder.Mesh_SetPosition 456.562, 0.005, 479.993
MeshBuilder.Mesh_AddBox "runway", "matsundown", 3, 0.01, 85.36, R3DBLENDMODE_NONE, R3DCULLMODE.DOUBLESIDED
MeshBuilder.Mesh_SetPosition 475.052, 0.005, 490.668
Call MeshBuilder.Mesh_SetRotation(0, 240, 0)
MeshBuilder.Mesh_Finalize

// Add the Runway Numbers to the ends of the runway.
// 25
MeshBuilder.Mesh_Create "runwaynumber25"
MeshBuilder.Mesh_AddBox "runwaynumber25", "matsundown", 3, 0.007, 3,
R3DBLENDMODE_NONE, R3DCULLMODE.DOUBLESIDED
'MeshBuilder.Mesh_SetPosition 509.40192, 0.009, 510.5
MeshBuilder.Mesh_SetPosition 546.357, 0.009, 531.836
Call MeshBuilder.Mesh_SetRotation(0, 60, 0)
MeshBuilder.Mesh_Finalize

// 7
MeshBuilder.Mesh_Create "runwaynumber7"
MeshBuilder.Mesh_AddBox "runwaynumber7", "matsundown", 3, 0.007, 3,
R3DBLENDMODE_NONE, R3DCULLMODE.DOUBLESIDED
MeshBuilder.Mesh_SetPosition 403, 0.009, 449
Call MeshBuilder.Mesh_SetRotation(0, 240, 0)
MeshBuilder.Mesh_Finalize

// Add the Main Taxi way parallel to the runway.
'MeshBuilder.Mesh_Create "maintaxi"
'MeshBuilder.Mesh_AddBox "maintaxi", "matsundown", 1, 0.01, 61, R3DBLENDMODE_NONE, R3DCULLMODE.DOUBLESIDED
'MeshBuilder.Mesh_SetPosition 516.5, 0.005, 504.206
'Call MeshBuilder.Mesh_SetRotation(0, 240, 0)
'MeshBuilder.Mesh_Finalize

// Add the Minor Taxi ways which are perpendicular to the runway and connect the
// main taxi way to the runway.
'// Middle Taxi.
'MeshBuilder.Mesh_Create "minortaxi"
'MeshBuilder.Mesh_AddBox "minortaxi", "matsundown", 2, 0.01, 3.5, R3DBLENDMODE_NONE, R3DCULLMODE_DOUBLESIDED
'MeshBuilder.Mesh_SetPosition 514.5, 0.005, 507.67
'Call MeshBuilder.Mesh_SetRotation(0, 150, 0)
'MeshBuilder.Mesh_Finalize

'// 25 Minor Taxi
'MeshBuilder.Mesh_Create "minortaxi25"
'MeshBuilder.Mesh_AddBox "minortaxi", "matsundown", 3, 0.01, 3.5, R3DBLENDMODE_NONE, R3DCULLMODE_DOUBLESIDED
'MeshBuilder.Mesh_SetPosition 564.729, 0.005, 536.67
'Call MeshBuilder.Mesh_SetRotation(0, 150, 0)
'MeshBuilder.Mesh_Finalize

'// 7 Minor Taxi
'MeshBuilder.Mesh_Create "minortaxi7"
'MeshBuilder.Mesh_AddBox "minortaxi", "matsundown", 3, 0.01, 3.5, R3DBLENDMODE_NONE, R3DCULLMODE_DOUBLESIDED
'MeshBuilder.Mesh_SetPosition 464.271, 0.005, 478.67
'Call MeshBuilder.Mesh_SetRotation(0, 150, 0)
'MeshBuilder.Mesh_Finalize

'// Add the Hanger
'MeshBuilder.Mesh_Create "hangersiding"
'MeshBuilder.Mesh_AddBox "hangersiding", "matsundown", 6, 3, 3, R3DBLENDMODE_NONE, R3DCULLMODE_DOUBLESIDED
'MeshBuilder.Mesh_SetTextureScale 1, 2
'MeshBuilder.Mesh_SetPosition 550, 1.5, 500
'Call MeshBuilder.Mesh_SetRotation(0, 150, 0)
'MeshBuilder.Mesh_Finalize

End Sub

Public Sub SetPosition()

'// Set the start position for the camera. The camera is set to the center
'// of the scene on a heading of 0 degrees.
snglPosition_X = 567.556
snglPosition_Y = 0
snglPosition_Z = 534.556
snglHeading = 0
PI = 3.14159265358979

snglCurrentFlightPathAngle = 0
snglCurrentPseudoRoll = 0
snglPreviousFlightPathAngle = 0
snglPreviousPseudoRoll = 0
dt = 0.05 'Seems a little slow, dropping to 0.03 was 0.05.
snglPixel_Per_Meter = 0.1
snglPosition_Y = 12 * snglPixel_Per_Meter

' // Point nose of aircraft towards on a heading of zero degrees.
Camera.SetRotation 0, 0, 0

End Sub

Public Sub CreatePAPI_Light_1()

' // FURTHEST FROM RUNWAY.

' // Load the texture for the PAPIs.
TextureLib.Texture_Load "PAPIred", "PAPIred.bmp"
TextureLib.Texture_Load "PAPIwhite", "PAPIwhite.bmp"

' // Create the Light for the 1st or far left PAPI and default it to white.
MeshBuilder.Mesh_Create "PAPI_1"
MeshBuilder.Mesh_AddBox "PAPIred", "matsundown", 0.5, 0.5, 0.01, R3DBLENDMODE_ADD, R3DCULLMODE_DUBLESIDED
' MeshBuilder.Mesh_SetPosition 496, 1, 489
MeshBuilder.Mesh_SetPosition 513.639, 1, 499.187
Call MeshBuilder.Mesh_SetRotation(0, 240, 0)
MeshBuilder.Mesh_Finalize

End Sub

Public Sub CreatePAPI_Light_2()

' // Load the texture for the PAPIs.
TextureLib.Texture_Load "PAPIred", "PAPIred.bmp"
TextureLib.Texture_Load "PAPIwhite", "PAPIwhite.bmp"

' // Create the Light for the 2nd or middle left PAPI and default it to white.
MeshBuilder.Mesh_Create "PAPI_2"
MeshBuilder.Mesh_AddBox "PAPIred", "matsundown", 0.5, 0.5, 0.01, R3DBLENDMODE_ADD, R3DCULLMODE_DUBLESIDED
' MeshBuilder.Mesh_SetPosition 494, 1, 491
MeshBuilder.Mesh_SetPosition 511.639, 1, 501.187
Call MeshBuilder.Mesh_SetRotation(0, 240, 0)
MeshBuilder.Mesh_Finalize

End Sub

Public Sub CreatePAPI_Light_3()

' // Load the texture for the PAPIs.
TextureLib.Texture_Load "PAPIred", "PAPIred.bmp"
TextureLib.Texture_Load "PAPIwhite", "PAPIwhite.bmp"

' // Create the Light for the 3rd or middle right PAPI and default it to white.
MeshBuilder.Mesh_Create "PAPI_3"
MeshBuilder.Mesh_AddBox "PAPIred", "matsundown", 0.5, 0.5, 0.01, R3DBLENDMODE_ADD, R3DCULLMODE_DOUBLESIDED
'MeshBuilder.Mesh_SetPosition 492, 1, 493
MeshBuilder.Mesh_SetPosition 509.639, 1, 503.187
Call MeshBuilder.Mesh_SetRotation(0, 240, 0)
MeshBuilder.Mesh_Finalize

End Sub

Public Sub CreatePAPI_Light_4()

' //CLOSEST TO RUNWAY

' // Load the texture for the PAPIs.
TextureLib.Texture_Load "PAPIred", "PAPIred.bmp"
TextureLib.Texture_Load "PAPIwhite", "PAPIwhite.bmp"

' // Create the Light for the 4th or far right PAPI and default it to white.
MeshBuilder.Mesh_Create "PAPI_4"
MeshBuilder.Mesh_AddBox "PAPIred", "matsundown", 0.5, 0.5, 0.01, R3DBLENDMODE_ADD, R3DCULLMODE_DOUBLESIDED
'MeshBuilder.Mesh_SetPosition 490, 1, 495
MeshBuilder.Mesh_SetPosition 507.639, 1, 505.187
Call MeshBuilder.Mesh_SetRotation(0, 240, 0)
MeshBuilder.Mesh_Finalize

End Sub

Public Sub SetPAPI_1(blnState As Boolean)

' TRUE = RED
' FALSE = WHITE

With MeshBuilder

If (blnState) Then
  .Mesh_SetPointer "PAPI_1"
  .Mesh_SetTexture "PAPIred", 0, R3DTEXTURELAYERCOMMAND_BUMP
Else
  .Mesh_SetPointer "PAPI_1"
  .Mesh_SetTexture "PAPIwhite", 0, R3DTEXTURELAYERCOMMAND_BUMP
End If

End With

End Sub

Public Sub SetPAPI_2(blnState As Boolean)

' TRUE = RED
' FALSE = WHITE
With MeshBuilder

If (blnState) Then
    .Mesh_SetPointer "PAPI_2"
    .Mesh_SetTexture "PAPIred", 0, R3DTEXTURELAYERCOMMAND_BUMP
Else
    .Mesh_SetPointer "PAPI_2"
    .Mesh_SetTexture "PAPIwhite", 0, R3DTEXTURELAYERCOMMAND_BUMP
End If

End With

End Sub

Public Sub SetPAPI_3(blnState As Boolean)

    ' TRUE = RED
    ' FALSE = WHITE

    With MeshBuilder
        If (blnState) Then
            .Mesh_SetPointer "PAPI_3"
            .Mesh_SetTexture "PAPIred", 0, R3DTEXTURELAYERCOMMAND_BUMP
        Else
            .Mesh_SetPointer "PAPI_3"
            .Mesh_SetTexture "PAPIwhite", 0, R3DTEXTURELAYERCOMMAND_BUMP
        End If
    End With

End Sub

Public Sub SetPAPI_4(blnState As Boolean)

    ' TRUE = RED
    ' FALSE = WHITE

    With MeshBuilder
        If (blnState) Then
            .Mesh_SetPointer "PAPI_4"
            .Mesh_SetTexture "PAPIred", 0, R3DTEXTURELAYERCOMMAND_BUMP
        Else
            .Mesh_SetPointer "PAPI_4"
            .Mesh_SetTexture "PAPIwhite", 0, R3DTEXTURELAYERCOMMAND_BUMP
        End If
    End With

End Sub
Public Sub Development_DisplayParameters()

    '// Development Display Parameters.
    Interface2D.Primitive_SetDrawColor 0, 0, 0
    Interface2D.Primitive_DrawText 1, 1, "Terrain Height: " + CStr(sng1TerrainHeight * 10)
    Interface2D.Primitive_DrawText 1, 15, "Camera Altitude : " + CStr(sng1Position_Y * 1)
    Interface2D.Primitive_DrawText 1, 30, "flight Alt : " + CStr(sng1Up * 3.28)
    Interface2D.Primitive_DrawText 1, 45, "Camera Roll : " + CStr(sng1Roll)
    Interface2D.Primitive_DrawText 1, 60, "flight Roll : " + CStr(sng1PseudoRoll)
    Interface2D.Primitive_DrawText 1, 75, "Camera Pitch : " + CStr(sng1Pitch)
    Interface2D.Primitive_DrawText 1, 90, "flight Pitch : " + CStr(sng1FlightPathAngle)
    Interface2D.Primitive_DrawText 1, 105, "Camera Heading : " + CStr(sng1Heading)
    Interface2D.Primitive_DrawText 1, 120, "flight GroundTrack : " + CStr(sng1GroundTrack)
    Interface2D.Primitive_DrawText 1, 135, "Ground Speed : " + CStr(sng1Speed * 1.9438)

    Interface2D.Primitive_DrawText 700, 1, " Local X : " + CStr(sng1Position_X)
    Interface2D.Primitive_DrawText 700, 15, " Local Y : " + CStr(sng1Position_Z)

End Sub

Public Sub DisplayFramesPerSecond()

    '// Current fps (frames per second)
    Interface2D.Primitive_SetDrawColor 0, 0, 0
    Interface2D.Primitive_DrawText 380, 1, "fps:" + CStr(PowerMonitor.GetFrameTime)

End Sub

Public Sub ConvertFlightDataToSingle()

    '// New parameters for processing "HUB" packet data. Should not need those
    // above once this is working.
    varSpeed = dblVelocity
    sng1Speed = varSpeed

    varGroundTrack = dblGroundTrack
    sng1GroundTrack = varGroundTrack

    varFlightPathAngle = dblPitch
    sng1FlightPathAngle = varFlightPathAngle

    varPseudoRoll = dblRoll
    sng1PseudoRoll = varPseudoRoll

    '// Convert East in Meters to East in Pixels.
    varEast = dblEast
    sng1East = varEast
    sng1Aircraft_X = sng1East * sng1Pixel_Per_Meter + 512

    '// The convention is X cross Z = Y, so Z is North.
varNorth = dblNorth
snglNorth = varNorth
snglAircraft_Z = snglNorth * snglPixel_Per_Meter + 512

varUp = dblUp
snglUp = varUp
snglAircraft_Y = snglUp * snglPixel_Per_Meter + 0.8

End Sub
### B.2 Interrupt Service Routine Form

Public `strGPS_String` As String
Public `strPOS_String` As String
Public `strVEL_String` As String
Public `blnProcess_String_FLAG` As Boolean

Private Sub `SerialComm1_OnComm()`
    Dim `varBuffer` As Variant
    Dim `strCharIn` As String
    Dim `strString` As String
    Dim `strPOSHeader` As String
    Dim `strVELHeader` As String
    Dim `strProcess_String` As String
    Dim `intStringLength` As Integer
    Dim `intPOSLength` As Integer
    Dim `intVELLength` As Integer
    Dim `intStringCount` As Integer
    Dim `strCRCchar` As String
    Dim `intCRCcheck` As Integer
    Dim `intStart` As Integer
    Dim `intStop` As Integer

    `strCRCchar` = ",,"

    '*** Read Serial Buffer ***
    If (SerialComm1.CommEvent = `conEvReceive`) Then
        SerialComm1.RThreshold = 0
        `varBuffer` = SerialComm1.Input
        `strCharIn` = `varBuffer`
        DoEvents

        '*** Test for the start of the next string. ***
        'If found then copy string to processing string and continue collecting.
        If ("#" = `strCharIn`) Then
            `blnPoundFound` = True
            `strGPS_String` = `strGPS_String` & `strCharIn`
        End If
        If Not ("#" = `strCharIn`) Then
            `blnPoundFound` = False
            `strGPS_String` = ""
            'Clear GPS string for incoming string.
            `blnProcess_String_FLAG` = True 'Inform processing algorithm to start.
End If

SerialComm1.RThreshold = 1

End If

If (blnProcess_String_FLAG) Then

    intCRCcheck = 0
    intStart = 1
    intStop = Len(strProcess_String)
    While (intStart < intStop)
        intStart = InStr(intStart, strProcess_String, strCRCchar, 1)
        intStart = intStart + 1
        intCRCcheck = intCRCcheck + 1
    Wend

    If (intCRCcheck = 9) Then
        Call ProcessPacket(strProcess_String)
        Print #intFile, strProcess_String
        strProcess_String = ""
        blnProcess_String_FLAG = False
    Else
        strProcess_String = ""
        blnProcess_String_FLAG = False
    End If

End If

End Sub
B.3 Data Abstraction Functions

* Author: Douglas Burch  
* Date: 17 Jan 2002

Option Explicit

'-- The Kernel32 API contains a function that pauses application execution  
'-- for a given amount of time, specified in milliseconds.  
'-- To use the function, it must first be declared in the  
'-- General Declarations section of the module in which it will be used:  
Private Declare Sub Sleep Lib "kernel32" (ByVal dwMilliseconds As Long)

'----- THIS IS USED TO PROCESS DATA SENT FROM THE "HUB" COMPUTER. -----  
Public Sub ProcessPacket(ByVal strDataString As String)

'Extract the data variables using PacketDataExtract.  
Call PacketDataExtract(strDataString)

'Convert strings to floating point numbers.  
'Call StringDataToFloat

dblTimeStamp = Val(strTimeStamp)  
dblEast = Val(strEast)  
dblNorth = Val(strNorth)  
dblUp = Val(strUp)  
dblVelocity = Val(strVelocity)  
dblGroundTrack = Val(strGroundTrack)  
dblPitch = Val(strPitch)  
dblRoll = Val(strRoll)

'Tell main form that an attitude and position is available.  
frmEHUD.AttitudeCalculated = True

End Sub

Public Sub PacketDataExtract(ByVal strDataString As String)
Dim CommaPlace1 As Integer  
Dim CommaPlace2 As Integer  
Dim PlaceCount As Integer  
Dim intFieldCount As Integer  
Dim strSearchChar As String * 1

' The data packet is formatted as follows:  
' #FLTDATA,GPSTIME,EAST,NORTH,UP,VELOCITY,GROUNDTRACK,PICTH,ROLL*  
' The GPSTIME is in field number 2.

PlaceCount = 1  
intFieldCount = 1
strSearchChar = ",;"

While ((PlaceCount < Len(strDataString)) And (intFieldCount <= 8))

    CommaPlace1 = InStr(PlaceCount, strDataString, strSearchChar, vbTextCompare)
    CommaPlace2 = InStr(CommaPlace1 + 1, strDataString, strSearchChar, vbTextCompare)
    PlaceCount = CommaPlace1 + 1

Select Case intFieldCount

    Case 1
        strTimeStamp = Mid(strDataString, CommaPlace1 + 1, (CommaPlace2 - CommaPlace1))
        'Debug.Print "Time Stamp: " & strTimeStamp

    Case 2
        strEast = Mid(strDataString, CommaPlace1 + 1, (CommaPlace2 - CommaPlace1))
        'Debug.Print "East: " & strEast

    Case 3
        strNorth = Mid(strDataString, CommaPlace1 + 1, (CommaPlace2 - CommaPlace1))
        'Debug.Print "North: " & strNorth

    Case 4
        strUp = Mid(strDataString, CommaPlace1 + 1, (CommaPlace2 - CommaPlace1))
        'Debug.Print "Up: " & strUp

    Case 5
        strVelocity = Mid(strDataString, CommaPlace1 + 1, (CommaPlace2 - CommaPlace1))
        'Debug.Print "Velocity: " & strVelocity

    Case 6
        strGroundTrack = Mid(strDataString, CommaPlace1 + 1, (CommaPlace2 - CommaPlace1))
        'Debug.Print "GroundTrack: " & strGroundTrack

    Case 7
        strPitch = Mid(strDataString, CommaPlace1 + 1, (CommaPlace2 - CommaPlace1))
        'Debug.Print "Pitch: " & strPitch

    Case 8
        strRoll = Mid(strDataString, CommaPlace1 + 1, (CommaPlace2 - CommaPlace1))
        'Debug.Print "Roll: " & strRoll

End Select

intFieldCount = intFieldCount + 1

Wend

End Sub

Public Sub POS_FileDataExtract(ByVal strGPS_String As String)
Public Sub VEL_FileDataExtract(strGPS_String As String)
'*** EXTRACT TIME STAMP ***
strVEL_TimeStamp = DataExtract(strGPS_String, 6)
'*** EXTRACT HORIZONTAL SPEED ***
strHorizontalSpeed = DataExtract(strGPS_String, 13)
'*** EXTRACT GROUND TRACK ***
strGroundTrack = DataExtract(strGPS_String, 14)
End Sub

Public Function DataExtract(ByVal strStringIn As String, ByVal intFieldNumber As Integer) As String
 Dim intFieldCount As Integer
 Dim intFieldLength As Integer
 Dim intCharCount As Integer
 Dim strFieldMark As String * 1
 Dim strTestString As String * 1

tstrFieldMark = ","
intFieldCount = 0 "Must start at zero.
intCharCount = 1 "Must Start at One.

Debug.Print strStringIn & " Field Number : " & CStr(intFieldNumber)
'While ((intFieldCount < intFieldNumber) And (intCharCount < Len(strStringIn)))

'*** EXTRACT LATITUDE ***
strLatitude = DataExtract(strGPS_String, 12)
'*** EXTRACT LONGITUDE ***
strLongitude = DataExtract(strGPS_String, 13)
'*** EXTRACT Height ***
strHeight = DataExtract(strGPS_String, 14)
End Sub
Public Function ReturnFieldLength(ByVal strStringIn As String, ByVal intFieldStart As Integer) As Integer
    Dim intFieldCharCount As Integer
    Dim strTestString As String * 1
    Dim strFieldMark As String * 1
    Dim blnFLAG As Boolean

    intFieldCharCount = 0
    strFieldMark = ",,"

    blnFLAG = True

    While (blnFLAG)
        strTestString = Mid(strStringIn, intFieldStart, 1)
        If (strTestString = strFieldMark) Then
            blnFLAG = False
        Else
            intFieldStart = intFieldStart + 1
            intFieldCharCount = intFieldCharCount + 1
        End If
    Wend

    ReturnFieldLength = intFieldCharCount
Public Function TimeStampCheck() As Boolean
    Dim intFLAG As Boolean

    intFLAG = False

    If (StrComp(strPOS_TimeStamp, strVEL_TimeStamp, vbTextCompare) = 1) Then
        intFLAG = True
    End If

    TimeStampCheck = intFLAG
End Function

Public Sub StringDataToFloat()

    dblPOS_TimeStamp = Val(strPOS_TimeStamp)
    dblLatitude = Val(strLatitude)
    dblLongitude = Val(strLongitude)
    dblHeight = Val(strHeight)

    dblVEL_TimeStamp = Val(strVEL_TimeStamp)
    dblHorizontalSpeed = Val(strHorizontalSpeed)
    dblGroundTrack = Val(strGroundTrack)
    dblVerticalSpeed = Val(strVerticalSpeed)

    ' New variables for packet data. Should not need those listed above.
    dblTimeStamp = Val(strTimeStamp)
    dblEast = Val(strEast)
    dblNorth = Val(strNorth)
    dblUp = Val(strUp)
    dblVelocity = Val(strVelocity)
    dblGroundTrack = Val(strGroundTrack)
    dblPitch = Val(strPitch)
    dblRoll = Val(strRoll)

End Sub

Public Sub ClearStrings()

    strPOS_TimeStamp = ""
    strLatitude = ""
    strLongitude = ""
    strHeight = ""

    strVEL_TimeStamp = ""
    strHorizontalSpeed = ""
    strGroundTrack = ""

End Sub
strVerticalSpeed = ""

End Sub
B.4 Synthetic Vision Variables

'** Author: Douglas Burch
'** Date: 12 Jan 2002

'*** The following are used as global variables in the eHUD Software.

'*** Constants ***
Public PI As Double
Public dblTimeRateOfChange As Double
Public i As Integer
Public j As Integer
Public k As Integer

' * East, North. Up. 
Public dblEast As Double
Public dblNorth As Double
Public dblUp As Double

' * File Manipulators. 
Public intGPSFileHandler As Integer
Public intFLTDATAHandler As Integer
Public blnAttitudeCalculated As Boolean

' * State Vector Information taken from GPS Strings. 
Public strPOS_Timestamp As String
Public strLatitude As String
Public strLongitude As String
Public strHeight As String
Public strVEL_Timestamp As String
Public strHorizontalSpeed As String
Public strVerticalSpeed As String
Public strGroundTrack As String

' * Coverted State Vector Information from strings to floating points. 
Public dblPOS_Timestamp As Double
Public dblLatitude As Double
Public dblLongitude As Double
Public dblHeight As Double
Public dblVEL_Timestamp As Double
Public dblHorizontalSpeed As Double
Public dblVerticalSpeed As Double
Public dblGroundTrack As Double

' * i,j,k Components of the Velocity Vector. 
Public dblVelocityVector(3) As Double
Public dblPreviousVelocityVector(3) As Double
Public dblVelocityNorth As Double
Public dblVelocityEast As Double
Public dblVelocityDown As Double
Public dblVelocityMagnitude As Double
Public dblUnitVelocityVector(3) As Double

'* i,j,k Components of the Acceleration Vector.
Public dblAcceleration(3) As Double
Public dblNormalAcceleration(3) As Double
Public dblTangentialAcceleration(3) As Double
Public dblAccelerationMagnitude As Double

'* i,j,k Components of the Gravitational Acceleration Vector.
Public dblGravity(3) As Double
Public dblNormalGravity(3) As Double
Public dblTangentialGravity(3) As Double
Public dblGravityMag As Double

'* Flight Path Angle Data.
Public dblFlightPathAngleDeg As Double
Public dblFlightPathAngleRAD As Double

'* Pseudo-Roll Data
Public dblPseudoRollRad As Double
Public dblPseudoRollDeg As Double

'* Lift and Reference Vectors.
Public dblLiftVector(3) As Double
Public dblLiftMag As Double
Public dblReferenceVector(3) As Double
Public dblReferenceMag As Double

'* Filters for Acceleration, Pseudo-Roll, and Flight Path Angle.
'Public dblAcceleration_FILTER(3)(3) as Double, NOT used at this time.
Public dblPSR_FILTER(3) As Double
Public dblFPA_FILTER(3) As Double

'*** The following variables are used to control the attitude of the display. ***
Public snglRoll As Single
Public snglPitch As Single
Public snglHeading As Single

***** Camera XYZ Position *****
Public snglPosition_X As Single 'Local X
Public snglPosition_Y As Single 'Local Height
Public snglPosition_Z As Single 'Local Z

***** Aircraft ENU position as single *****
Public varEast As Single
Public varNorth As Single
Public varUp As Single

Public varEast As Single
Public varNorth As Single
Public varUp As Single

' Convention for Y and Z are switched. X cross Z = Y.
Public snglAircraft_X As Single ' Local Aircraft X (East)
Public snglAircraft_Y As Single ' Local Aircraft Y (Height)
Public snglAircraft_Z As Single ' Local Aircraft Z (North)

'**** Camera Positioning on Terrain ****
Public snglTerrainHeight As Single
Public snglTerrainOffset As Single
Public snglCameraOffset As Single

'**** Flight Data Information *****
Public snglAltitude As Single
Public snglSpeed As Single
Public snglGroundTrack As Single
Public snglFlightPathAngle As Single
Public snglPseudoRoll As Single

'**** Variant Flight Data Information *****
Public varAltitude As Variant
Public varSpeed As Variant
Public varGroundTrack As Variant
Public varFlightPathAngle As Variant
Public varPseudoRoll As Variant

'**** Temporary Flight Data Storage ****
Public snglCurrentFlightPathAngle As Single
Public snglCurrentPseudoRoll As Single
Public snglCurrentAltitude As Single
Public snglTempAltitude As Single
Public snglCurrentGroundTrack As Single

Public snglPreviousFlightPathAngle As Single
Public snglPreviousPseudoRoll As Single
Public snglPreviousAltitude As Single
Public snglPreviousGroundTrack As Single

'**** Trig functions expect a double as input, must convert sngl to double through variant. *****
Public dblFPARad As Double
Public dblFPADeg As Variant
Public varFPADeg As Variant

'**** Movement of Camera through Bitmap *****
Public dt As Single
Public snglPixel_Per_Meter As Single
Public snglAltitudeCorrection As Single
Public gColResult As R3DCOLLISIONRESULT_TYPE ' Collision Detection.

Public blnRUN As Boolean

Public snglStartTime As Single
Public singleStopTime As Single
Public singleDelayTime As Single
Public intFile As Integer
Public strFileHeader As String
Public strFilePath As String

'***** New Variables for Processing Packet Data *****
Public strTimeStamp As String
Public strEast As String
Public strNorth As String
Public strUp As String
Public strVelocity As String
Public strGroundTrack As String
Public strPitch As String
Public strRoll As String

Public dblTimeStamp As Double
Public dblEast As Double
Public dblNorth As Double
Public dblUp As Double
Public dblVelocity As Double
Public dblGroundTrack As Double
Public dblPitch As Double
Public dblRoll As Double

Public strDataPacket As String
Public blnPoundFound As Boolean

'***** Glide Slope *****
Public dblGlideSlope As Double