NAVAL
POSTGRADUATE
SCHOOL

Monterey, California

THESIS

AUTONOMOUS LANDING SYSTEM FOR A UAV

by

Mariano I. Lizarraga

March 2004

Committee Supervisor:	Roberto Cristi
Committee Co-Supervisor:	Isaac Kaminer
Committee Member:	Robert G. Hutchins

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** March 2004	**3. REPORT TYPE AND DATES COVERED** Engineer's Thesis

4. TITLE AND SUBTITLE: Autonomous Landing System for a UAV	**5. FUNDING NUMBERS**
6. AUTHOR(S) : Mariano I. Lizarraga	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	**12b. DISTRIBUTION CODE**

13. ABSTRACT *(maximum 200 words)*

This thesis is part of an ongoing research conducted at the Naval Postgraduate School to achieve the autonomous shipboard landing of Unmanned Aerial Vehicles (UAV). Two main problems are addressed in this thesis. The first is to establish communication between the UAV's ground station and the Autonomous Landing Flight Control Computer effectively. The second addresses the design and implementation of an autonomous landing controller using classical control techniques. Device drivers for the sensors and the communications protocol were developed in ANSI C. The overall system was implemented in a PC104 computer running a real-time operating system developed by The Mathworks, Inc. Computer and hardware in the loop (HIL) simulation, as well as ground test results show the feasibility of the algorithm proposed here. Flight tests are scheduled to be performed in the near future.

14. SUBJECT TERMS Unmanned Aerial Vehicles, UAV, Silver Fox, Autonomous Landing, Shipboard Landing, UAV Control System, Real Time Workshop, xPC Target, Piccolo.	**15. NUMBER OF PAGES** 147
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

AUTONOMOUS LANDING SYSTEM FOR A UAV

Mariano I. Lizarraga
Lieutenant Junior Grade, Mexican Navy
B.S. Naval Sciences Engineering, Mexican Naval Academy, 1995

Submitted in partial fulfillment of the
requirements for the degrees of

ELECTRICAL ENGINEER

and

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
March 2004

Author:	Mariano I. Lizarraga
Approved by:	Roberto Cristi, Committee Supervisor
	Isaac Kaminer, Committee Co-Supervisor
	Robert G. Hutchins, Committee Member
	John P. Powers, Chairman Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis is part of an ongoing research conducted at the Naval Postgraduate School to achieve the autonomous shipboard landing of Unmanned Aerial Vehicles (UAV). Two main problems are addressed in this thesis. The first is to establish communication between the UAV's ground station and the Autonomous Landing Flight Control Computer effectively. The second addresses the design and implementation of an autonomous landing controller using classical control techniques. Device drivers for the sensors and the communications protocol were developed in ANSI C. The overall system was implemented in a PC104 computer running a real-time operating system developed by The Mathworks, Inc. Computer and hardware in the loop (HIL) simulation, as well as ground test results show the feasibility of the algorithm proposed here. Flight tests are scheduled to be performed in the near future.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION ... 1
- A. BACKGROUND ... 1
- B. OBJECTIVES ... 1
- C. CURRENT CONTROL SYSTEM SETUP ... 3
  - 1. The Piccolo Avionics ... 5
  - 2. Ground Station ... 6
  - 3. Operator Interface ... 7
  - 4. Pilot Manual Control ... 7
  - 5. Two-Layer Piccolo Protocol ... 8
- D. SUMMARY ... 9

II. AUTONOMOUS LANDING CONTROL SETUP ... 11
- A. OVERVIEW OF THE SOLUTION APPROACH ... 11
  - 1. Position and Orientation of the Recovering Net ... 13
  - 2. Glide Slope Calculation ... 14
  - 3. Recovery Segment One: Glide Slope Capture ... 17
  - 4. Recovery Segment Two: Glide Slope Tracking ... 19
- B. INTERCONNECTION TO THE CURRENT CONTROL SETUP ... 23
  - 1. RS-232 Reader of the Piccolo Protocol ... 24
  - 2. Decoder of the Piccolo Protocol ... 25
  - 3. Encoder of the Piccolo Protocol ... 27
  - 4. RS-232 Writer of the Piccolo Protocol ... 28
  - 5. Limitations on the Data Rate ... 29
- C. AUTONOMUS LANDING FLIGHT CONTROL COMPUTER ... 30
- D. DEVELOPMENT OF THE DEVICE DRIVERS ... 33
  - 1. GPS ... 33
  - 2. Differential Correction Beacon ... 36
- E. SUMMARY ... 36

III. CONTROL SYSTEM DESIGN ... 37
- A. CONTROLLER DESIGN ... 37
  - 1. Decoupling of Motion ... 38
  - 2. Inner Loop Autopilot ... 39
    - *a. Lateral Channel ... 39*
    - *b. Longitudinal Channel ... 42*
  - 3. Outer Loop Control Feedback ... 46
    - *a. Longitudinal and Lateral Channels ... 47*
    - *b. Path Realignment ... 51*
- B. INNER LOOP AUTOPILOT IMPLEMENTATION IN THE UAV AVIONICS ... 52
- C. COMPLETE AUTONOMUS LANDING SYSTEM INTEGRATION ... 54
  - 1. Graphical User Interface (GUI) ... 56

D. SUMMARY 57

IV. SIMULATION AND GROUND TESTS RESULTS 59

A. SIMULATION 59

1. Simulation Scenarios 59

a. Ideal Conditions: No Measurement Noise and No Wind 62

b. Measurement Noise and No Wind 65

c. Measurement Noise and Wind Disturbances 69

d. Results Summary 73

B. HARDWARE IN THE LOOP SIMULATION 76

1. Setup 76

2. Results 78

C. GROUND TEST 82

1. Setup 82

2. Results 84

D. SUMMARY 86

V. CONCLUSIONS AND FOLLOW-ON WORK 87

A. CONCLUSIONS 87

B. FOLLOW-ON WORK 88

APPENDIX A: SOURCE CODE OF THE DRIVERS 89

APPENDIX B: DERIVATION OF THE 6-DOF AERODDYNAMIC MODEL OF THE SILVER FOX UAV 107

A. 6-DOF AIRCRAFT MODEL 107

1. Equations of Motion 107

a. Linear Dynamics Equations 107

b. Angular Dynamics Equations 109

c. Attitude (Euler) Equations 110

2. Forces and Moments on Aircraft 112

a. Aerodynamic Forces and Moments 112

b. Gravitational and Propulsive Forces and Moments 114

3. Dimensionless Aerodynamic Coefficients 115

4. Engine Model 117

5. Atmosphere and Wind Model 119

6. Actuators Models 121

APPENDIX C: NUMERICAL VALUES OF THE AERODYNAMIC COEFFICIENTS OF THE SILVER FOX 6DOF MODEL 123

LIST OF REFERENCES 125

INITIAL DISTRIBUTION LIST 127

LIST OF FIGURES

Figure 1. Current Control System Setup of the Silver Fox UAV. 4
Figure 2. Piccolo Closed Loop Control System. 5
Figure 3. Piccolo Avionics System. 6
Figure 4. Telemetry Screen of the Piccolo Operator Interface Software. 7
Figure 5. Piccolo Pilot Manual Control. 8
Figure 6. Piccolo Communications Two-layer Protocol Structure. 9
Figure 7. Landing Segment One: Alignment of the Velocity Vectors in the *XY* Plane in Preparation for Landing. 11
Figure 8. Segment Two: Keeping the Velocity Vectors Aligned in the *XY* Plane (a) while Initiating Descent by Aligning the Velocity Vector in the *YZ* Plane with a Predetermined Glide Slope (b). 12
Figure 9. Recovering Net Layout. 13
Figure 10. Glide Slope Localization with Respect to the Middle Point of the Recovering Net. 14
Figure 11. A New Coordinate Frame Fixed to the Recovering Net and Aligned with the Glide Slope. 15
Figure 12. Simulink Implementation of the Glide Slope Calculation. 16
Figure 13. Simulink Implementation of Recovery Segment One: Glide Slope Capture. 19
Figure 14. Desired Inertial Trajectory for Recovery Segment Two: Glide Slope Tracking. 20
Figure 15. Simulink Implementation of Recovery Segment Two: Glide Slope Tracking. 23
Figure 16. Interconnection of the Autonomous Landing Flight Control Computer to the Piccolo Ground Station. 24
Figure 17. Complete Reading/Decoding of the Piccolo Protocol. (a) Receiver, Reader, and Decoder. (b) Reader/Decoder. (c) Decoder. 26
Figure 18. Encoder of the Piccolo Protocol; (a) Inner Level Wrapper; (b) Outer Level Wrapper. 28
Figure 19. Position and Velocity Filter. 29
Figure 20. Complementary Filter in the Vertical Channel. 30
Figure 21. Autonomous Landing Flight Control Computer. 31
Figure 22. Complete Interconnection of the Autonomous Landing Flight Control Computer to Peripheral Devices. 32
Figure 23. NMEA-0183 Message Structure (From: Ref. [6].) 34
Figure 24. Complete Reading/Decoding of the GPS Data (a) Receiver, Reader, and Decoder. (b) Reader/Decoder. (c) Decoder. 35
Figure 25. Top Level View of the Inner and Outer Loop Control System Design. 38
Figure 26. Root Locus and Bode Plot of the Transfer Function from δ_a to *r*. 40

Figure 27. Root Locus and Bode Plot of the Transfer Function from δ_a to r with a Compensator in the Forward Path. 41
Figure 28. Root Locus and Bode Plot of the Transfer Function from δ_r to r with a Compensator in the Forward Path. 42
Figure 29. Root Locus and Bode Plot of the Transfer Function from δ_e to q with a Gain Compensator in the Forward Path. 43
Figure 30. Root Locus of the Closed Loop Speed Controller. 44
Figure 31. Inner Loop Control (Autopilot) Implementation in Simulink. 45
Figure 32. Top Level Diagram of the Control Scheme Implemented in the Outer Loop. 47
Figure 33. Root Locus of the Outer Loop Longitudinal Channel Using Only PD Control. 48
Figure 34. Root Locus and Bode Plot of the Outer Loop Longitudinal Channel Using PD Control and a Controller in the Feedback Path. 48
Figure 35. Root Locus and Bode Plot of the Outer Loop Lateral Channel Using PD Control and a Controller in the Feedback Path. 49
Figure 36. Simulink Implementation of the Complete Outer Loop Control Feedback 50
Figure 37. Scenario Showing a Large Angle Difference between the UAV's Velocity Vector and $\vec{T}$ in the XY Plane. 51
Figure 38. Simulink Implementation of the Path Realignment Logic. 52
Figure 39. NPS and Piccolo Autopilot Implementation in the Avionics. 53
Figure 40. Top Level View of the System Integration. 55
Figure 41. Graphical User Interface (GUI) Used to Configure the Autopilot Parameters, Present In-Flight Data and Log the Variables. 56
Figure 42. Simulink Model Used for Ship Landing Simulation. 60
Figure 43. Simulink Model Used for Runway Landing Simulation. 61
Figure 44. *YZ* Plot of the Position of the UAV's CG When Crossing the Plane of the Recovering Net. (a) Shipboard Landing. (b) Runway Landing. No Measurement Noise and No Wind. 62
Figure 45. Three-Dimensional Plot of a Landing Approach. (a) Shipboard Landing. (b) Runway Landing. No Measurement Noise and No Wind. 63
Figure 46. *XY* and *YZ* Plots of a Landing Approach. (a) Shipboard Landing. (b) Runway Landing. No Measurement Noise and No Wind. 64
Figure 47. Position and Velocity Errors in Y Compared with the Commanded Yaw Rate. (a) Shipboard Landing. (b) Runway Landing. No Measurement Noise and No Wind. 65
Figure 48. Position and Velocity Errors in Z Compared with the Commanded Pitch Rate. (a) Shipboard Landing. (b) Runway Landing. No Measurement Noise and No Wind. 65
Figure 49. *YZ* Plot of the Position of the UAV's CG When Crossing the Plane of the Recovering Net. (a) Shipboard Landing. (b) Runway Landing. Measurement Noise and No Wind. 66

Figure 50. Three-Dimensional Plot of a Landing Approach. Measurement Noise and No Wind. (a) Shipboard Landing. (b) Runway Landing. Measurement Noise and No Wind. 67
Figure 51. *XY* and *YZ* Plots of a Landing Approach. (a) Shipboard Landing. (b) Runway Landing. Measurement Noise and No Wind. 68
Figure 52. Position and Velocity Errors in Y Compared with the Commanded Yaw Rate. (a) Shipboard Landing. (b) Runway Landing. Measurement Noise and No Wind. 69
Figure 53. Position and Velocity Errors in Z Compared with the Commanded Pitch Rate. (a) Shipboard Landing. (b) Runway Landing. Measurement Noise and No Wind. 69
Figure 54. YPG's Wind Profile. (After Ref. [12].) 70
Figure 55. *YZ* Plot of the Position of the UAV's CG When Crossing the Plane of the Recovering Net. (a) Shipboard Landing. (b) Runway Landing. Measurement Noise and No Wind. 71
Figure 56. Three-Dimensional Plot of a Landing Approach. (a) Shipboard Landing. (b) Runway Landing. Measurement Noise and Wind. 72
Figure 57. *XY* and *YZ* Plots of a Landing Approach. (a) Shipboard Landing. (b) Runway Landing. Measurement Noise and Wind. 72
Figure 58. Position and Velocity Errors in Y Compared with the Commanded Yaw Rate. (a) Shipboard Landing. (b) Runway Landing. Measurement Noise and Wind. 73
Figure 59. Position and Velocity Errors in Z Compared with the Commanded Pitch Rate. (a) Shipboard Landing. (b) Runway Landing. Measurement Noise and Wind. 73
Figure 60. Histograms of the Occurrences of the Position Errors in *Y* and *Z* for Each of the Simulation Conditions. Shipboard Landing. 75
Figure 61. Histograms of the Occurrences of the Position Errors in *Y* and *Z* for Each of the Simulation Conditions. Runway Landing. 75
Figure 62. Diagram of the HIL Simulation. 77
Figure 63. HIL Simulation Setup at the Naval Postgraduate School Unmanned Systems Lab. 77
Figure 64. Three-Dimensional Plot of a Landing Approach in HIL Simulation. Noise and Wind 78
Figure 65. *XY* and *YZ* Plots of a Landing Approach in HIL Simulation. Noise and Wind. 79
Figure 66. *YZ* Plot of the Position of the UAV's CG When Crossing the Plane of the Recovering Net. HIL Simulation. 80
Figure 67. Position and Velocity Errors in Y Compared with the Commanded Yaw Rate in HIL Simulation. Noise and Wind. 81
Figure 68. Position and Velocity Errors in Z Compared with the Commanded Pitch Rate in HIL Simulation. Noise and Wind. 81
Figure 69. Dropouts in the Sensor Data during HIL Simulation. 82

Figure 70. Ground Test Setup at the NPS's UAV Lab. (a) Piccolo Avionics Tied to a Wheel Cart. (b) Ground Station GPS Receivers. (c) Ground Station Setup. 83
Figure 71. Triangular Path Followed by the Piccolo Avionics during the Ground Tests. 84
Figure 72. Drifting and Variation of the Signals Provided from the Sensors in the Ground Test. 85
Figure 73. GPS Seconds from the Telemetry Message of the Piccolo Protocol and the Delays between Updates Obtained during the Ground Test. 86
Figure B 1. Main Configuration Window of LinAir Pro. Top View of the Modeled Silver Fox UAV. 115
Figure B 2. Nondimensionalized Section Lift Coefficient vs. Panel Y Coordinate. LinAir Pro. 116
Figure B 3. Silver Fox's Static Thrust Diagram. 118
Figure B 4. Variation of Thrust Available with Forward Speed. (From Ref. [10].) 119
Figure B 5. Simulink Implementation of the Engine Model. 119
Figure B 6. Atmosphere and Wind Model in Simulink. 120
Figure B 7. Complete Simulink Model of the Silver Fox. 121
Figure B 8. Servomechanism (Actuator) on a UAV. 121
Figure B 9. Aileron, Elevator, Rudder and Throttle Actuators. Simulink Model. . 122

LIST OF TABLES

Table 1. Location of the Dominant Closed Loop Poles, Gain Margins and Phase Margins of the Inner Loop (Autopilot) Controller. 46
Table 2. Location of the Dominant Closed Loop Poles, Gain Margins and Phase Margins of the Outer Loop Control. 50
Table 3. Means and Standard Deviations of the Landing Errors. No Measurement Noise and No Wind. 63
Table 4. Variances for the White Additive Noise Used in the Sensors for Simulation. 66
Table 5. Means and Standard Deviations of the Landing Errors. Measurement Noise and No Wind. 67
Table 6. Means and Standard Deviations of the Landing Errors. Measurement Noise and Wind. 71
Table 7. Summary of Mean and Standard Deviations for the Three Scenarios 74

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

First and foremost, I would like to thank my beloved wife Erika for her invaluable support throughout our stay here in Monterey. Without her, this would have been much harder. Thanks to our daughter Valeria for being our inspiration to be better each day.

My sincerest gratitude to the Mexican Navy for giving me this opportunity, specially to Viceadmiral Alberto Castro Rosas. An NPS alumni himself, his support to come here, and during my stay, has been invaluable.

I am grateful to Drs. Vladimir Dobrokhodov and Oleg Yakimenko for their support and patience during the endless hours working at the Unmanned Systems Lab. To them, my deepest gratitude, for always, regardless of their schedule, going out of their way to make complicated topics understandable and showing me what teaching is all about.

Special thanks are due to Dr. Isaac Kaminer for giving me the opportunity to work in this interesting project and sharing his insight in control systems. Without his support, my stay here at NPS would not have been as rewarding as it has been.

I would also like to thank my control systems professors and advisers, Drs. Roberto Cristi, Robert Hutchins, and Xiaoping Yun. Thank you for all those in-class and after-class hours you took to explain the *ins and outs* of control systems to me.

Thanks are also due to Jerry Lentz, for his support in many technical details of the hardware involved in this project. I would also like to thank him for encouraging all sorts of technical and non-technical discussions and helping me understand the topics in question.

Finally, I would like to thank Ron Russell for all his support during the editing process of this thesis and for showing a very high sense of responsibility and professionalism in what he does.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

This thesis is part of an ongoing research conducted at the Naval Postgraduate School to achieve the autonomous shipboard landing of Unmanned Aerial Vehicles, particularly in a ship underway. The Silver Fox UAV is used as a test platform for the algorithm proposed here.

The control algorithm presented takes advantage of the Piccolo system developed by Cloud Cap Technologies. All the control commands generated by the Autonomous Landing system are sent through the underlying communications channel between Piccolo's ground station and the UAV.

Two main problems are addressed in this thesis. The first is to establish communication between the UAV's ground station and the Autonomous Landing Flight Control Computer effectively. The second addresses the design and implementation of an autonomous landing controller, using classical control techniques. Generally speaking, the controller forces the UAV to track a glide slope from an imaginary point in space to the landing point.

Device drivers for the sensors and the communications protocol were developed in ANSI C and implemented in S-Functions inside Simulink models. Mathworks' XPC Target and Real Time Workshop tools were used to compile and download the system into a PC104 form factor computer running a real-time operating kernel.

Computer and hardware in the loop (HIL) simulation under different scenarios, as well as ground test results, present the feasibility of the scheme proposed here.

Flight tests are scheduled to be performed in Tucson, Arizona, in the near future.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. BACKGROUND

The use of Unmanned Aerial Vehicles in the battlefield has given the modern war fighter advantages over the enemy, proven to be extremely useful in the recent wars in Afghanistan and Iraq. The ability to relay real-time imagery to field commanders, collect intelligence and launch precise attacks on identified targets were very important roles that the UAV played in these wars.

Also, at the tactical level, small lightweight UAV that were carried to the battlefront and launched in a short time without the need of a landing strip proved to be useful and provided an invaluable advantage to fulfill surveillance missions for Marines and Special Forces units.

This thesis addresses the autonomous landing of such small, lightweight tactical UAV, particularly on a ship underway. Currently, when a ship recovers those types of UAVs, the pilot flies the UAV close to the vessel, stops the engine and opens a parachute to drop the airplane to the sea. Then the ship stops and recovers the UAV in any way possible.

The work presented herein is part of a major effort to achieve an unassisted landing of the UAV by taking control of the airplane when it is sufficiently close to the moving vessel and guiding it to a landing point. In the first phase, the UAV will be guided to a recovery net aboard the ship. Once this algorithm is successfully tested, future work may focus on a more robust control algorithm that actually lands the airplane in a short-landing platform aboard a ship.

B. OBJECTIVES

The entire project aims to give battle ready intelligence about what lies ahead for the troops, allowing them to be better prepared for an impending engagement. The NPS's current efforts focus on making the Silver Fox UAV a true hands-off aircraft with an autonomous recovering capability. The system potentially promises to be seamlessly integrated with manned aircraft missions, carrier air traffic control, and deck operations. This approach guarantees greater military safety, simplicity and flexibility than anything previously available in sea operations. Such capability, when fully integrated, will eliminate the need for the Army and Navy units to train and to maintain Radio Control (RC) pilots to launch and recover the UAVs. The implementation will allow safe, high quality reconnaissance at a fraction of the cost of current sea-based aircrafts with much longer endurance and utility for a much wider range of ships. The proposed solution invaluably contributes not only onboard implementation of the autonomous landing algorithm but also provides compatibility of Silver Fox avionics with many UAVs currently used by the Army and Navy, and with many research projects in industry and academia.

This thesis is organized as follows: The remaining of this chapter presents a summary of the current control setup used onboard the UAV. Chapter II describes the solution approach successfully used to guide the UAV from an imaginary point in space to the net aboard the recovering vessel. It also addresses all the issues concerned with the connectivity between the Ground Station and the proposed Autonomous Landing Flight Control Computer. Chapter III describes the control system design, a detailed six-degrees-of-freedom (6-DOF) aircraft model used to test the control system, and the implementation of the controller to run in real-time. Chapter IV presents the simulation and ground test results of the proposed control system. Chapter V presents the conclusions and the recommendations for follow-on work to this research. Appendix A lists the C source code for the device drivers developed for this work. Appendix B presents a detailed derivation of the 6-DOF model for the Silver Fox UAV. Appendix C lists the numerical values of the dimensionless aerodynamic coefficients of the Silver Fox UAV.

This thesis seeks to implement an autonomous recovery control system in a PC104 computer using the Silver Fox UAV as a test platform. The Silver Fox is a small tactical UAV with a 6-foot long fuselage, 8-foot wingspan, and 20-pound takeoff weight, which can fly up to 1,000 feet for four hours at a cruise speed of 23 meters per second.

The scope of the work presented here includes the following:

- Effectively establish communication between the Ground Station and the Autonomous Landing Flight Control Computer.
- Develop all the necessary device drivers to read and to decode data from the Ground Station.
- Develop all the necessary device drivers to read data from the devices connected to the Autonomous Landing Flight Control Computer.
- Design a controller so the UAV is capable of tracking a trajectory from an imaginary point in space to the landing point.
- Implement the controller in a PC104 form factor computer to be connected to the Ground Station via RS232 serial connection.
- Perform real-time and Hardware In the Loop (HIL) simulation of the overall design to prove its feasibility.

C. CURRENT CONTROL SYSTEM SETUP

The Silver Fox UAV mission and flight control is done using a system called Piccolo, developed by Cloud Cap Technologies, Inc., which is capable of controlling multiple UAVs in flight.

The Piccolo control system setup, shown in Figure 1, consists of four main parts: an Avionics control system located onboard the UAV, a Ground Station, a Pilot Manual Control, and the Operator Interface. These four elements provide a reliable way to fly the UAV and enable the end user to program desired routes for the UAV via waypoints. For take off and landing of the UAV a *pilot in the loop* mode is enabled in which the ground station only retransmits the signal generated by the pilot using the Manual Control.

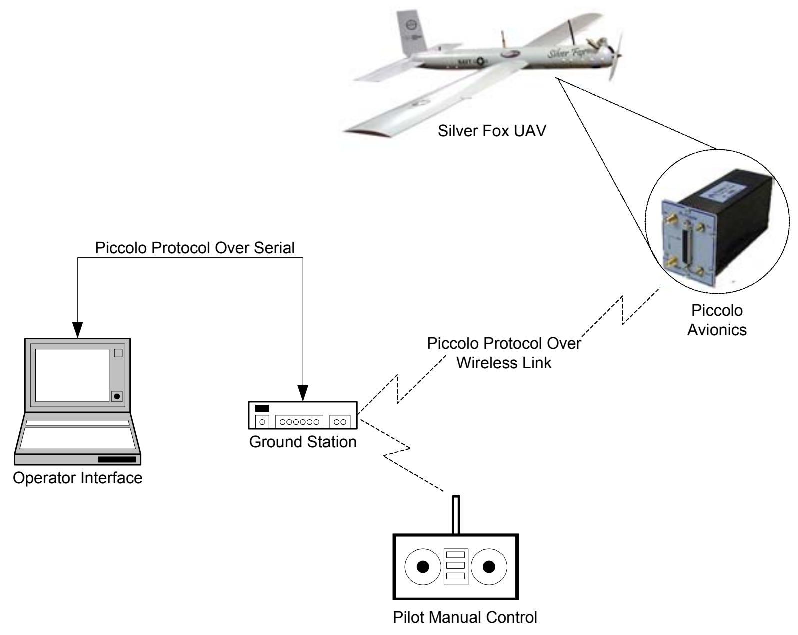


Figure 1. Current Control System Setup of the Silver Fox UAV.

As shown in Figure 2, the Piccolo Control setup has two control loops: a faster, inner loop onboard the Silver Fox and a slower outer loop, which is implemented on the Ground Station.

This current control setup uses application-specific two-layer protocol to communicate between all the components of the system. A more detailed description of the main components is given in the following subsections.

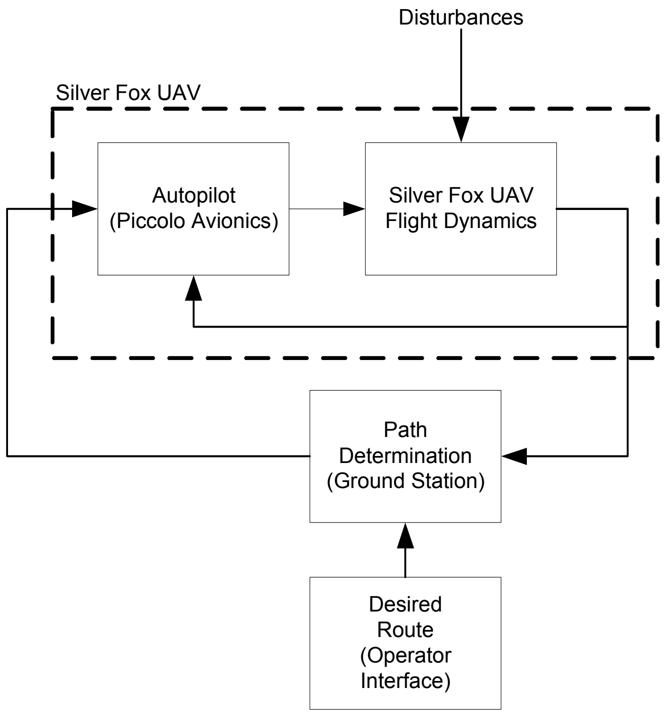

Disturbances
Silver Fox UAV
Autopilot (Piccolo Avionics)
Silver Fox UAV Flight Dynamics
Path Determination (Ground Station)
Desired Route (Operator Interface)

Figure 2. Piccolo Closed Loop Control System.

1. The Piccolo Avionics

The Piccolo Avionics System, shown in Figure 3, is an autopilot designed to track the commanded path transmitted by the Ground Station. It generates the required signals for the control surfaces of the airplane (ailerons, elevator and rudder) and the engine’s thrust. The main processor is an MPC555 microcontroller based on the PowerPC architecture delivering a 40-MHz operation, including the hardware floating point.

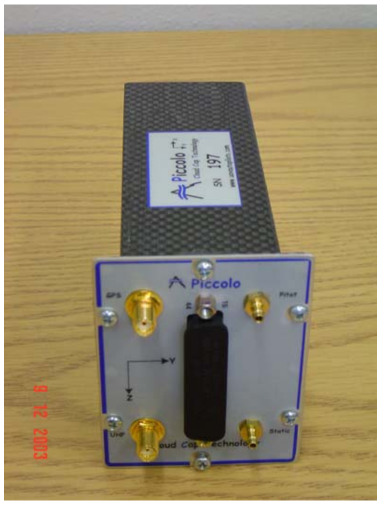


Figure 3. Piccolo Avionics System.

The sensor group of the system includes: three rate gyros and three accelerometers used to determine the UAV's attitude; a Motorola G12 Global Positioning System (GPS) to determine its geodetic position, and a set of dynamic and static pressure sensors coupled with a thermometer to determine the airplane's true air speed and altitude.

The data link to the Ground Station is provided by a 900 MHz /2.4 GHz radio modem. All the data transmitted is wrapped in a custom-developed two-layer protocol, which will be thoroughly explained in the last subsection of this chapter.

2. Ground Station

The Piccolo Ground Station has two very important roles in the system: to provide the communication link between the Avionics, the Operator Interface, and the Pilot Manual Control; and to convert the intentions of the end user captured through the Operator Interface, into meaningful commands for the autopilot.

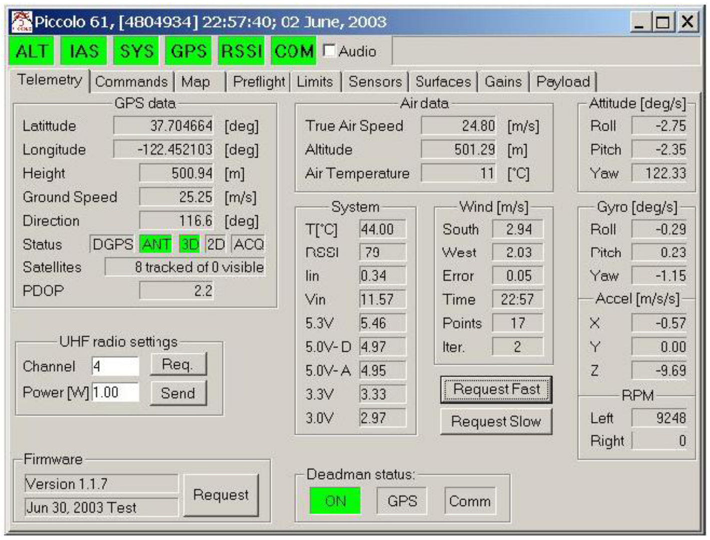
The main processor, as in the Avionics, is an MPC555 microcontroller based on the PowerPC architecture. It is equipped with a Subminiature B Connector (SMB) for the GPS antenna and a Bayonet Neill-Concelman (BNC) connector for the Ultra High Frequency (UHF) communications antenna. The connection to the Operator Interface is done through a standard 9-pin serial cable.

3. Operator Interface

The Operator Interface consists of a portable computer running a Windows 2000 ® operating system and a custom developed software to allow the end user to configure and to operate the Piccolo System.

The Operator Interface software, shown in Figure 4, has nine configuration screens that provide the end user with an interface to:

- perform a preflight checklist,
- configure the desired gains of the control system in the Autopilot,
- program a desired route for the UAV via waypoints,
- display the reading of all the sensors aboard the UAV, and
- display the current position of the UAV in a geographical chart.



GPS data

Latitude	37.704664	[deg]
Longitude	-122.452103	[deg]
Height	500.94	[m]
Ground Speed	25.25	[m/s]
Direction	116.6	[deg]
Status	DGPS ANT 3D 2D ACQ	
Satellites	8 tracked of 0 visible	
PDOP	2.2	

Air data

True Air Speed	24.80	[m/s]
Altitude	501.29	[m]
Air Temperature	11	[°C]

Attitude [deg/s]

Roll	-2.75
Pitch	-2.35
Yaw	122.33

System

T[°C]	44.00
RSSI	79
Iin	0.34
Vin	11.57
5.3V	5.46
5.0V-D	4.97
5.0V-A	4.95
3.3V	3.33
3.0V	2.97

Wind [m/s]

South	2.94
West	2.03
Error	0.05
Time	22:57
Points	17
Iter.	2

Gyro [deg/s]

Roll	-0.29
Pitch	0.23
Yaw	-1.15

Accel [m/s/s]

X	-0.57
Y	0.00
Z	-9.69

RPM

Left	9248
Right	0

UHF radio settings

Channel	4
Power [W]	1.00

Firmware

Version 1.1.7
Jun 30, 2003 Test

Deadman status: ON GPS Comm

Figure 4. Telemetry Screen of the Piccolo Operator Interface Software.

4. Pilot Manual Control

Mainly used for take off and landing, the Pilot Manual Control provides the end user with a way to override the commands generated by the Ground Station and allows a qualified UAV Pilot to take control of the UAV.

The Pilot Manual Control, shown in Figure 5, is a commercial Futaba® radio remote control typically used for midsize hobby radio control airplanes.

Figure 5. Piccolo Pilot Manual Control.

5. Two-Layer Piccolo Protocol

As mentioned above, the data transmitted between all the devices of the Piccolo Control System are wrapped in a custom-developed two-layer communications protocol.

Shown in Figure 6 is the overall structure of the Piccolo Protocol. The inner layer is structured as follows: two synchronization bytes used to signal the receiving state machine that a stream packet may be forthcoming; one byte to specify whether the message is a control, telemetry, waypoint data, or any of the 24 other types of message that the protocol can handle; one byte to indicate the size of the payload; the payload itself according to the type of message; and, a two-byte Cyclic Redundancy Check (CRC) code used to detect transmission errors.

The outer layer is structured as follows: two synchronization bytes used to signal the receiving state machine that a stream packet may be forthcoming; two network address bytes that contain the network address of the Piccolo Avionics that is sending the message; a one-byte stream identifier, which can be any of the eight types supported by the protocol; one byte to indicate the size of the payload, which in this case is the whole inner layer; one byte containing an eight bit XOR checksum of the header bytes, bytes zero through six; the payload, which in this case is the complete inner layer; and, a two-byte CRC code to detect for transmission errors.

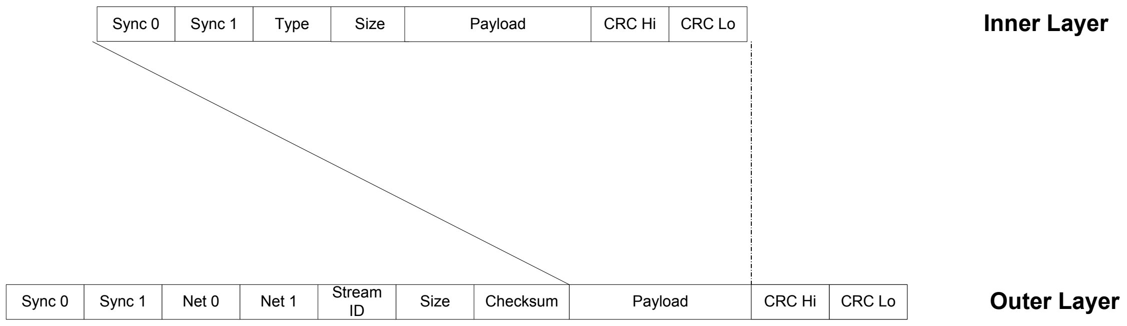


Figure 6. Piccolo Communications Two-layer Protocol Structure.

As will be shown in the following chapters, understanding the Piccolo protocol will be extremely important to this work. The autonomous landing control system proposed decodes the information received from the avionics in this format and wrap the issued control commands in the protocol so that the onboard avionics has no problem interpreting them.

D. SUMMARY

In this chapter, the importance of UAVs in modern warfare and how they relate to this work was briefly presented. It stated the objectives to be achieved by this work and explained the current control setup used by the Silver Fox UAV. The following chapter presents the proposed hardware/software approach to solve the autonomous landing of the UAV on a ship underway.

THIS PAGE INTENTIONALLY LEFT BLANK

II. AUTONOMOUS LANDING CONTROL SETUP

This chapter presents the autonomous landing algorithm and discusses all the issues related to the interconnection of the current control setup with the proposed solution. The hardware, its peripherals, and the development of the required device drivers are explained.

A. OVERVIEW OF THE SOLUTION APPROACH

The position of an object in a three-dimensional space is typically expressed with respect to a predetermined coordinate system. For purposes of this work, let a local coordinate system, or reference frame *{L},* be a right-handed system with its *X* axis pointing to the true north, the *Y* axis pointing east, and the *Z* axis be orthogonal to *X* and *Y*, thus pointing to the zenith.

Guiding an aircraft from any point in space to a predetermined landing point moving in a straight line can be described in two segments. The first, shown in Figure 7, is to fly the aircraft from any initial condition to a point in space where the aircraft not only has the correct position behind the moving landing point, but also its velocity vector in the *XY* plane is aligned with that of the landing point.

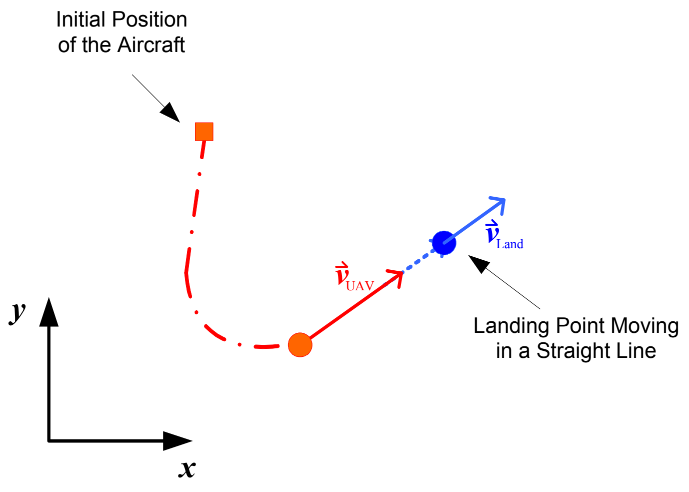
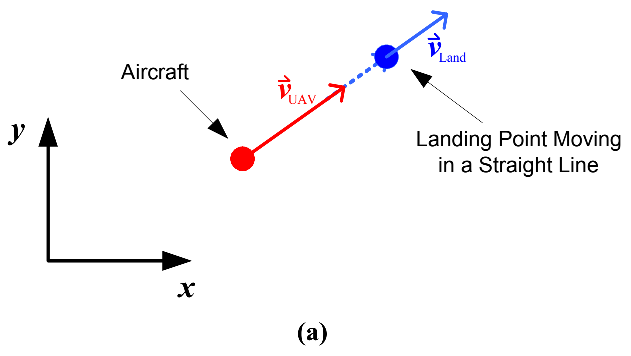

Initial Position of the Aircraft
$\vec{v}_{\text{Land}}$
$\vec{v}_{\text{UAV}}$
Landing Point Moving in a Straight Line
y
x

Figure 7. Landing Segment One: Alignment of the Velocity Vectors in the *XY* Plane in Preparation for Landing.

The second segment, shown in Figure 8, consists of keeping the velocity vector of the aircraft in the *XY* plane aligned with that of the moving landing point (Figure 8a) as the aircraft initiates its descent by aligning its velocity vector in the *YZ* plane with a predetermined glide slope that ends in the desired touchdown point (Figure 8b).

From the approach described above, it can be seen that five parameters that must be known at all times. These are the position and velocity vector of the landing point, and the position, velocity vector and attitude of the aircraft.



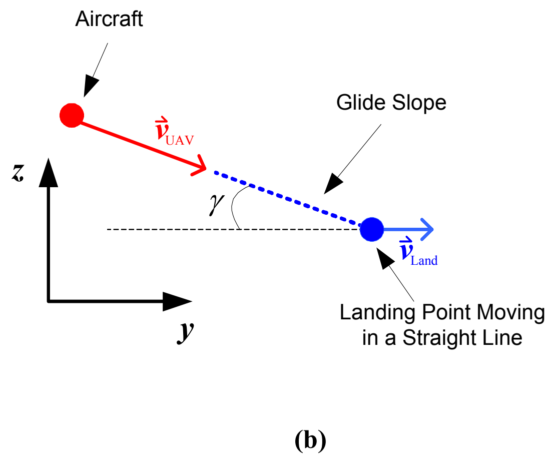


Figure 8. Segment Two: Keeping the Velocity Vectors Aligned in the *XY* Plane (a) while Initiating Descent by Aligning the Velocity Vector in the *YZ* Plane with a Predetermined Glide Slope (b).

Furthermore, the following assumptions are made:

- Assumption 1: If the ship does not have a deck long enough to be used as a landing strip, then the UAV must be forced to collide with a recovering net.
- Assumption 2: The recovering ship does not change course throughout the recovery approach.

1. Position and Orientation of the Recovering Net

In order to describe the position and orientation of the recovering net unequivocally, the following must be known: a point in the net and a normal vector [1]. Since a normal vector to the net is not readily available, it must be calculated.

Let the vector $\hat{z}_n = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$ denoting the vertical orientation of the net be known and constant and two known points in the net be p_1 and p_2, as described in Figure 9.

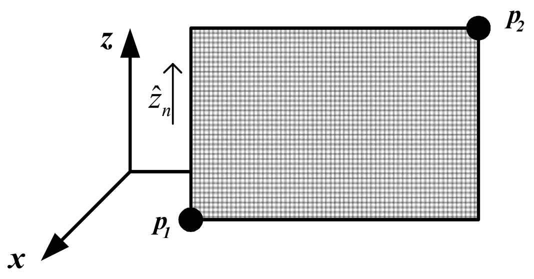


Figure 9. Recovering Net Layout.

Then the direction of the main diagonal of the net can be calculated as:

$$\hat{d} = \frac{1}{\left\| \vec{p}_2 - \vec{p}_1 \right\|} \left(\vec{p}_2 - \vec{p}_1 \right) \tag{1}$$

and the position of the middle point in the net can be obtained by

$$\vec{p}_t = \frac{\left(\vec{p}_2 - \vec{p}_1 \right)}{2}. \tag{2}$$

With these two coplanar vectors, then a normal vector to the net can be obtained by:

$$\hat{n} = \frac{1}{\left\| \hat{z}_n \times \hat{d} \right\|} \left(\hat{z}_n \times \hat{d} \right) \tag{3}$$

which, together with any of the three known positions, fully describe the plane of the net in a three-dimensional space.

2. Glide Slope Calculation

Once the exact position and orientation of the recovering net is known, then the glide slope can be calculated. Two parameters determine the position of the glide slope: the angle γ formed between the glide slope and the horizon, and the offset *D*, which specifies the length of the glide slope.

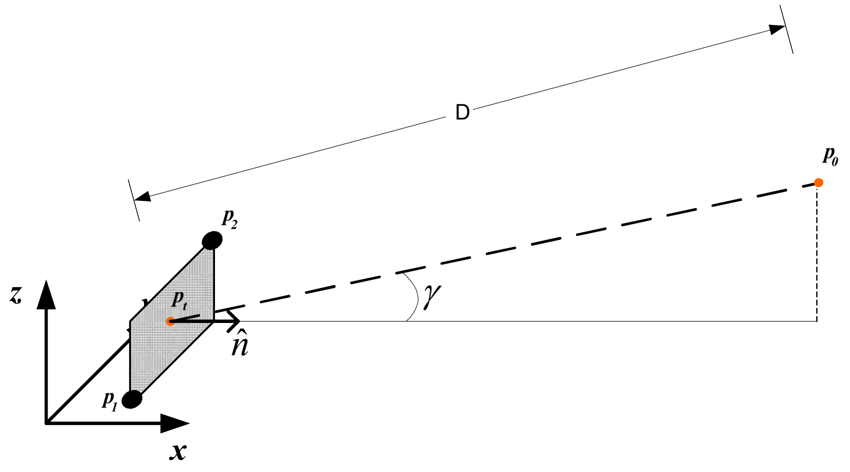


Figure 10. Glide Slope Localization with Respect to the Middle Point of the Recovering Net.

From the geometry of the glide slope with respect to the net, shown in Figure 10, the position of the top of the glide slope is obtained as follows:

$$\begin{aligned} x_t &= D\cos\gamma \\ y_t &= 0 \\ z_t &= D\sin\gamma. \end{aligned} \tag{4}$$

The above equations assume that the origin is located at $\vec{p}_t$; therefore, a more general way to state the position of the top of the glide slope without making any assumptions about the position of $\vec{p}_t$ is given by:

$$\vec{p}_0 = \vec{p}_t + [D\cos\gamma \quad 0 \quad D\sin\gamma]. \tag{5}$$

Even though Equation (5) is general, it still puts a large constraint in the calculation of $\vec{p}_0$, which is that the normal vector to the net, $\hat{n}$, be aligned with the X axis; therefore, a more general solution is needed. To do that it is necessary to introduce a new coordinate frame fixed to the point $\vec{p}_t$ with its X axis aligned with the glide slope, as shown in Figure 11, where the *local* or *fixed* frame *{L}* is denoted with a left superscript *L*, and those described in the *glide slope* frame *{G}* are denoted with a left superscript *G*.

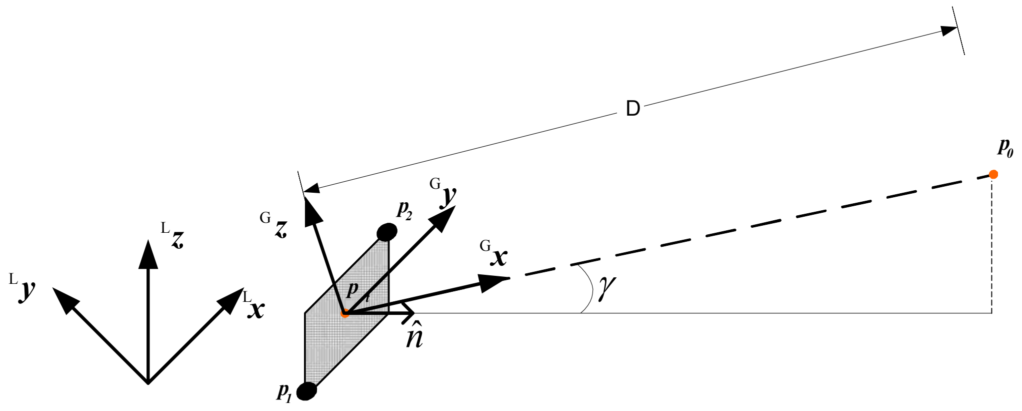


Figure 11. A New Coordinate Frame Fixed to the Recovering Net and Aligned with the Glide Slope.

The above figure shows that the vector that describes the direction of the net, in the *glide slope* frame, is the unitary vector ${}^G\hat{x}$ or $\left({}^G[1 \quad 0 \quad 0]\right)^T$. Also, by inspecting the above figure, one can determine that the Euler angles [2] that describe the rotation of the *glide slope* frame with respect to the *local* frame are given by:

$$\begin{aligned} \phi &= 0 \\ \theta &= \gamma \\ \psi &= \tan^{-1}\left(\frac{y_{\hat{n}}}{x_{\hat{n}}}\right), \end{aligned} \tag{6}$$

where $y_{\hat{n}}$ and $x_{\hat{n}}$ denote the *x* and *y* components of the vector $\hat{n}$. Knowing this, one can calculate the position of the top of the glide slope as:

$${}^{L}\vec{p}_0 = {}^{L}\vec{p}_t + {}^{L}_{G}R\hat{n}, \tag{7}$$

where ${}^{L}_{G}R$ denotes the rotation matrix from *{G}* to *{L}* derived in [2], written below for reference:

$${}^{G}_{L}R = \begin{bmatrix} \cos\theta\cos\psi & \sin\phi\sin\theta\cos\psi - \cos\phi\sin\psi & \cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi \\ \cos\theta\sin\psi & \sin\phi\sin\theta\sin\psi + \cos\phi\cos\psi & \cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi \\ -\sin\theta & \sin\phi\cos\theta & \cos\phi\cos\theta \end{bmatrix}, \tag{8}$$

and

$${}^{L}_{G}R = \left({}^{G}_{L}R\right)^{T}. \tag{9}$$

It is important to note that, because the vector $\hat{z}_n$ is held constant, this stabilizes the glide slope, allowing it to move in the XY plane with the ship, but uncouples it from the rolling movement of the ship.

Figure 12 shows the complete implementation in Simulink of the above described procedure to calculate the top of the glide slope:

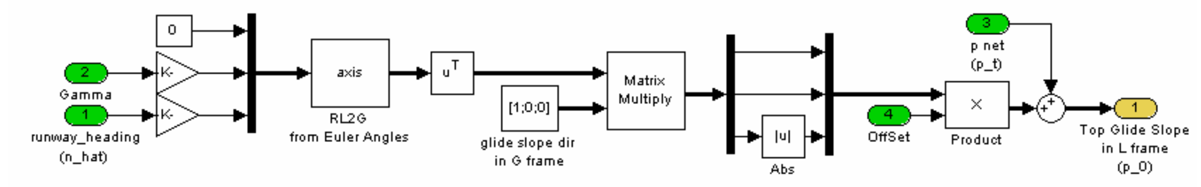


Figure 12. Simulink Implementation of the Glide Slope Calculation.

In order to simplify the notation, for the rest of the work presented herein, all the vectors are assumed to be resolved in the local reference frame *{L}* unless they are identified by a left superscript, denoting that they are resolved in a different frame.

3. Recovery Segment One: Glide Slope Capture

During the first segment of the recovery, it is assumed that the UAV's initial height is approximately the same as that of the glide slope, and that the UAV's autopilot keeps it constant during this segment.

Given that the position of the UAV is known, then the distance from the top of the glide slope to the UAV is given by:

$$d = \left\| \vec{p}_0 - \vec{p}_{UAV} \right\|. \tag{10}$$

Let a new frame be fixed to the center of gravity (CG) of the UAV and denoted by the unit orthogonal vectors $\vec{T}$, $\vec{N}$ and $\vec{B}$, where the vector $\vec{T}$ points from the UAV's CG to the top of the glide slope. Such a frame is generally called a *TNB* or *Frenet* Frame *{F}* [1]. Then $\vec{T}$, resolved in *{L}* is given by:

$$\vec{T} = \frac{1}{d}\left(\vec{p}_0 - \vec{p}_{UAV}\right). \tag{11}$$

The vector $\vec{N}$ lies on the local horizon of the UAV and is determined as follows:

$$\vec{N} = \frac{1}{\left\| \begin{bmatrix} -y_T & x_T & 0 \end{bmatrix} \right\|} \begin{bmatrix} -y_T \\ x_T \\ 0 \end{bmatrix}, \tag{12}$$

where x_T and y_T are the *X* and *Y* components of $\vec{T}$, respectively. The vector $\vec{B}$ is given by:

$$\vec{B} = \vec{T} \times \vec{N}, \tag{13}$$

where $\times$ denotes a vector cross product.

The approach used to assure that the UAV will be located at the top of the glide slope is to align the velocity vector of the UAV with the vector $\vec{T}$. To do that, one must know the velocity error resolved in *{F}* in order to generate the appropriate control signal.

The velocity error resolved in *{L}* is given by:

$$\vec{V}_e = \vec{T} v_c - \vec{v}_{UAV}, \tag{14}$$

where v_c is the commanded speed.

Resolving $\vec{V}_e$ in *{F}* leads to the following:

$${}^F\vec{V}_e = \begin{bmatrix} \vec{T}^T \\ \vec{N}^T \\ \vec{B}^T \end{bmatrix} \left(\vec{T} - \vec{v}_{UAV} \right) = \begin{bmatrix} v_c - \vec{T} \cdot \vec{v}_{UAV} \\ -\vec{N} \cdot \vec{v}_{UAV} \\ -\vec{B} \cdot \vec{v}_{UAV} \end{bmatrix}. \tag{15}$$

From the above result, one can see that the *Y* and *Z* components of the velocity error resolved in *{F}* are the projections of the velocity and taken with the opposite sign. It is important to note that when the velocity vector of the UAV is aligned with the vector $\vec{T}$ and has the desired magnitude, the error becomes $\begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T$ as expected.

Although the above result is feasible, it means that a control signal for the UAV's thrust must also be generated (the *X* component of the velocity error). In the early design process, it was decided that thrust would be controlled by the autopilot and not by the landing algorithm; therefore, a simplified velocity error for the first segment is given by:

$$\vec{V}_e = \begin{bmatrix} 0 \\ -\vec{N} \cdot \vec{v}_{UAV} \\ -\vec{B} \cdot \vec{v}_{UAV} \end{bmatrix}, \tag{16}$$

where the *X* component of Equation (15) was arbitrarily made to be zero.

The above approach guarantees that the UAV will pass through the top of the glide slope but does not take into consideration the fact that by the time the

UAV reaches the top of the glide slope, its velocity vector should be aligned with the glide slope. To solve this, the distance to the top of the glide slope is constantly calculated. When the distance is less than a predetermined value, then the velocity and position errors are calculated by segment two, which forces the alignment of the UAV's velocity vector with the glide slope. This will be described in the following subsection.

Figure 13 shows the implementation of the first segment, in Simulink.

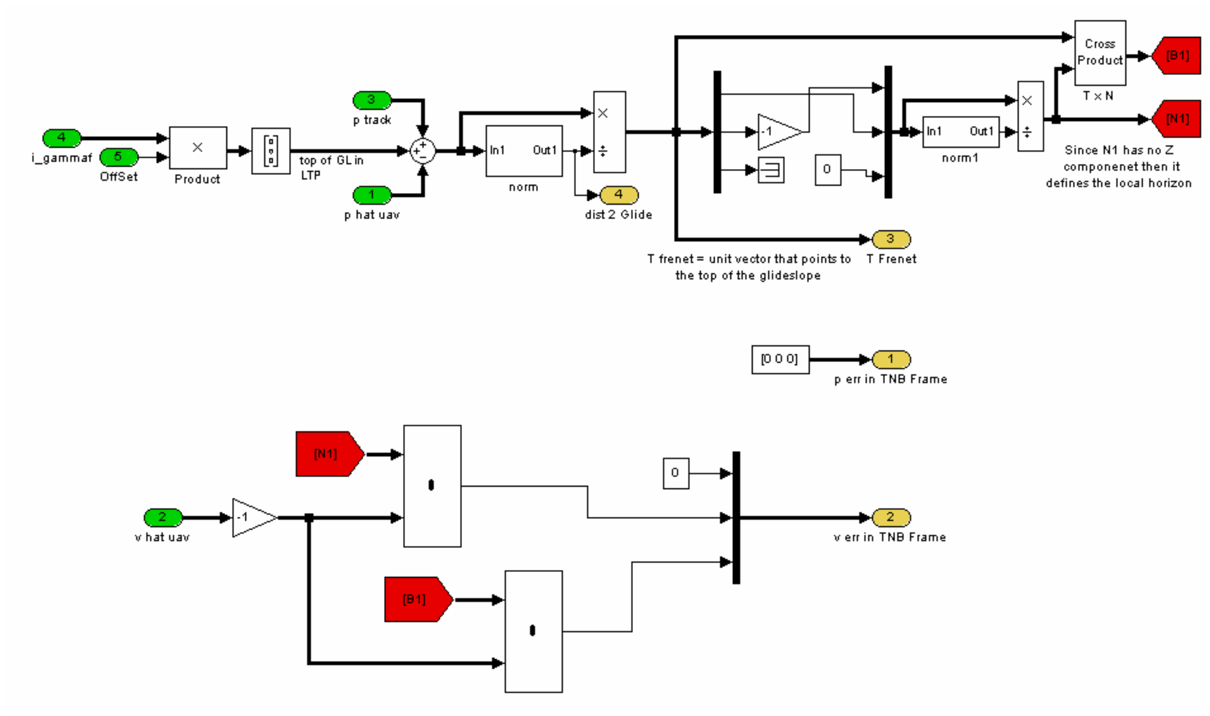


Figure 13. Simulink Implementation of Recovery Segment One: Glide Slope Capture.

4. Recovery Segment Two: Glide Slope Tracking

As described in the beginning of this chapter, the glide slope tracking segment is slightly more complicated than the capture segment because the velocity vectors of the UAV and the target have to be aligned in the *XY* and *YZ* planes, and also because the position of the UAV with respect to the glide slope is relevant.

For this segment, the vector $\vec{T}$ is defined as the unit vector that points from the top of the glide slope to the center of the recovering net. The unit vectors $\vec{N}$ and $\vec{B}$ are obtained in the same way as described in the previous subsection.

Let the desired inertial trajectory be parameterized, as presented in [3], in terms of its arc-length *S.* From the geometry shown in Figure 14, one can see that $\vec{P}_c(S)$ is defined as follows:

$$\vec{P}_c(S) = \vec{p}_0 + \vec{T}S \tag{17}$$

where $\vec{p}_0$ is the top of the glide slope and *S* can take values between 0 and D, thus $0 \leq S \leq D$.

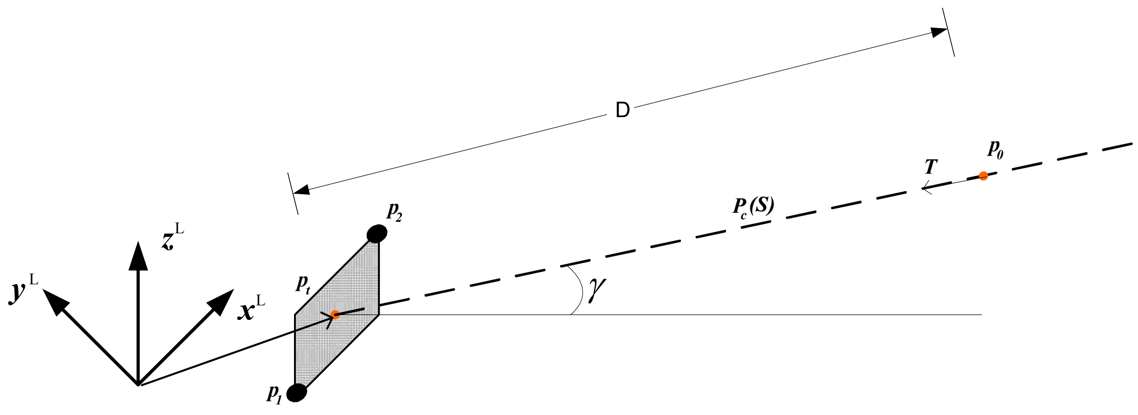


Figure 14. Desired Inertial Trajectory for Recovery Segment Two: Glide Slope Tracking.

Also, from Figure 14, it can be stated that:

$$\vec{p}_0 = \vec{p}_t - \vec{T}D; \tag{18}$$

and substituting Equation (18) into Equation(17) this can be rewritten as:

$$\vec{P}_c(S) = \vec{p}_t + \vec{T}(S - D). \tag{19}$$

Having defined the desired inertial trajectory, the position error can be defined as the difference between the desired and the actual position of the UAV, thus:

$$\vec{P}_e(S) = \vec{P}_c(S) - \vec{p}_{UAV}. \tag{20}$$

Therefore, it is necessary to find the parameter *S* that minimizes the position error in the mean square sense, which can be mathematically stated as:

$$\min_S \left(\left\| \vec{P}_c(S) - \vec{p}_{UAV} \right\|_2^2 \right). \tag{21}$$

The above problem can be restated as the solution of the following equation [1]:

$$\frac{d}{dS}\left(\left(\vec{P}_c(S) - \vec{p}_{UAV} \right)^T \left(\vec{P}_c(S) - \vec{p}_{UAV} \right) \right) = 0. \tag{22}$$

Substituting the actual value of $\vec{P}_c(S)$, Equation (22) can be rewritten as:

$$\frac{d}{dS}\left(\left(\vec{p}_t + \vec{T}(S+D) - \vec{p}_{UAV} \right)^T \left(\vec{p}_t + \vec{T}(S+D) - \vec{p}_{UAV} \right) \right) = 0 \tag{23}$$

from which the optimal value of *S* is obtained and is given by:

$$S = \frac{1}{2}\left(\vec{T}^T \left(\vec{p}_{UAV} - \vec{p}_t \right) + \left(\vec{p}_{UAV}{}^T - \vec{p}_t{}^T \right) \vec{T} \right) + D. \tag{24}$$

Substituting the above value of *S* into Equation (19) after the algebraic simplifications, this can be rewritten as:

$$\vec{P}_c(S) = \vec{p}_t + \frac{1}{2}\vec{T}\left(\left(\vec{p}_{UAV} - \vec{p}_t \right)\vec{T} + \vec{T}^T \left(\vec{p}_{UAV} - \vec{p}_t \right) \right). \tag{25}$$

Finally, substituting the above expression for $P_c(S)$ into Equation (20), the position error resolved in *{L}* is given by:

$$\vec{P}_e(S) = \left(\vec{p}_t - \vec{p}_{UAV} \right) + \frac{1}{2}\vec{T}\left(\left(\vec{p}_{UAV} - \vec{p}_t \right)\vec{T} + \vec{T}^T \left(\vec{p}_{UAV} - \vec{p}_t \right) \right). \tag{26}$$

In the same manner as in segment one, the position error is resolved in *{F}* by rotating it as follows:

$$^{F}\vec{P}_{e}\left(S\right)=\begin{bmatrix}\vec{T}^{T}\\ \vec{N}^{T}\\ \vec{B}^{T}\end{bmatrix}\left[\left(\vec{p}_{t}-\vec{p}_{UAV}\right)+\frac{1}{2}\vec{T}\left(\left(\vec{p}_{UAV}-\vec{p}_{t}\right)\vec{T}+\vec{T}^{T}\left(\vec{p}_{UAV}-\vec{p}_{t}\right)\right)\right], \tag{27}$$

which, after doing the algebra simplifications, can be rewritten as:

$$^{F}\vec{P}_{e}\left(S\right)=\begin{bmatrix}0\\ \vec{N}\cdot\left(\vec{p}_{t}-\vec{p}_{UAV}\right)\\ \vec{B}\cdot\left(\vec{p}_{t}-\vec{p}_{UAV}\right)\end{bmatrix}. \tag{28}$$

Following the same procedure as above, one can show that the velocity error is given by:

$$^{F}\vec{V}_{e}\left(S\right)=\begin{bmatrix}0\\ \vec{N}\cdot\left(\vec{v}_{t}-\vec{v}_{UAV}\right)\\ \vec{B}\cdot\left(\vec{v}_{t}-\vec{v}_{UAV}\right)\end{bmatrix}. \tag{29}$$

Figure 15 shows the implementation of the second segment in Simulink.

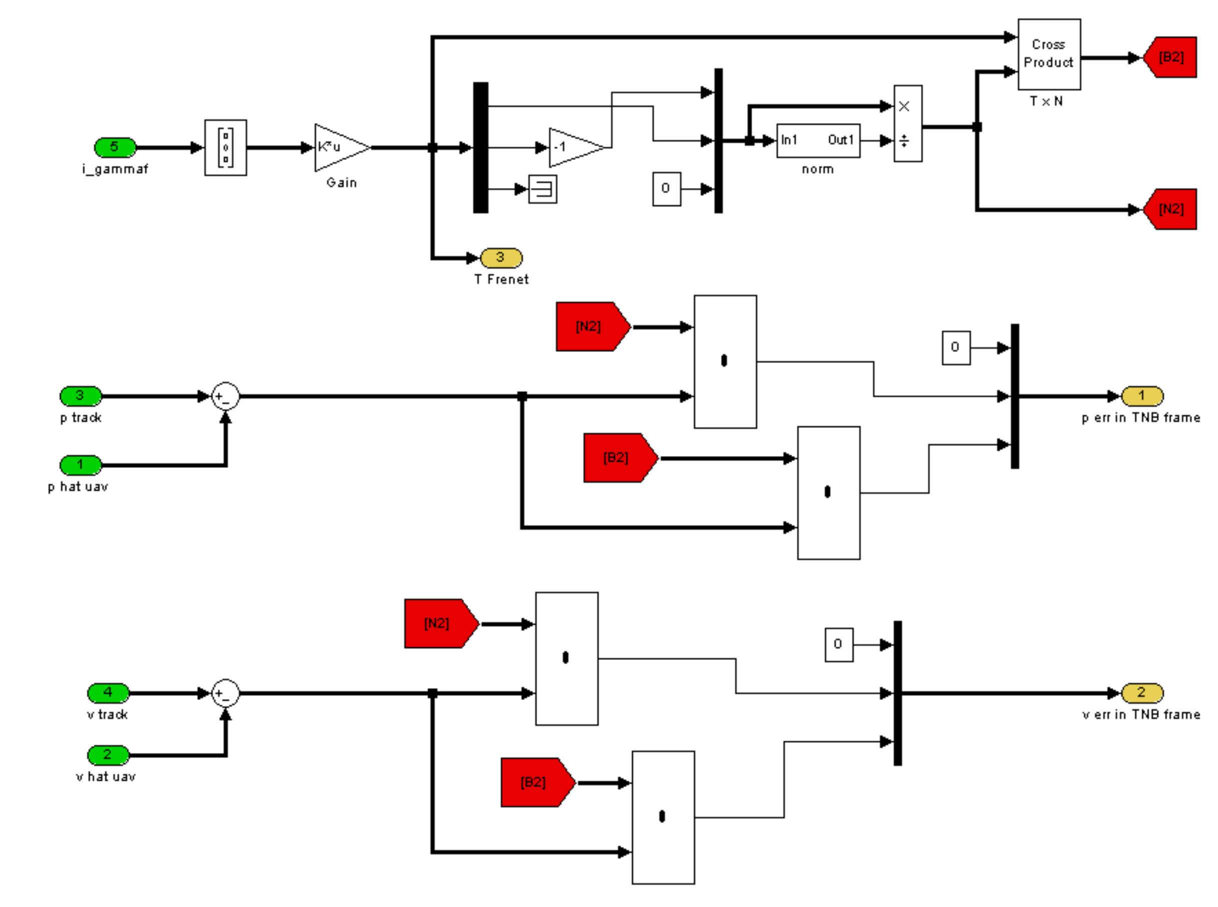


Figure 15. Simulink Implementation of Recovery Segment Two: Glide Slope Tracking.

B. INTERCONNECTION TO THE CURRENT CONTROL SETUP

So far, it has been assumed that the data from the UAV is available. This is actually possible due to the interconnection between the Autonomous Landing Flight Control Computer and Piccolo’s Ground Station.

To establish this interconnection, drivers had to be developed in C programming language, implemented in S-Functions in Simulink, and then fine-tuned to produce the data required, in the correct units and at the highest sampling rate possible.

As in any protocol, two main libraries had to be developed, one to read data and one to write (or send) data. For simplicity and to keep the amount of code manageable, each one was separated in two parts:

- For the reader, one that actually handled the serial port and stripped the useful data from the headers, and another to actually decode the bits into meaningful measurements of the UAV's sensors.
- For the writer, one that packed the commands for the UAV in a byte format the autopilot would understand and put the headers to it , and another that actually handled the serial communication.

An overall picture of the interconnection is shown in Figure 16:

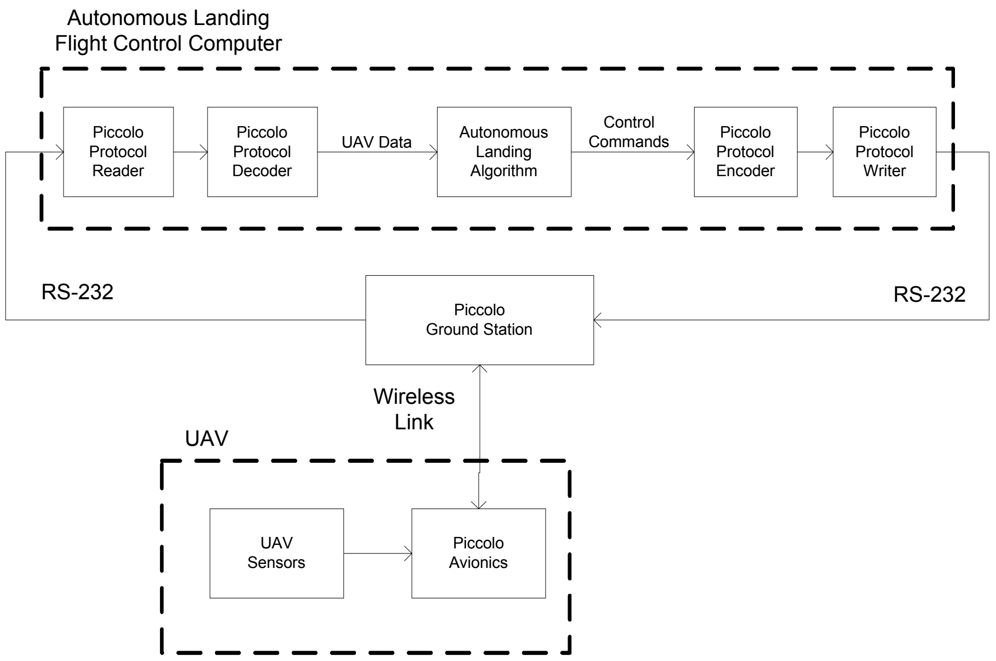


Figure 16. Interconnection of the Autonomous Landing Flight Control Computer to the Piccolo Ground Station.

1. RS-232 Reader of the Piccolo Protocol

The reader of the Piccolo Protocol was, without doubt, the longest and hardest of all the drivers developed for the work presented here, consisting of more than 650 lines of code. The reader performs the following sequence of operations:

- Buffers data as it is received from the serial port.
- Performs the outer level checksum verification.
- Performs the outer level CRC verification.
- Performs the inner level CRC verification.
- Verifies that the received message is of interest, i.e., that it contains data that is required by the autonomous landing algorithm.

- Parses the useful data and produces three outputs: the valid data, the header type that indicates which type of message it is, and a function call that lets the system know that new data is available.

The complete source code of the S-Function implementation is included in Appendix A.

2. Decoder of the Piccolo Protocol

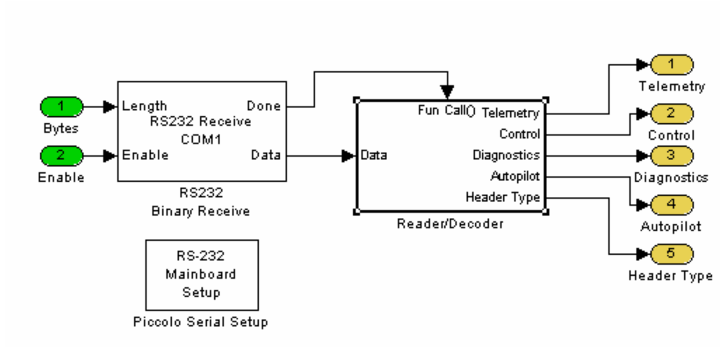
Once the data has been validated and stripped of the headers of both layers, it must be converted to meaningful data that can be used by the autonomous landing algorithm.

Four types of messages from the Piccolo Protocol were required: telemetry, control, diagnostics and autopilot. Each of these four messages has its own format as described in [4]. Therefore, four separate C S-Functions were written to decode the received data. The complete source code of each of the decoders is included in Appendix A.

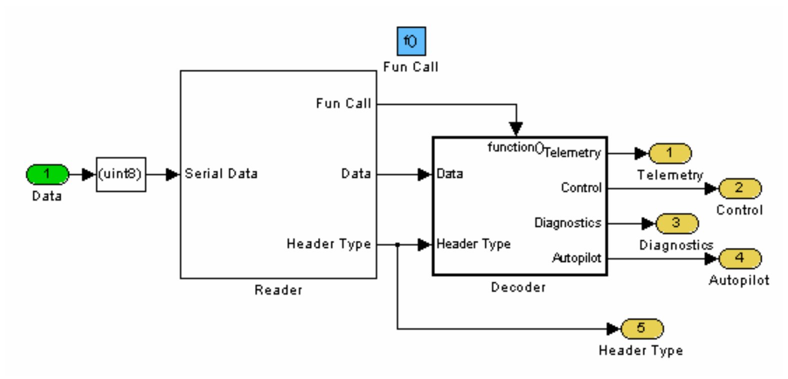
The complete reading/decoding scheme is shown in Figure 17. The raw data is received through the serial port and streamed to a predetermined size buffer. Once the buffer is full, the serial receiver issues a function call that indicates to the following block that new data is available (Figure 17a). Once the reader receives a complete buffer, it processes it, as described in the previous section and sends to the output the following (Figure 17b):

- The header type which indicates what type of message was received,
- a function call which indicates to the decoder that new data is available, and
- the data.

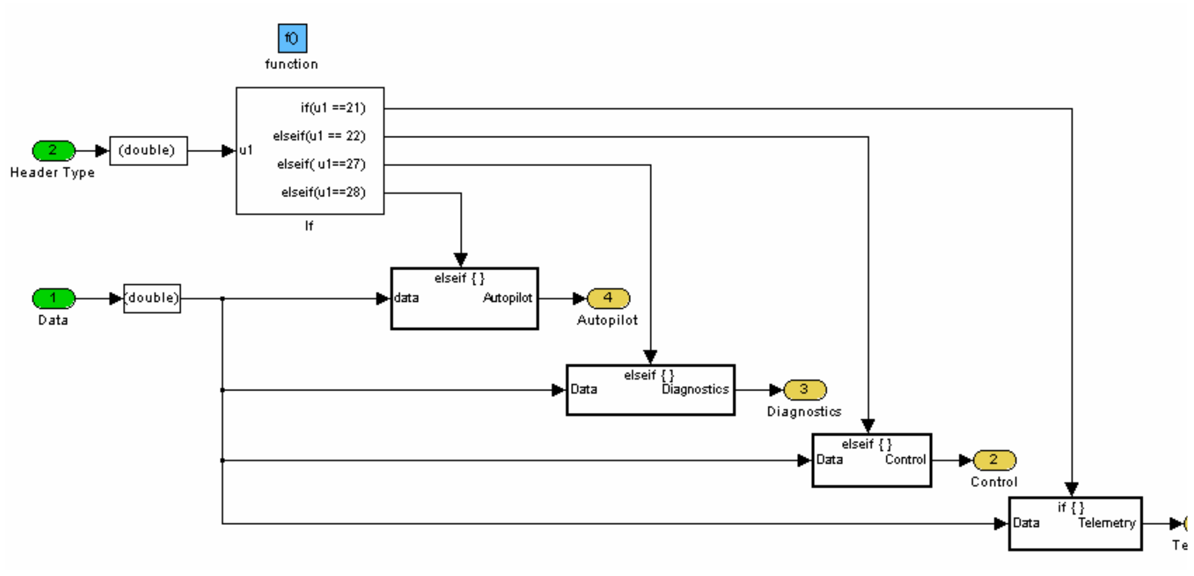
The decoder, according to the type of message that will be decoded, enables the appropriate block that generates the meaningful data (Figure 17c).



(a)



(b)



(c)

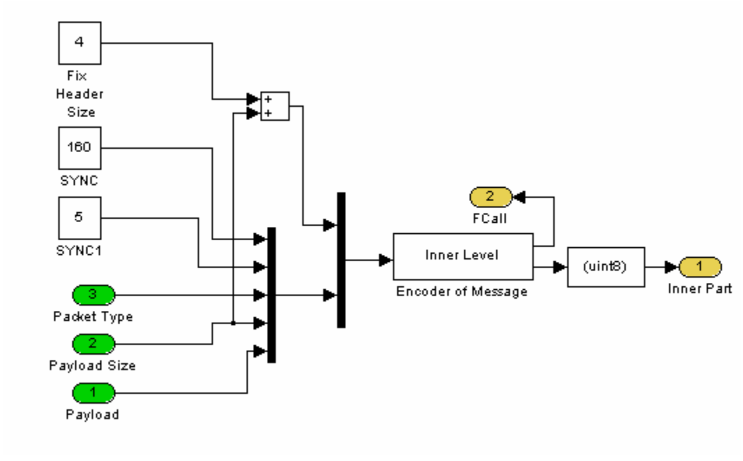
Figure 17. Complete Reading/Decoding of the Piccolo Protocol. (a) Receiver, Reader, and Decoder. (b) Reader/Decoder. (c) Decoder.

3. Encoder of the Piccolo Protocol

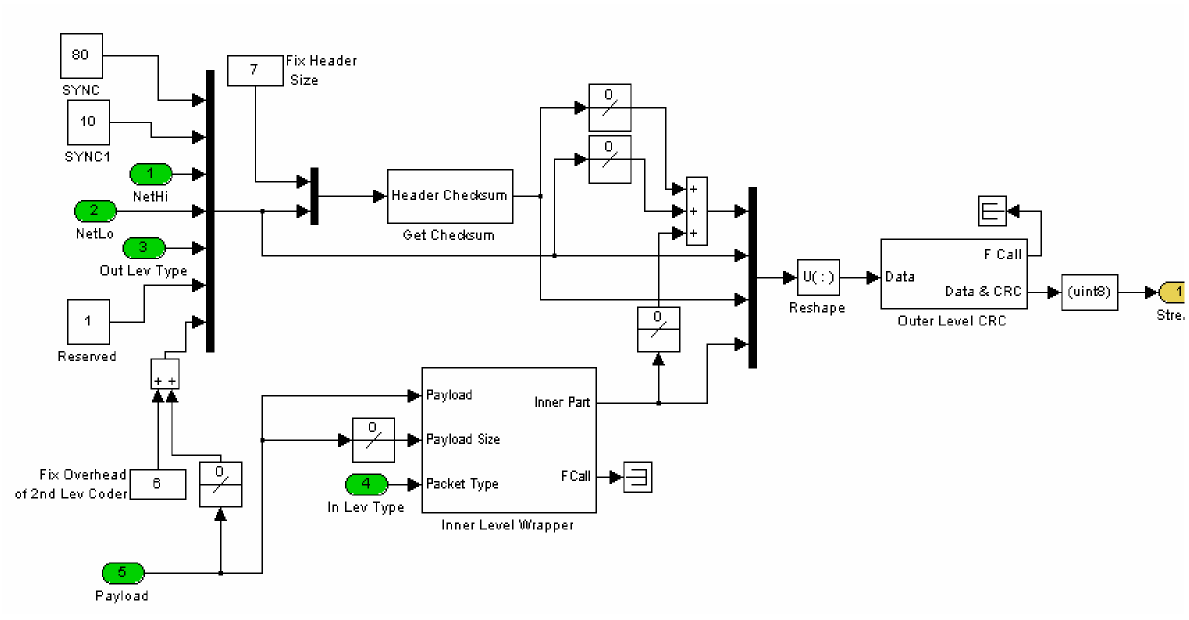
Once the autonomous landing algorithm has generated the commands that are required to transmit to the UAV, it is necessary to put those commands in a format that the ground station and, most importantly, the autopilot, can understand and execute.

In the same format as the one shown in Figure 6 in Chapter I, each set of commands is wrapped appropriately in the two-layer protocol. In order to compute the CRC of both levels and the checksum, three C S-Functions were coded and are included in Appendix A.

The complete assembly of the stream to send, shown in Figure 18, mimics the scheme shown in Figure 6 in Chapter I. First, the inner level wrapper receives the payload (the commands to send) and the data type, and it wraps it, appending the header and calculating the CRC (Figure 18a). Then the outer level header is assembled and the checksum is obtained. All the data is put together in the correct order and the outer level CRC is calculated, delivering a packet ready to be sent through the serial port (Figure 18b).



(a)



(b)

Figure 18. Encoder of the Piccolo Protocol; (a) Inner Level Wrapper; (b) Outer Level Wrapper.

4. RS-232 Writer of the Piccolo Protocol

Due to a limitation on the Piccolo Ground Station, the commands generated by the control algorithm could not be sent at the same operating frequency as the rest of the controller (40 Hz); therefore, a different sampling rate for the serial writer had to be implemented. So even though the algorithm used the serial port in full duplex mode, the reading and writing procedures were done at different sampling rates.

5. Limitations on the Data Rate

According to the Piccolo's Communications Protocol Specifications [4], the data rate at which the data is received is not fixed, and moreover, it is nondeterministic. Because of this, the data had to be estimated for an unknown period until new data was available.

After analyzing the messages that were required for the autonomous landing algorithm, it was noticed that the longest *drop outs* occurred in the GPS data embedded in the telemetry message. Thus the position of the UAV was not known at every sample time. Because of this, a filter was implemented to perform dead reckoning between fixes.

The filter shown in Figure 19 performs the fusion of two measurements when available, the position of the UAV and its velocity. From those measurements, it estimates the true position of the UAV. If the GPS position is updated, then it is considered in the filter. If not, then the position is estimated from the velocity.

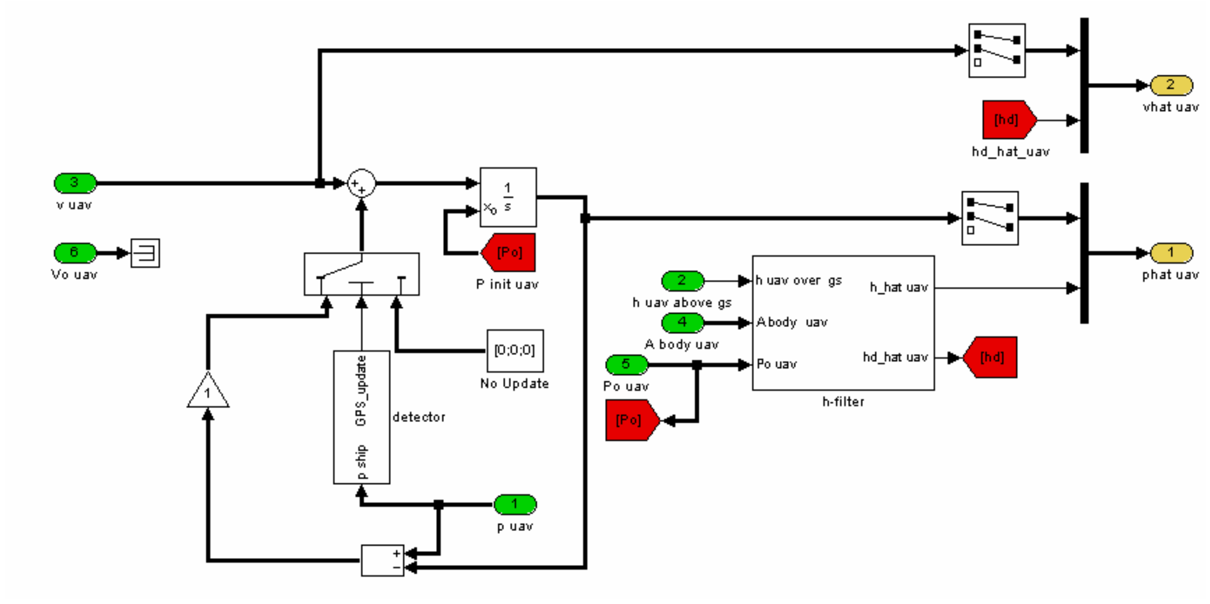


Figure 19. Position and Velocity Filter.

For the vertical channel, a complementary filter, shown in Figure 20 and denoted as *h-filter* in Figure 19, was implemented. This filter fuses the signal of the onboard accelerometers and the altitude measurements. A simple logic block detects whether there has been an update in the altitude readings or not and sets a flag accordingly. If there has been an update, then it is used to estimate the altitude and the vertical velocity of the airplane. If not, then those values are obtained directly from the accelerometers. The values for the coefficients of such filter were determined experimentally.

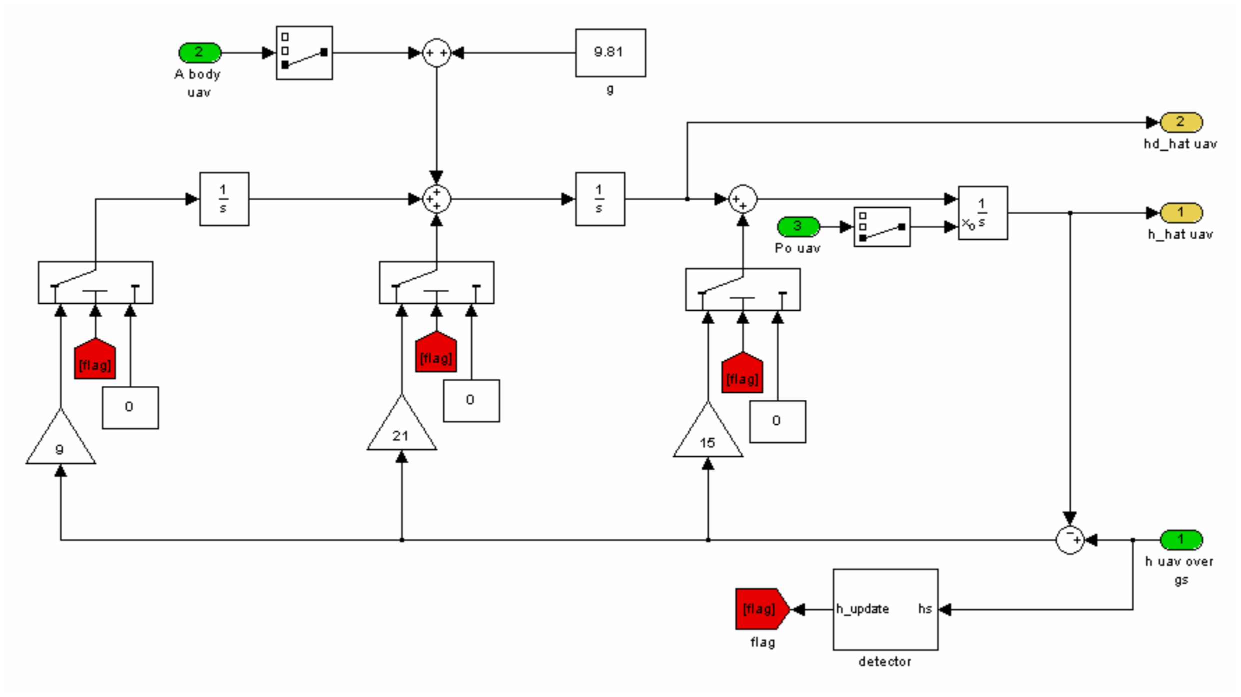


Figure 20. Complementary Filter in the Vertical Channel.

C. AUTONOMUS LANDING FLIGHT CONTROL COMPUTER

The Autonomous Landing Flight Control Computer, shown in Figure 21, is a PC104 form factor computer manufactured by Integrated Systems Inc., running a Pentium I microprocessor at 266 MHz. It contains a 64 Mb *disk-on-chip* memory drive that performs similar tasks to the hard drive in personal computers. The operating system is an IBM PC-DOS version 3.4 and on top of that runs the Mathworks' XPC Target kernel, which allows the implementation of models developed in Simulink [5] to run in real-time.

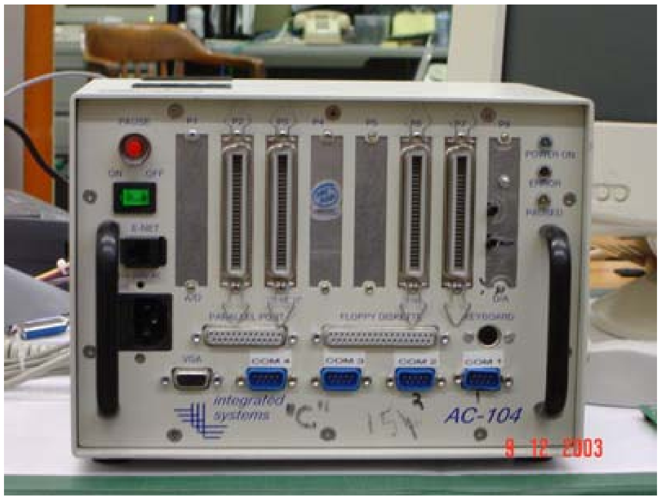


Figure 21. Autonomous Landing Flight Control Computer.

The Autonomous Landing Flight Control Computer has four serial ports that are used as follows:

- Serial port 1 (COM1) is used to receive the information from the Piccolo's Ground Station and to send the control commands; thus it is used in full duplex mode.
- Serial ports 2 and 3 (COM2 and COM3) receive information from two Trimble agGPS114 GPS units that provide the corner points of the recovering net (points $\vec{p}_1$ and $\vec{p}_2$ described in Section A of this chapter).
- Serial port 4 (COM4) receives information from a Garmin GBR21 differential correction receiver to provide differential correction for the GPS onboard the UAV.

The Ethernet port is used for two purposes. One is to communicate with the host computer, that is, the computer where all the models are compiled and from where they are downloaded to the *disk-on-chip*. A more detailed description of what the Host computer is and what functions it performs will be given in the following chapter when the implementation issues are discussed.

The second purpose of the Ethernet port is to provide communication with a dedicated personal computer (PC) on the network that performs data logging for all the required internal variables. Data logging is a necessary task in order to analyze the flight data offline and provide a way to debug any problems in the code. The logged data is sent from the Autonomous Landing Flight Control Computer using the User Datagram Protocol (UDP), which due to its connectionless nature, provides a very convenient way to send real-time data through a network.

Figure 22 shows the complete interconnection of the Autonomous Landing Flight Control Computer to all the peripheral devices mentioned above.

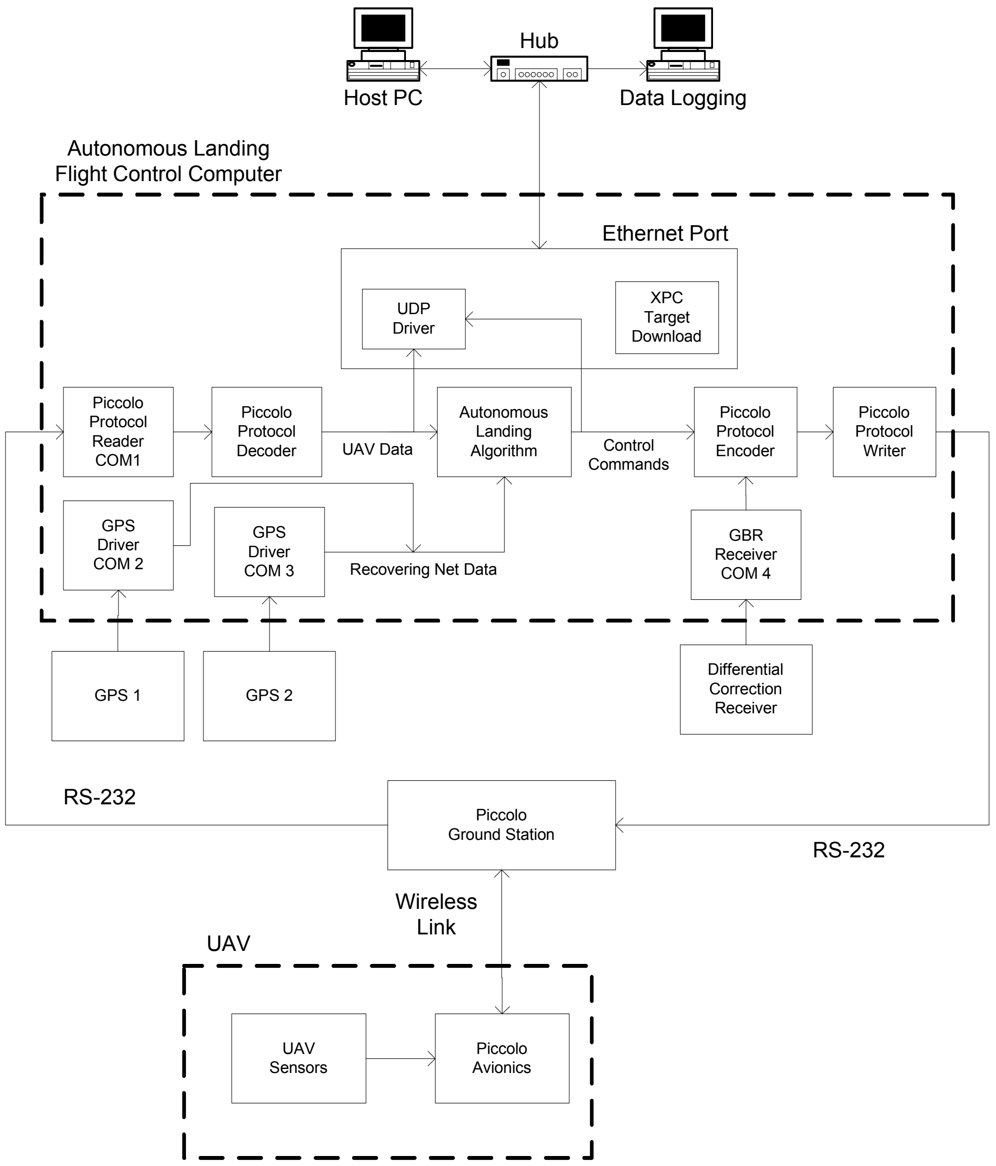


Figure 22. Complete Interconnection of the Autonomous Landing Flight Control Computer to Peripheral Devices.

D. DEVELOPMENT OF THE DEVICE DRIVERS

So far, it has been assumed that the position of the corner points of the recovering net (points $\vec{p}_1$ and $\vec{p}_2$ described in Section A of this chapter) are obtained using two GPS receivers, but no detail has been given on how this data is read from the GPS and made available to the autonomous landing algorithm. Since Simulink does not provide any readily available method to decode the serial GPS data, a set of custom made C S-Functions had to be developed.

For the GPS differential correction provided by the Garmin GBR21 and required by the Piccolo's Avionics GPS, the case was slightly easier. Since the data provided by the GBR21 was not necessary for the autonomous landing algorithm, just a reader and a protocol wrapper (similar to the ones described for the control commands) were created to pass trough the data to the Avionics.

Both device drivers are described in the following subsections.

1. GPS

The Trimble agGPS114 GPS sends out messages in the NMEA-0183 format. These messages consist of streams of comma-delimited ASCII text as shown in Figure 23. The NMEA message structure includes a message identifier at the beginning of each stream to identify unequivocally the type of message. Each NMEA message contains a different number of fields according to a predetermined structure and includes a checksum value, which is used to check the integrity of the data included in the message. The checksum on each stream is calculated by performing an 8-bit exclusive OR of the characters between the dollar sign and the asterisk delimiter.

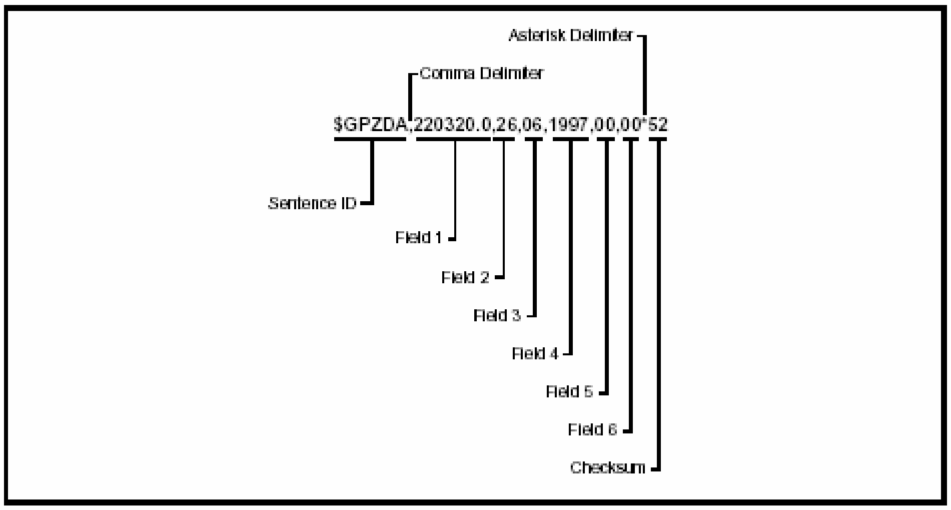
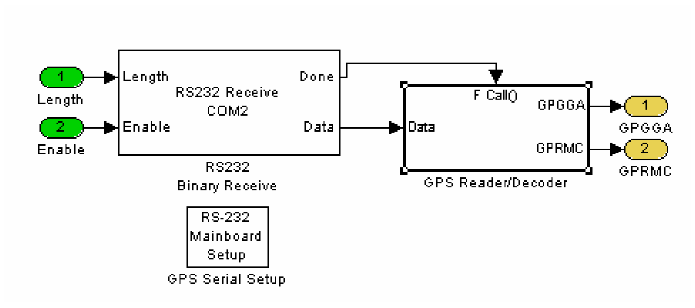


Figure 23. NMEA-0183 Message Structure (From: Ref. [6].)

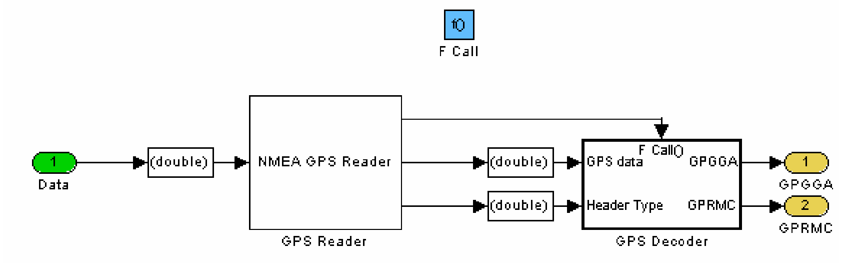
In the same way as the driver developed for the Piccolo protocol, this driver is separated in two parts—a reader that actually handles the serial port and strips the useful data from the headers and a decoder that decodes the received bits into meaningful measurements provided by the GPS messages.

Two of the sixteen NMEA messages provided by the agGPS114 were of particular interest because of the navigation data they provided: the GPS fixed data, which uses the identifier *GGA* in the NMEA format; and the recommended minimum specific GPS data, which uses the identifier *RMC* in the NMEA format. These two messages provided the necessary data for the autonomous landing algorithm.

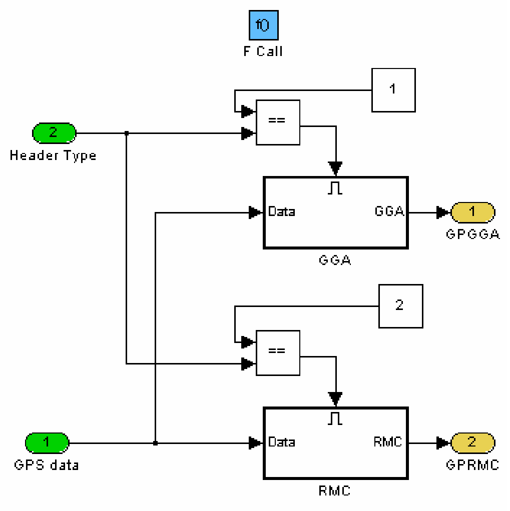
The complete reading/decoding scheme is shown in Figure 24. The raw data is received through the serial port and streamed to a buffer of predetermined size. Once the buffer is full, the serial receiver issues a function call that indicates to the following block that new data is available (Figure 24a). Once the reader receives a complete buffer, it identifies which type of NMEA message it is, removes the header and checksum, and sends to the output the NMEA message type, the data, and a function call that indicates to the decoder that new data is available (Figure 24b). The decoder, according to the type of message that will be decoded, enables the appropriate block that generates the meaningful data (Figure 24c). The C source code for the implemented S-Functions used to perform these tasks is included in Appendix A.



(a)



(b)



(c)

Figure 24. Complete Reading/Decoding of the GPS Data (a) Receiver, Reader, and Decoder. (b) Reader/Decoder. (c) Decoder.

2. Differential Correction Beacon

Geographical position data provided by the GPS is generally corrupted by four types of measurement errors: ionospheric and tropospheric noise, multipath effect, satellite position errors, and receiver noise[7]. In order to improve the quality of the position provided by the GPS, the signals are differentially corrected by signals emitted by radio beacons (supervised by the United States Coast Guard), which essentially contain an approximation of the values for each of the measurement errors in the area of the receiver. Applying differential corrections to GPS data can improve its accuracy by an order of magnitude or more [7].

In order to provide the differential corrections provided by the Garmin GBR21 through serial port 4 (COM4) to the Piccolo's Avionics, a protocol encoder and wrapper identical to the one shown in Figure 18 was implemented.

E. SUMMARY

This chapter discussed the tracking and capture segments of the landing approach in detail. It explained how the flight data is made available to the system by the Piccolo protocol reader/decoder developed in C and Simulink. It presented the Autonomous Landing Flight Control Computer hardware and discussed all the peripherals required by the algorithm. The next chapter presents the design, development and implementation of the control system.

III. CONTROL SYSTEM DESIGN

For the control system design presented in this chapter, a six-degrees-of-freedom (6-DOF) nonlinear model of the Silver Fox UAV was first developed in Simulink based on the available flight characteristics and structural data provided by the manufacturer. The autopilot was then designed using classical control techniques for the innermost, thus a faster, loop. For this purpose, the dynamic model of the Silver Fox UAV around the trimmed flight condition was linearized and used as the plant. Then the outer loop structure was developed to drive to zero the error signals generated by the autonomous landing segments described in Chapter II.

Once the real-time simulation showed successful results, the autopilot design was passed on to Cloud Cap Technologies, Inc. so it could be programmed and included in the Piccolo's Avionics firmware.

The last section of this chapter presents the integration of the autonomous landing algorithm developed in the previous chapter with the control system presented in this one.

A. CONTROLLER DESIGN

The controller design presented here is based on the classical inner/outer control loop approach. The inner loop is designed to be much faster in response than the outer loop. Thus, it produces enough bandwidth separation to make the plant look like a point mass to the outer loop. The outer loop is a regulator designed to drive the position and velocity errors (described in Chapter II Section A Subsection 3 and 4) to zero. In both cases, the longitudinal and lateral motion were decoupled by using several assumptions described in the following subsection.

A complete top-level block diagram of the inner and the outer control system used for the controller design is shown in Figure 25. This figure shows how the inner loop receives as inputs the angular rates and the speed of the UAV and compares them with the set points created by the outer loop to produce the control surface deflection and the thrust command. The outer loop monitors the velocity and position errors generated by the autonomous landing algorithm and generates the set points accordingly. A more detailed explanation of the inner loop controller (autopilot) and the outer loop controller (regulator) is given in the following subsections.

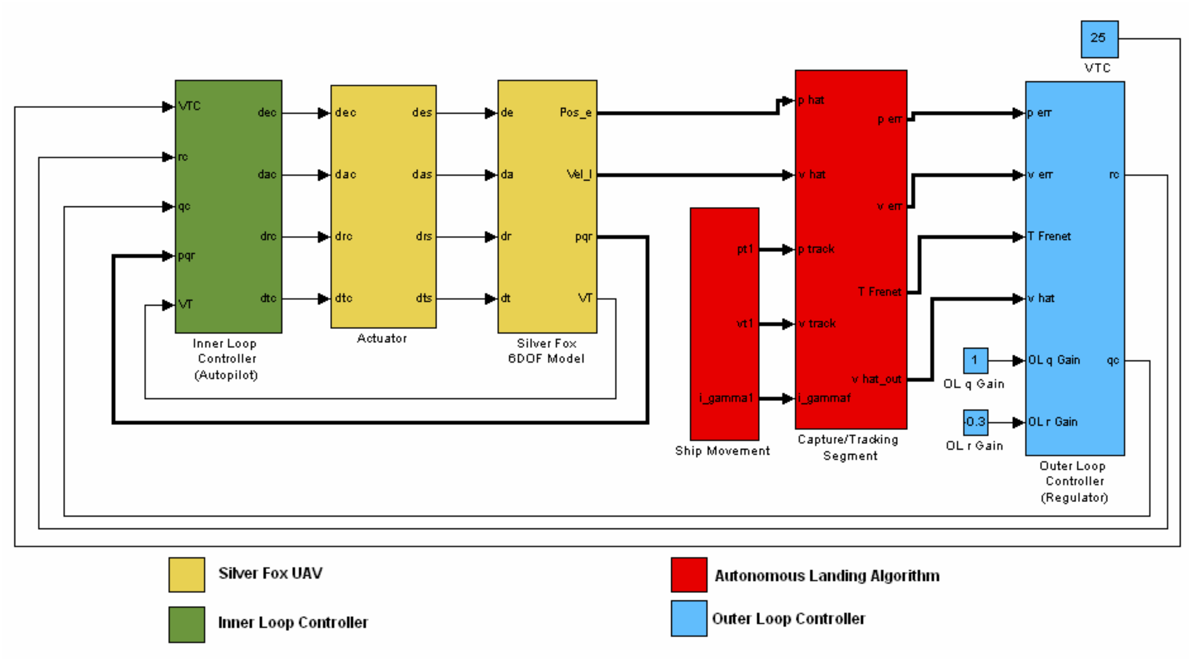


Figure 25. Top Level View of the Inner and Outer Loop Control System Design.

1. Decoupling of Motion

One of the commonly used techniques that simplify analysis of the spatial motion of an aircraft is the decoupling of its motion onto longitudinal and lateral components.

A study of equations (B8) and (B9), shown in Appendix B, demonstrate that two nonlinear vector equations are required to describe the spatial motion of an aircraft. Nevertheless these equations can be separated into two subsets.

Assume that the aircraft is in leveled flight, thus in trim condition, and that this condition is disturbed by the deflection of the aircraft’s elevator. The deflection generates a pitching moment only,without affecting the rolling or yawing moments. It also produces a change in the forces applied in the *X* and *Z* direction in the body frame *{B},* without affectiing the dynamics in the *Y* direction. In other words, we consider the dynamics in the longitudinal plane of the aircraft.

Therefore, the longitudinal motion is characterized by the following parameters taken in body frame *{B}* : ${}^{B}x, {}^{B}z, {}^{B}v_x, {}^{B}v_z,\ \alpha,\ \theta$, and *q,* which are defined in Appendix B. This motion is caused by the forces and moments acting in the local vertical plane of the aircraft: thrust (assumed that there is no deflection of this vector), drag, lift, gravity (projection to the local vertical plane obtained through pitch) and the pitch moment ${}^{B}G_y$.

Proceeding in the same way as for the longitudinal motion, but this time disturbing the trim flight by a deflection of the aileron or the rudder, the parameters that describe the lateral motion can be obtained. These parameters are ${}^{B}y, {}^{B}v_y,\ \beta,\ \varphi,\ \psi,\ p$, and *r*, again, defined in Appendix B. The lateral motion is produced by the projection of gravity and aerodynamic forces and moments in the sideslip direction.

It is also important to mention that such a separation allows decoupling of the control surfaces deflection onto two separate *channels*: elevator for the longitudinal channel and ailerons and rudder for the lateral channel.

2. Inner Loop Autopilot

a. Lateral Channel

The control in the lateral channel was designed to stabilize the aircraft's yawing and make the response fast enough so the lateral channel of the outer loop control *"saw"* only a point mass. As stated previously, two of the control surfaces, the aileron and the rudder, have influence over the aircraft's lateral motion. Therefore, two separate analyses had to be performed.

The first step was to obtain a linearized model of the Silver Fox, which for the control purposes was the plant. The Silver Fox 6DOF model described in the first section of this chapter was linearized about a trim condition in leveled flight at 200 meters altitude with an airspeed of 21 meters per second. The linearization was accomplished by using Simulink's linear analysis tool. The following figure shows the Root Locus and Bode plot of the transfer function obtained from the deflection of the aileron (δ_a) to the yaw rate (*r*).

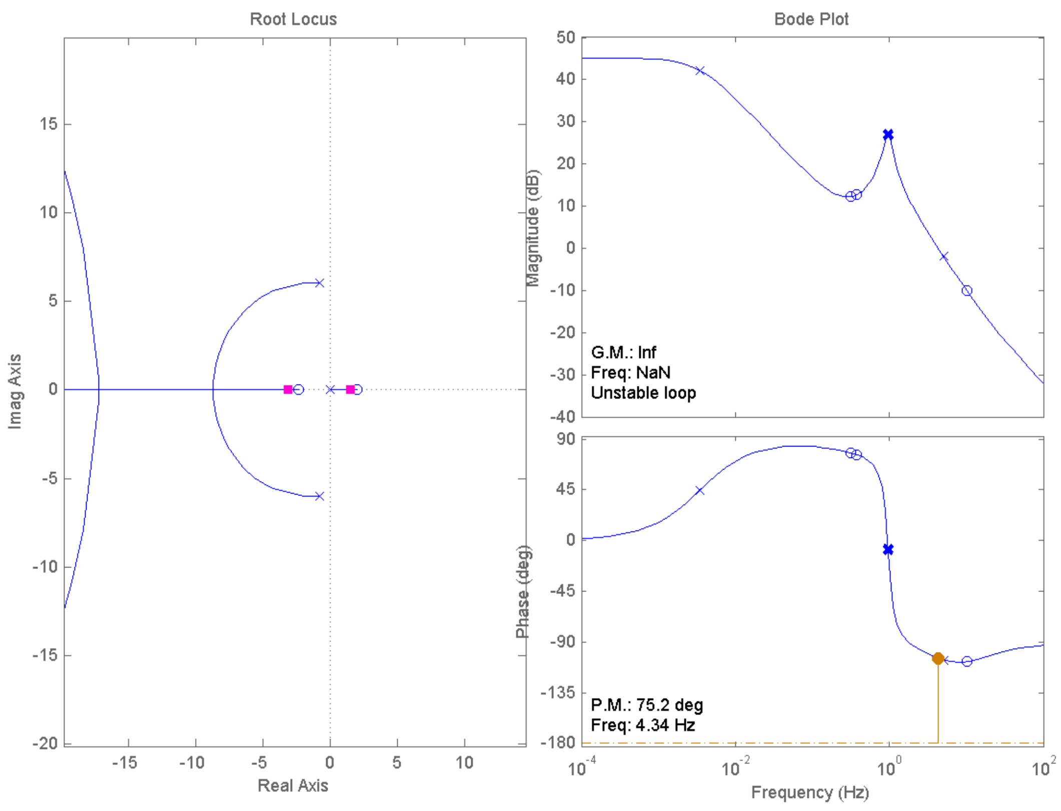


Figure 26. Root Locus and Bode Plot of the Transfer Function from δ_a to *r.*

A simple analysis of the above figure shows that using only a feedback gain will not result in a stable system. This is because, in order to bring the pole located at $1.48+0j$ to the left side of the imaginary axis, a feedback gain would not be sufficient. To be able to bring the unstable pole to the stable region quickly, a compensator with a zero located at $2+0j$, a pole located at $-2+0j$, and a gain of 0.1 was introduced in the forward control path. This resulted in a system with a root locus shown in Figure 27, which is stable for the selected gain, with the dominant poles located at $-2.48 \pm 4.39j$ and $-1.05 \pm 0.216j$. The selection of this controller results in a closed loop system with a gain margin of 7.76 dB and a phase margin of 60.7°, also shown in Figure 27.

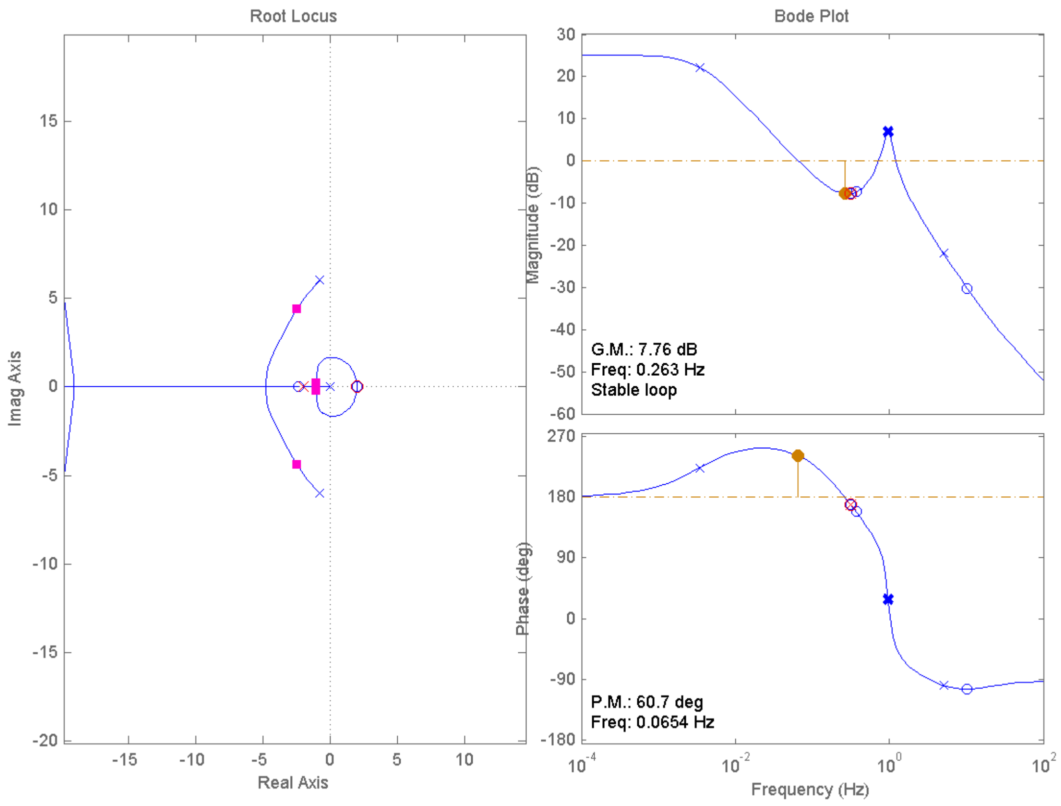


Figure 27. Root Locus and Bode Plot of the Transfer Function from δ_a to r with a Compensator in the Forward Path.

Using the same procedure as the one described above for the aileron, a compensator for the rudder was implemented, with a zero at the origin, a pole at $-2+0j$, and a gain of 0.1. The root locus of the transfer function from δ_r to r with the compensator in the forward path is shown in Figure 28.

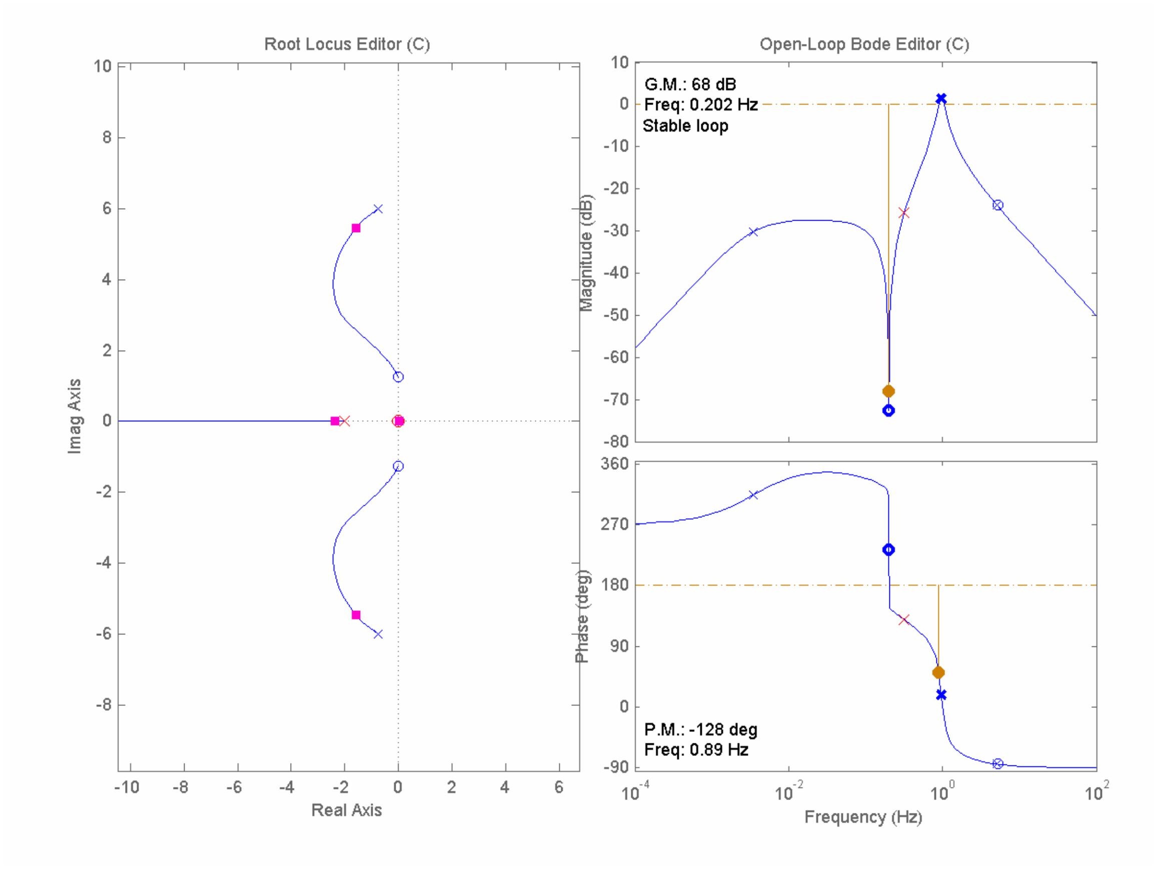


Figure 28. Root Locus and Bode Plot of the Transfer Function from δ_r to r with a Compensator in the Forward Path.

As shown in the above figure, choosing this compensator places the dominant closed loop poles at the origin and at $-1.59 \pm 5.45j$. This controller produces a closed loop system with a gain margin of 68 dB and a phase margin of -128^o.

b. Longitudinal Channel

The control in the longitudinal channel was designed to stabilize the aircraft's pitch and make the response fast enough so the longitudinal channel of the outer loop control *"saw"* only a point mass. As stated before, the elevator is the only control surface that has influence over the aircraft's longitudinal motion. The commanded thrust to the aircraft also has an important influence in the longitudinal channel, since an increment in thrust is reflected in an increment in speed and this generates an increment in lift, thus making the aircraft gain altitude.

Therefore, two compensators were designed for the longitudinal channel, one to stabilize the aircraft's pitch and have a fast response, and one to keep the speed of the aircraft constant, thus avoiding the variation of the influence of the air speed in the longitudinal channel.

Analyzing the root locus of the transfer function from δ_e to q, shown in Figure 29, it was noticed that a pure gain compensator was enough to make the system stable and to have the desired response. Therefore, a gain of 0.05 was chosen to be placed in the forward control path.

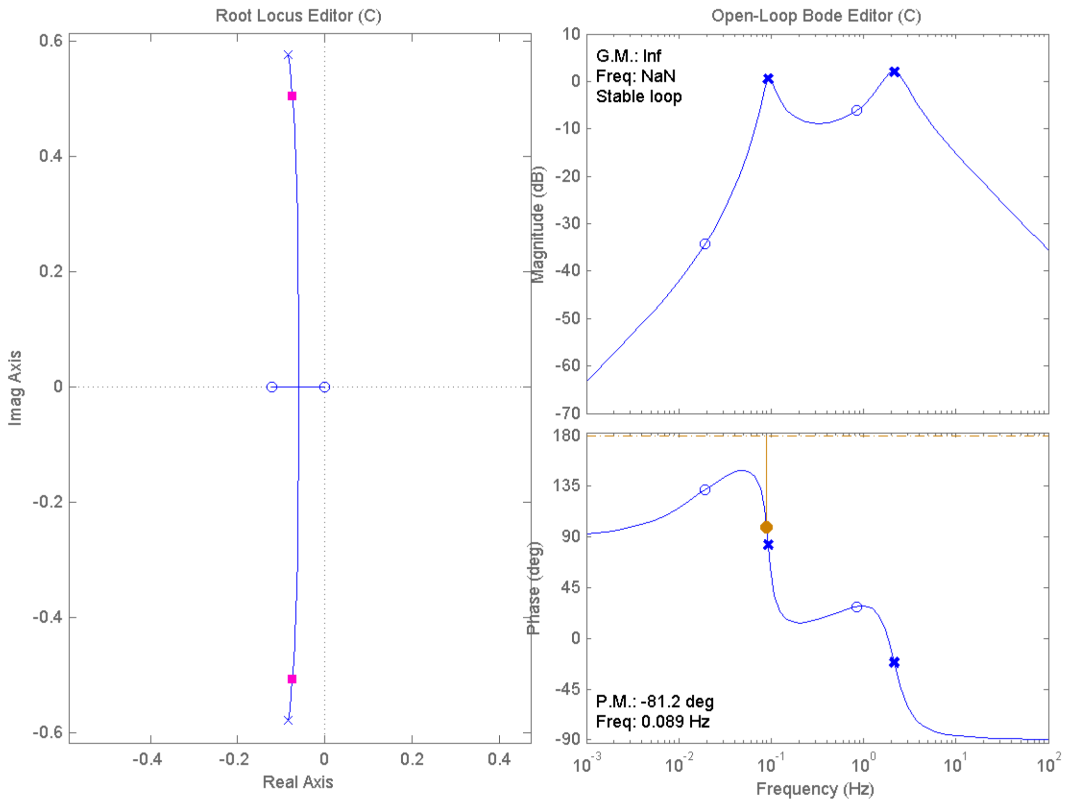


Figure 29. Root Locus and Bode Plot of the Transfer Function from δ_e to q with a Gain Compensator in the Forward Path.

As a result of this gain the dominant closed loop poles were located at $-0.0744 \pm 0.506j$. The system had a infinite gain margin and -81.2^{o} of phase margin.

For the speed controller, a Proportional-Integral (PI) controller was implemented to keep the speed constant. The gains implemented for the controller were determined experimentally. Figure 30 shows the root locus and bode plot of the closed loop speed controller implemented.

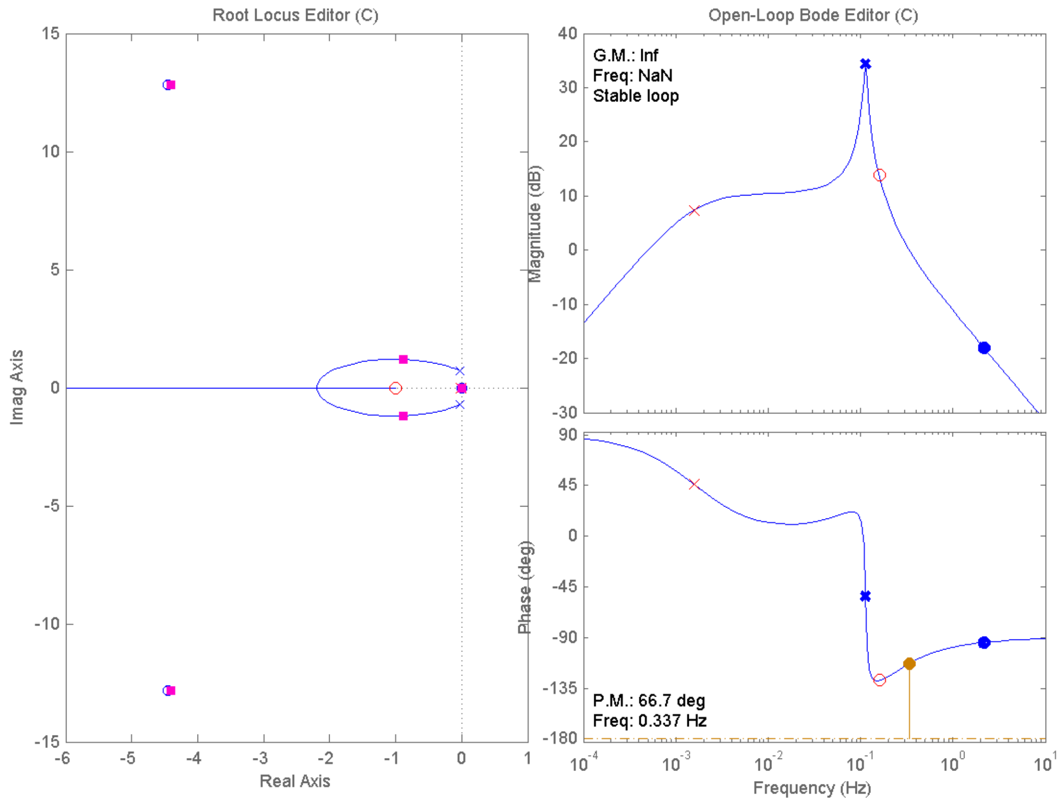


Figure 30. Root Locus of the Closed Loop Speed Controller.

As a result of placing the PI controller in the forward path, the dominant closed loop poles where located at $-0.00231 \pm 0.0j$ and at $-0.879 \pm 1.91j$. This resulted in an infinite gain margin and a 66.7^{o} phase margin.

The complete implementation in Simulink of the lateral and longitudinal control previously discussed is shown in Figure 31. As it can be seen, besides implementing the control scheme previously discussed, limiters have been added to each of the control signals to avoid generating signals that are not physically possible.

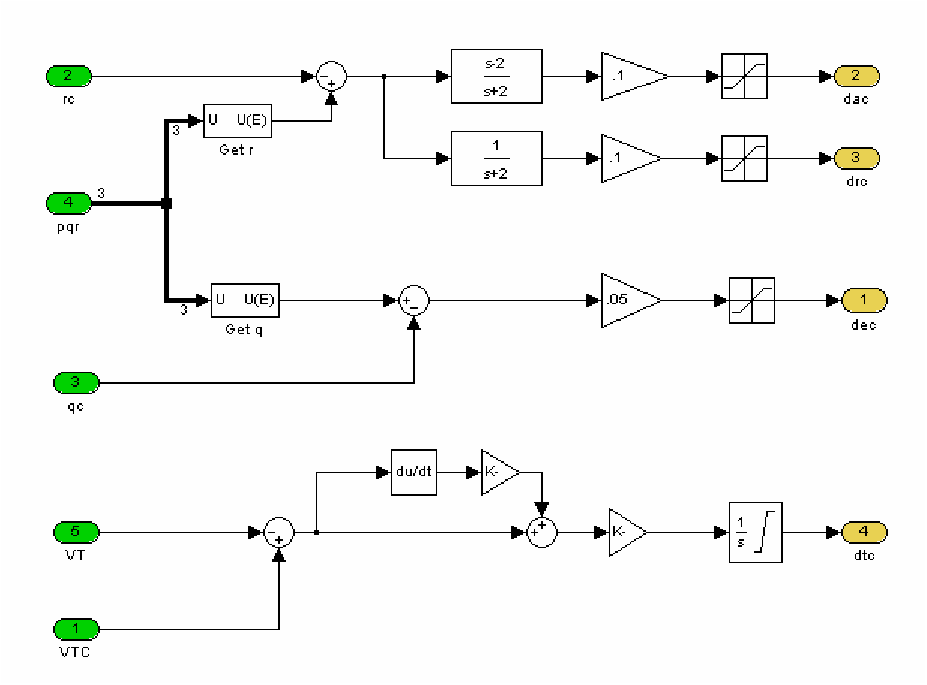


Figure 31. Inner Loop Control (Autopilot) Implementation in Simulink.

Table 1 shows a summary of the location of the dominant closed loop poles, the gain margins, and phase margins for each channel of the inner loop control.

		DOMINANT CLOSED LOOP POLES	GAIN MARGIN	PHASE MARGIN
Lateral Channel	*Aileron*	$-2.48 \pm 4.39j$ $-1.05 \pm 0.216j$	7.76 dB	60.7^o
	Rudder	$0.0 + 0.0j$ $-1.59 \pm 5.45j$	68 dB	-128^o
Longitudinal Channel	*Elevator*	$-0.0744 \pm 0.506j$	Infinite	-81.2^o
	Thrust	$-0.00231 \pm 0.0j$ $-0.879 \pm 1.91j$	Infinite	66.7^o

Table 1. Location of the Dominant Closed Loop Poles, Gain Margins and Phase Margins of the Inner Loop (Autopilot) Controller.

3. Outer Loop Control Feedback

As described above, in control terms, the outer loop is a regulator. For this particular case, the signals to be driven to zero are the position and velocity errors generated by the tracking or capture segments of the autonomous landing algorithm. Since both position and velocity errors were readily available, this resulted in a convenient way to implement a PD control. During the initial phase of the control system design, it was decided that, in order to provide a better control scheme, feedback controllers should be introduced in the feedback path.

In the same way as for the inner loop, the complete plant, including the inner loop, was linearized using Simulink's Linear Analysis tool. The controllers were designed from the linear model obtained.

Figure 32 shows a top level view of how the control strategy was implemented for the outer loop. Note that the plant transfer functions (shown in green) are the transfer functions obtained from the linearization of the Silver Fox 6-DOF model, the autopilot and the capture/tracking segment (shown in Figure 25 in yellow, green, and red, respectively) altogether.

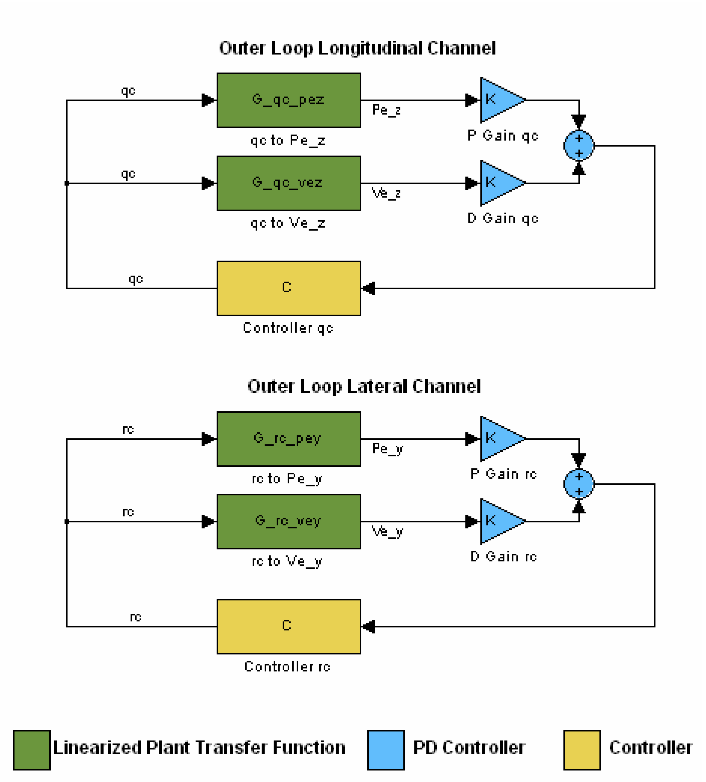


Figure 32. Top Level Diagram of the Control Scheme Implemented in the Outer Loop.

a. Longitudinal and Lateral Channels

The design of the controller in this case was an iterative process. The PD gains were first set to make the system marginally stable. Then the feedback controllers were designed. Finally, the PD gains were fine-tuned in simulation to obtain the desired response.

The following figure shows the Root Locus of the outer loop's longitudinal channel with just the PD controller. It can be seen that the system, even with a very small feedback gain, can become unstable due to the fact that the pole located at the origin will move to the right-hand side of the plane, thus making the system unstable. To correct for that, a controller was introduced in the feedback path bringing the system to stability and reshaping the Root Locus to the one shown in Figure 34.

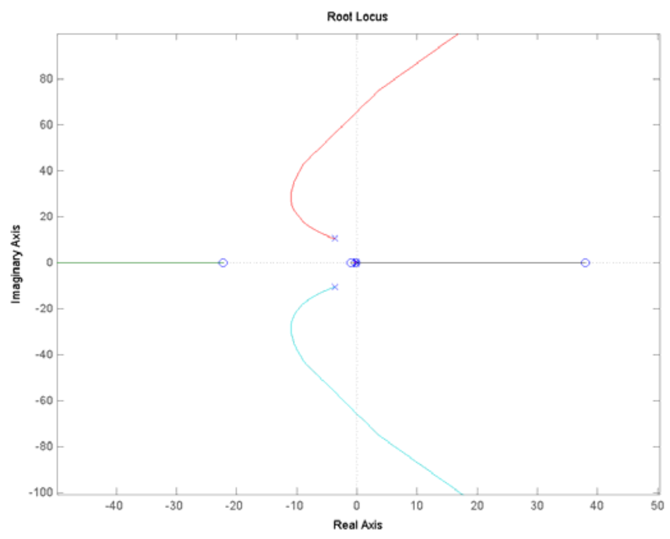


Figure 33. Root Locus of the Outer Loop Longitudinal Channel Using Only PD Control.

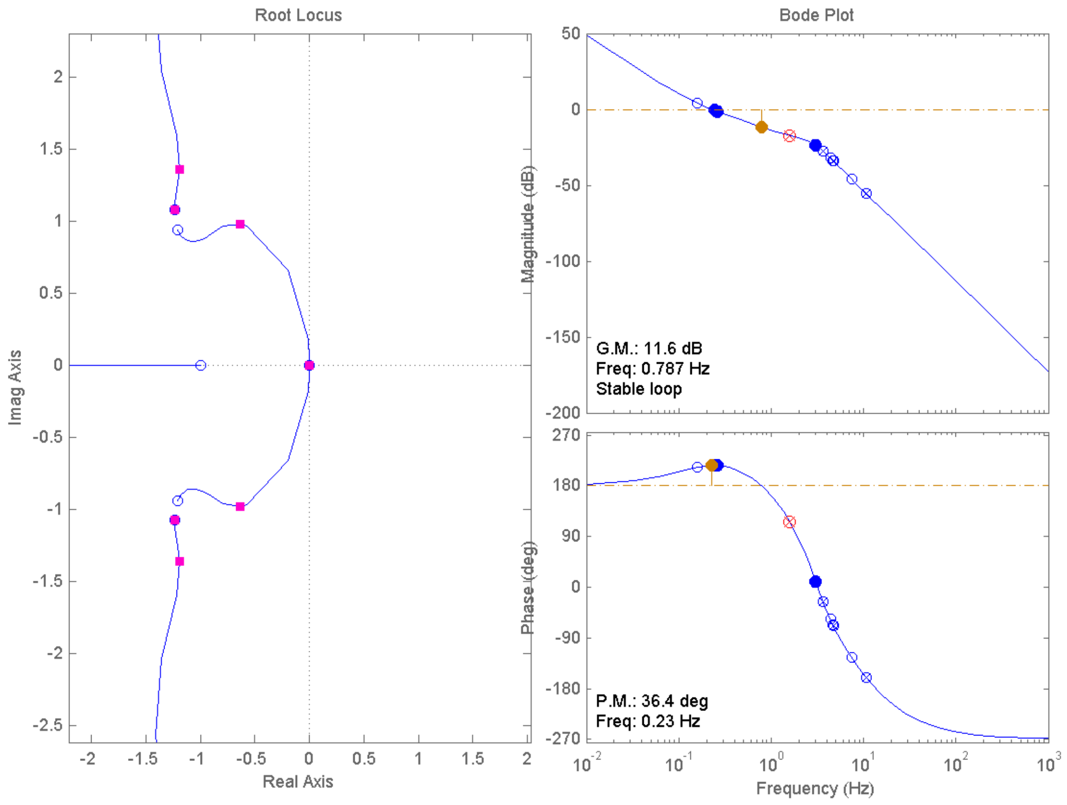


Figure 34. Root Locus and Bode Plot of the Outer Loop Longitudinal Channel Using PD Control and a Controller in the Feedback Path.

This controller placed the dominant closed loop poles at $0.0+0.0j$, $-0.634\pm0.981j$, $-1.19\pm1.36j$, and at $-1.24\pm1.08j$. The resulting gain margin was 11.6 dB and a 36.4^o phase margin.

Using the same methodology as the one described above for the longitudinal channel, the lateral channel was designed and fine-tuned in simulation. Figure 35 shows the Root Locus and Bode plot of the longitudinal channel.

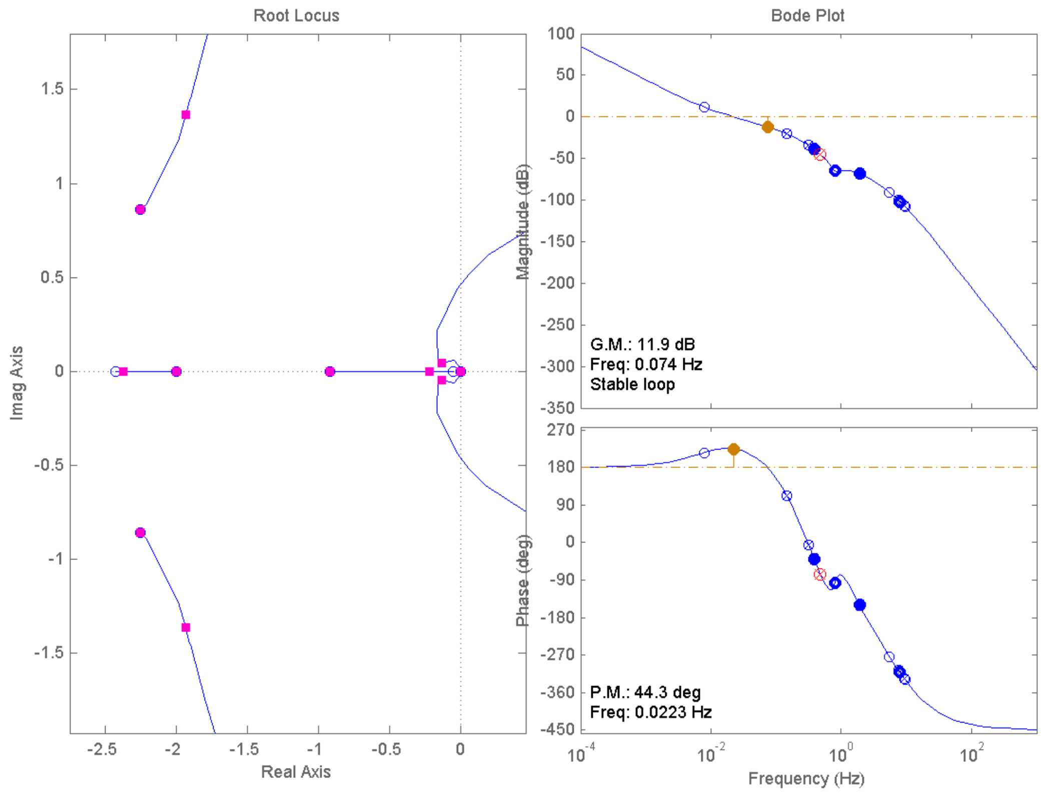


Figure 35. Root Locus and Bode Plot of the Outer Loop Lateral Channel Using PD Control and a Controller in the Feedback Path.

This controller placed the dominant closed loop poles at $0.0+0.0j$, $-0.133\pm0.0451j$, $-0.218+0.0j$, $-0.919+0.0j$. The resulting gain margin was 11.9 dB and a phase margin of 44.3^o.

The complete outer loop controller is shown in Figure 36. Note that a *realign* block has been introduced in the lateral channel. This block will be explained in the following subsection.

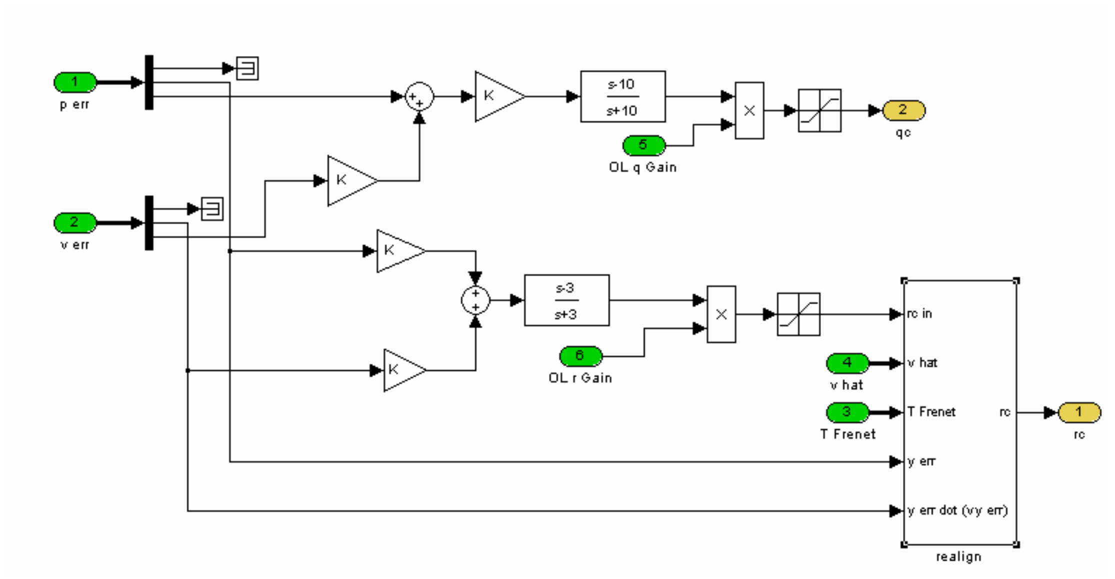


Figure 36. Simulink Implementation of the Complete Outer Loop Control Feedback .

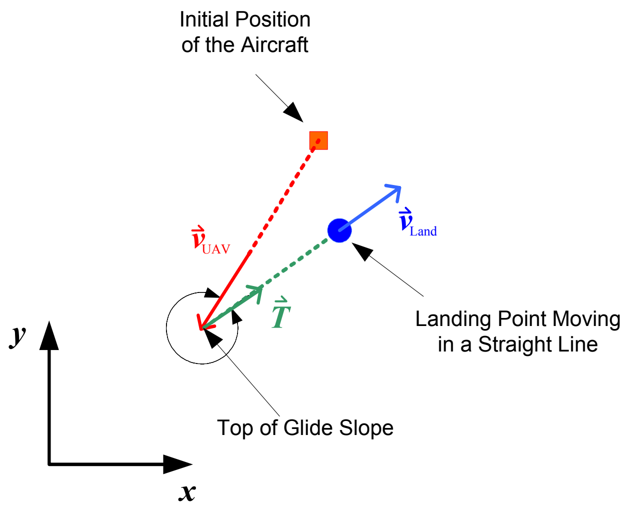
Table 2 shows a summary of the location of the dominant closed loop poles, the gain margins and phase margins for each channel of the outer loop control.

CHANNEL	DOMINANT CLOSED LOOP POLES	GAIN MARGIN	PHASE MARGIN
Longitudinal	$0.0+0.0j$ $-0.634\pm0.981j$ $-1.19\pm1.36j$ $-1.24\pm1.08j$	11.6 dB	36.4°
Lateral	$0.0+0.0j$ $-0.133\pm0.0451j$ $-0.218+0.0j$ $-0.919+0.0j$	11.9 dB	44.3°

Table 2. Location of the Dominant Closed Loop Poles, Gain Margins and Phase Margins of the Outer Loop Control.

b. Path Realignment

Since the system was designed to take control of the UAV at any in-flight initial condition, a possibility exists that when the angle between the UAV's velocity vector and $\vec{T}$ in the XY plane is noticeably large, it can impact the performance of the algorithm (see Figure 37).



Initial Position of the Aircraft
$\vec{v}_{UAV}$
$\vec{v}_{Land}$
$\vec{T}$
Landing Point Moving in a Straight Line
Top of Glide Slope
y
x

Figure 37. Scenario Showing a Large Angle Difference between the UAV's Velocity Vector and $\vec{T}$ in the XY Plane.

It was noticed in the simulation that under this condition the algorithm might diverge or have slow convergence. To avoid that, an additional logic in the lateral channel was introduced in the outer loop control to force the UAV to realign its velocity vector with the glide slope in the XY plane much faster than it would normally do, by commanding a high constant yaw rate. In control terms, this was done to force the control algorithm to *slide* into a convergence region.

This logic calculates the angle between these two vectors and decides whether to let the calculated yaw rate be passed through, or to command a high constant yaw rate. If so, the position error and velocity error are used to decide whether the high constant yaw rate should be positive or negative.

The implementation of the path realignment block is shown in Figure 38.

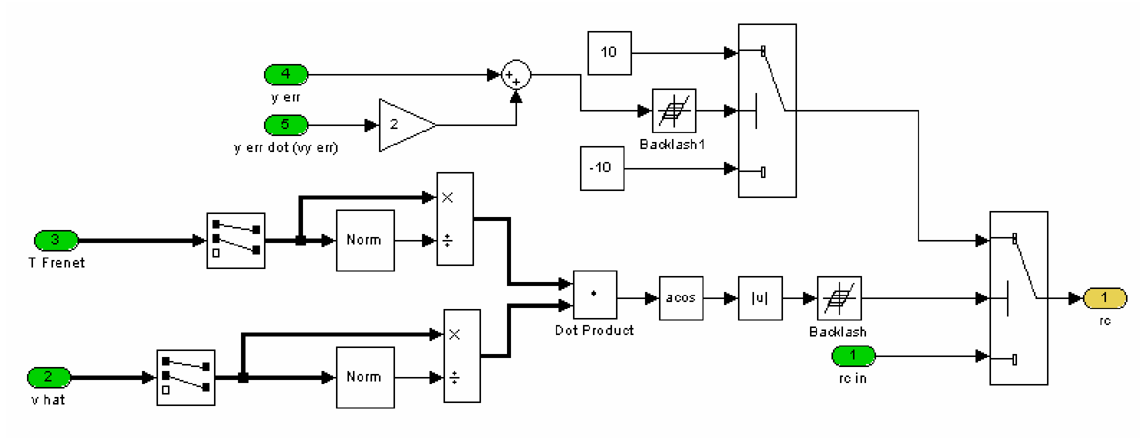


Figure 38. Simulink Implementation of the Path Realignment Logic.

B. INNER LOOP AUTOPILOT IMPLEMENTATION IN THE UAV AVIONICS

As previously stated, the inner loop autopilot must provide a fast response by feeding back the information from the onboard sensors. However, if the autopilot designed were to reside on the ground station, then the communication delays would significantly impact its performance. To avoid this, the NPS workgroup and Cloud Cap Technologies, the developers of the Piccolo system, agreed that the NPS autopilot needed to reside in the Piccolo Avionics as well.

Cloud Cap, in coordination with the NPS workgroup, modified the proposed autopilot shown in Figure 31, to include anti-windup logic to the integrators, channel shut-off capabilities, and configurable values for the gains and limits of the saturation blocks. Also, additional logic was included to permit the ground control to decide which autopilot was active, NPS's or Piccolo's. A top-level diagram of the C code implemented in the avionics microprocessor is shown in the following figure.

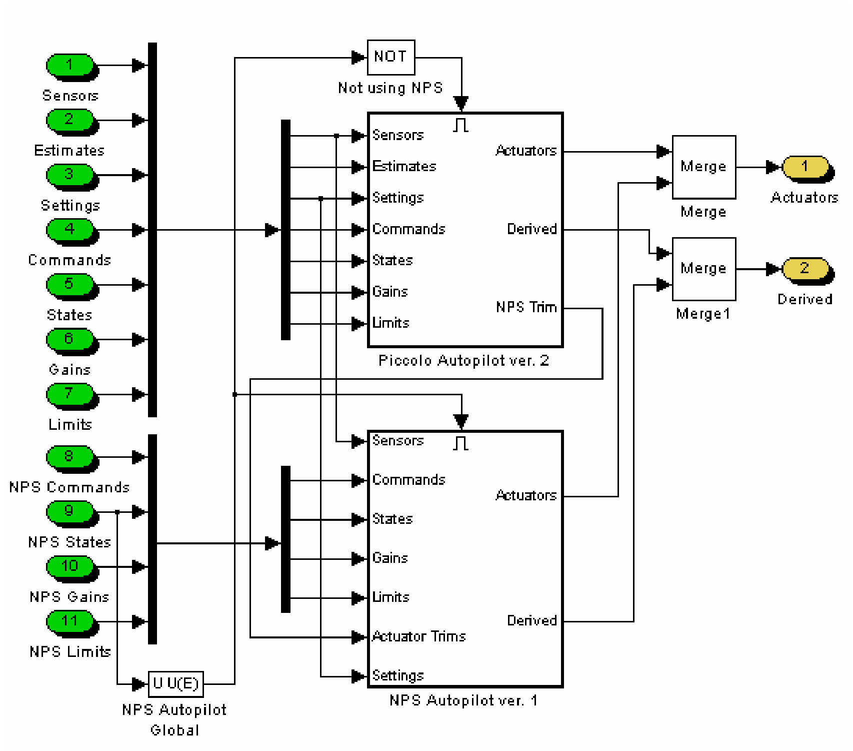


Figure 39. NPS and Piccolo Autopilot Implementation in the Avionics.

It is important to note that as seen in Figure 39 this new control scheme introduces several changes. The input named *Settings* and the output named *Derived* contained data such as altitude, air density, and true air speed, that were relevant to the Ground Station. Cloud Cap Technologies implemented these directly.

In addition to the previously implemented commands received by the autopilot (pitch rate command and yaw rate command), three new messages were implemented. Two of these messages (*NPS Gains* and *NPS Limits*) configure the numerical values of the gains and saturation limits from the ground. A third message (*NPS States*) shuts down the lateral and/or longitudinal channels mainly for debugging purposes. This third message also indicates which of the two autopilots is active.

The last change introduced as a result of the autopilot implementation in C was the *NPS Trim* signal, which was an output of the Piccolo autopilot and an input to the NPS autopilot. This signal was created to set all the initial conditions of the NPS autopilot so when the transition occurred there was no instantaneous change of the position of the control surfaces, thus permitting a smooth transition from normal flight to autonomous landing.

C. COMPLETE AUTONOMUS LANDING SYSTEM INTEGRATION

The Mathworks Inc., the developer of MATLAB, provides a set of tools that can deploy a system developed in Simulink to any PC running a real-time operating system (RTOS) that they develop. Two products (besides Simulink and MATLAB) are necessary to make this happen. One is the *Real Time Workshop*, which automatically generates the C source code and the object file to be downloaded to the microprocessor. The other is the *XPC Target,* which enables the connection and download of the C code generated by the *Real Time Workshop* to a physical system and executes it in real-time.

Therefore, all the Simulink models, like the ones shown in Figure 12, Figure 13, and Figure 15, were ready to be applied to a real-world scenario. The only remaining task was to perform the integration of all the elements previously described into one fully working system. The integration was performed as it is shown in Figure 40. The letter and number sequences included in most of the components of the system shown in the figure below, indicate the Chapter – Section – Subsection of this work that present and discuss how the component works. For example, the *Piccolo Protocol Reader/Decoder* is discussed in Chapter II Section B Subsections 1 and 2.

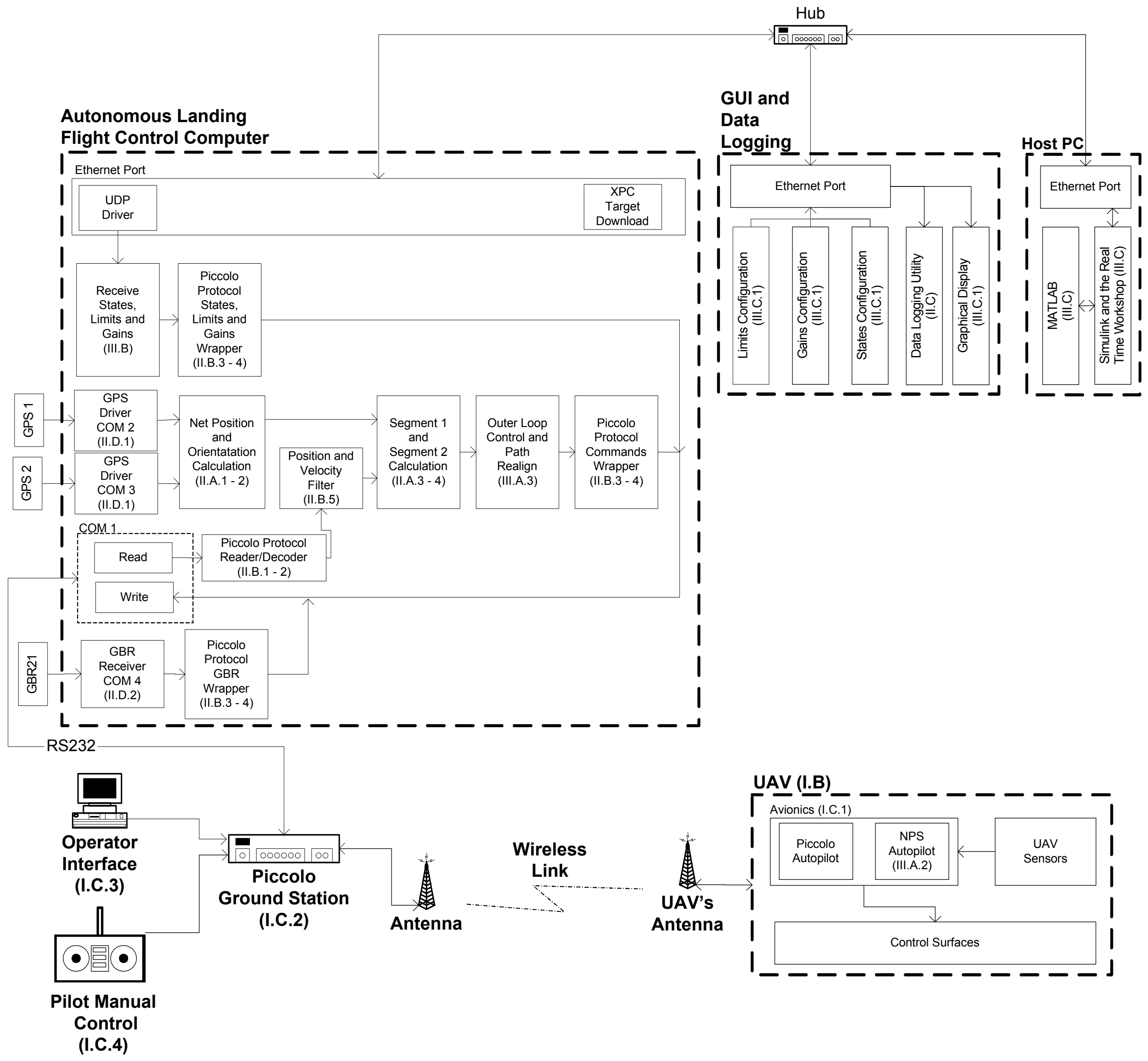

Hub
Autonomous Landing Flight Control Computer
Ethernet Port
UDP Driver
XPC Target Download
Receive States, Limits and Gains (III.B)
Piccolo Protocol States, Limits and Gains Wrapper (II.B.3 - 4)
GPS 1
GPS 2
GPS Driver COM 2 (II.D.1)
GPS Driver COM 3 (II.D.1)
Net Position and Orientatation Calculation (II.A.1 - 2)
Position and Velocity Filter (II.B.5)
Segment 1 and Segment 2 Calculation (II.A.3 - 4)
Outer Loop Control and Path Realign (III.A.3)
Piccolo Protocol Commands Wrapper (II.B.3 - 4)
COM 1
Read
Write
Piccolo Protocol Reader/Decoder (II.B.1 - 2)
GBR21
GBR Receiver COM 4 (II.D.2)
Piccolo Protocol GBR Wrapper (II.B.3 - 4)
RS232
GUI and Data Logging
Ethernet Port
Limits Configuration (III.C.1)
Gains Configuration (III.C.1)
States Configuration (III.C.1)
Data Logging Utility (II.C)
Graphical Display (III.C.1)
Host PC
Ethernet Port
MATLAB (III.C)
Simulink and the Real Time Workshop (III.C)
Operator Interface (I.C.3)
Piccolo Ground Station (I.C.2)
Antenna
Wireless Link
UAV's Antenna
Pilot Manual Control (I.C.4)
UAV (I.B)
Avionics (I.C.1)
Piccolo Autopilot
NPS Autopilot (III.A.2)
UAV Sensors
Control Surfaces

Figure 40. Top Level View of the System Integration.

A more detailed explanation of the GUI is given in the following subsection.

1. Graphical User Interface (GUI)

To configure all the values required for the three new message structures introduced by the implementation of the autopilot in C (States, Limits and Gains) a Graphical User Interface (GUI) had to be developed. The GUI also provided a convenient way to present in-flight data to the user in real time.

The GUI and data logging was performed by a laptop computer running Windows ® XP Pro, MATLAB and Simulink. As previously mentioned, the GUI provided a way to tune all the parameters of the autopilot, and it also provided a graphical display of the important variables of the system. All the data were exchanged through the network using UDP protocol. The following figure shows the GUI developed in Simulink for a flight test.

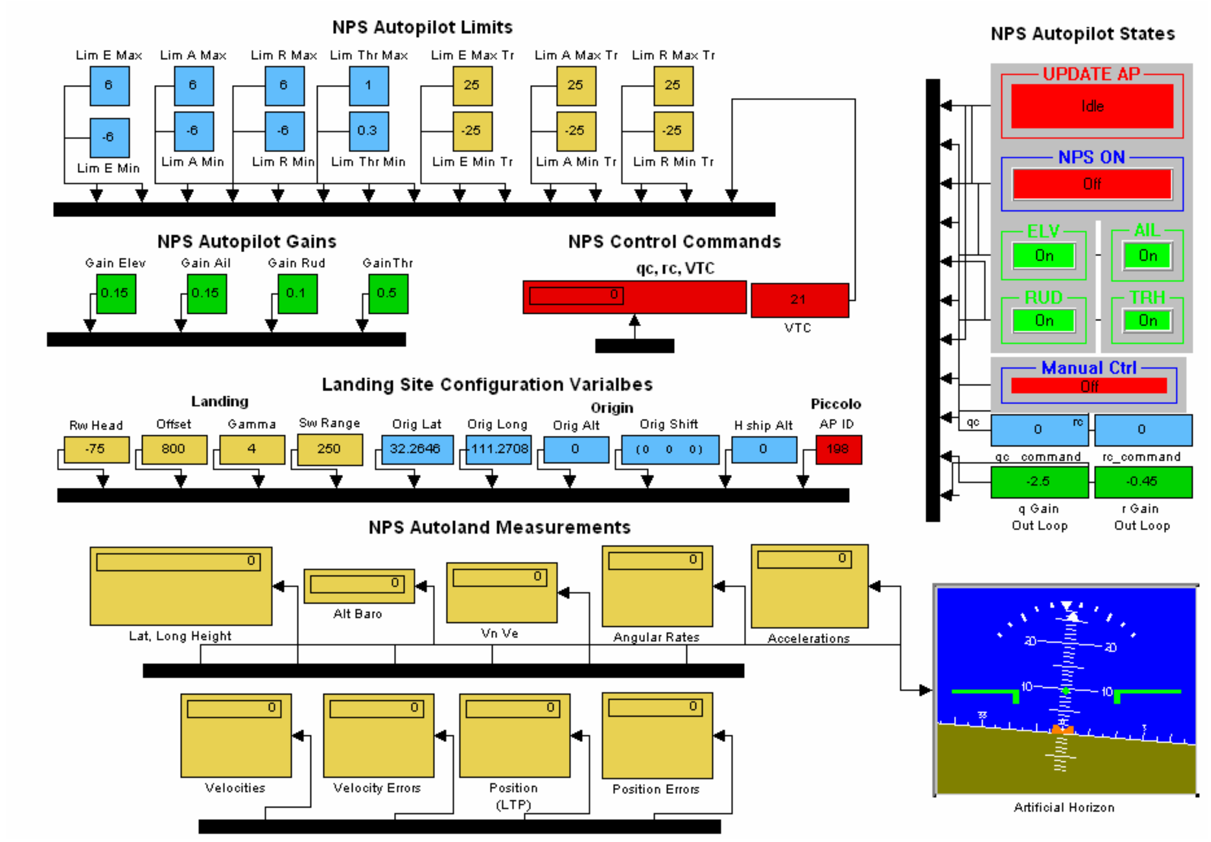


Figure 41. Graphical User Interface (GUI) Used to Configure the Autopilot Parameters, Present In-Flight Data and Log the Variables.

D. SUMMARY

This chapter presented how the linearized Silver Fox 6-DOF model was used as the plant model to design the control system using classical control techniques based on the inner and outer loop approach. Also, system integration and implementation issues were addressed. The following chapter presents the results obtained from the computer and hardware in the loop (HIL) simulations as well as the ground tests performed on the system.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. SIMULATION AND GROUND TESTS RESULTS

This chapter presents the results of extensive computer simulations performed to verify the feasibility of the solution proposed. Results from computer simulation, hardware in the loop, and ground tests are presented.

A. SIMULATION

The first approach used to explore the success rate of the algorithm proposed was to create a model in Simulink that contained the dynamics of the UAV represented by the 6DOF Silver Fox model, a very simple model of a ship underway with constant heading, and the implementation of the autonomous landing algorithm. More than 900 different runs were done for the simulation, varying the initial conditions of the aircraft, the existence of measurement noise and wind.

Following these successful initial results, the model was then simplified to land the UAV on a landing strip, in order to be ready for flight tests. This was done because the first phase of the testing protocol contemplated making a landing attempt at a landing strip in Tucson, Arizona.

1. Simulation Scenarios

Figure 42 shows a top-level view of the Simulink model used to simulate the autonomous landing on a ship underway. The movement of a ship was simulated by angular rates in roll and pitch of 0.15 rad/sec and 0.5 rad/sec respectively. The roll and pitch was considered to be 0.5 rad for both cases. The yaw was kept constant, thus having a constant heading. Additional blocks were introduced to add measurement noise when the scenario so required.

Figure 43 shows a top-level view of the Simulink model used to perform the simulation of the autonomous landing on a landing strip on ground. Several modifications from the previous diagram were introduced. The autopilot was now positioned next to the actuator model for clarity sake. Configurable parameters were added to set the angle of the glide slope (γ), offset (D), and the switching range. Also a way to configure the heading of the landing strip was introduced.

Since at this time the landing point would not be moving, the origin of the local reference frame *{L}* would be considered as the landing point.

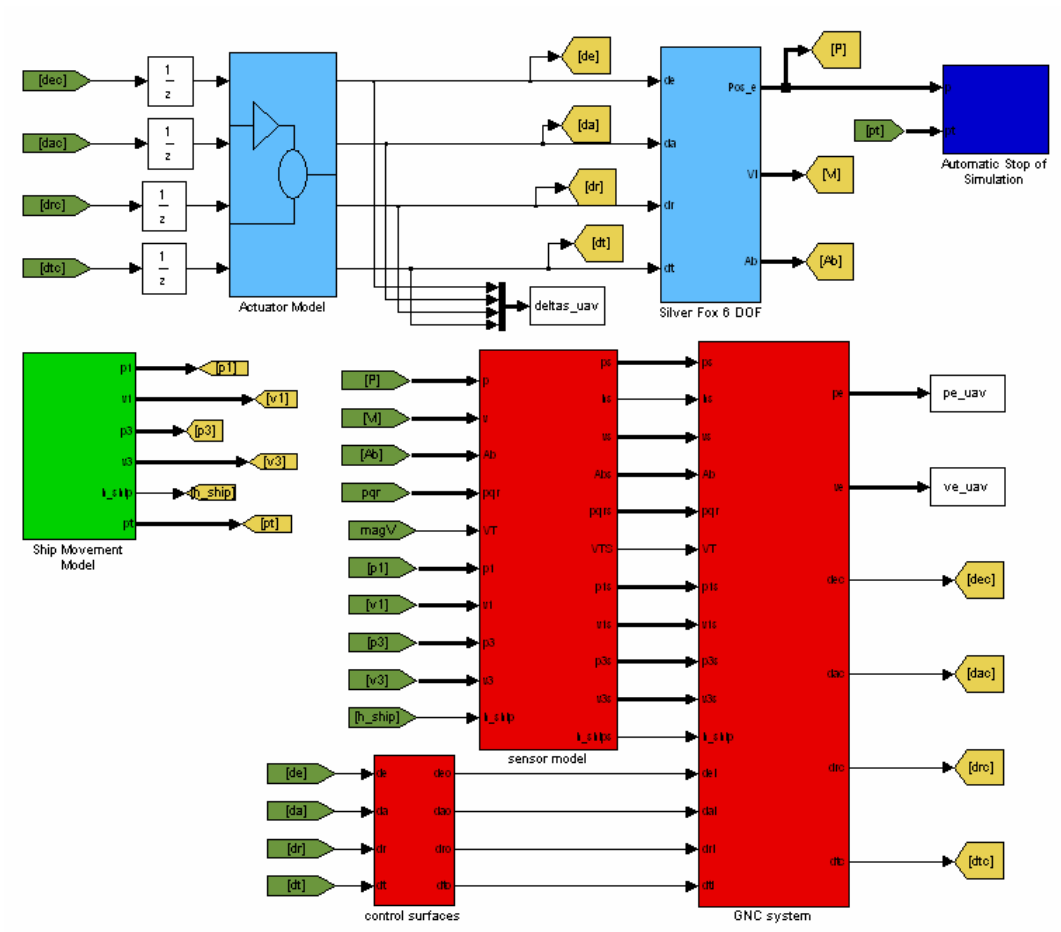


Figure 42. Simulink Model Used for Ship Landing Simulation.

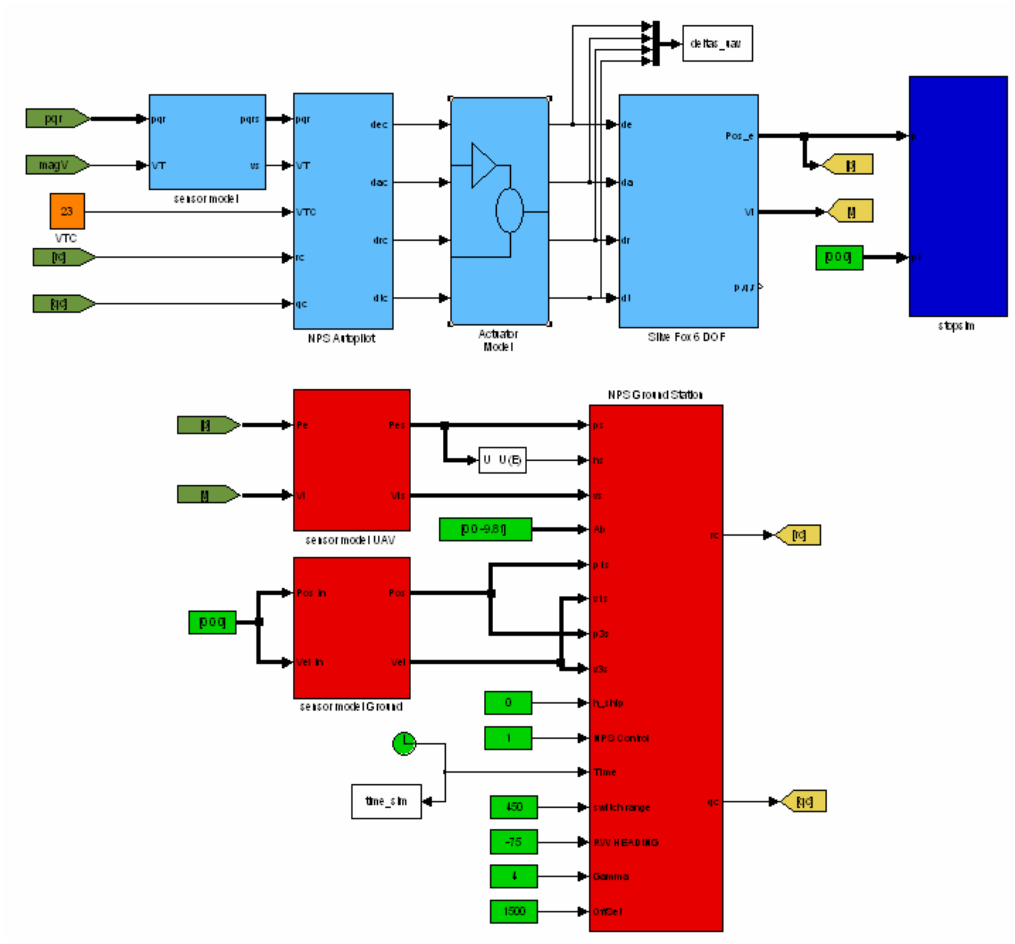


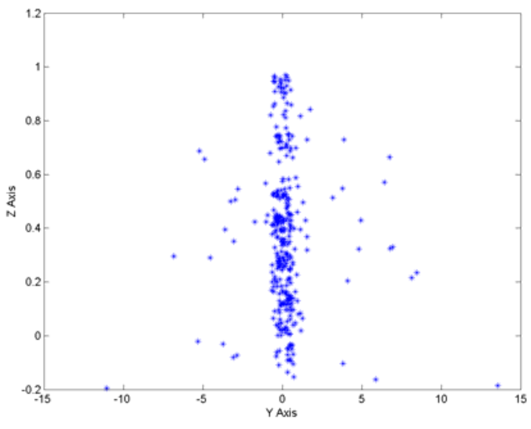
Figure 43. Simulink Model Used for Runway Landing Simulation.

Three major runs of the simulations were performed. The first run tested ideal conditions with no wind and the measurements being perfect, thus there was no measurement noise. The second run still assumed there was no wind disturbances but considered measurement noise. The last scenario, the most realistic, considered wind disturbances and measurement noise. The results are shown in the following subsections.

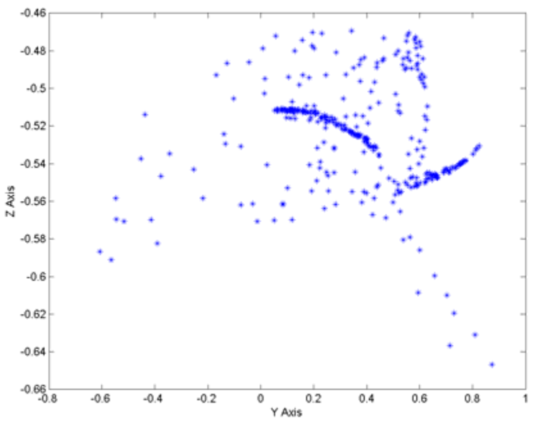
For the three cases described above the initial heading of the UAV was varied from 0 to 345 degrees in increments of 15 degrees. The initial position of the UAV was varied from -750 to 750 in increments of 250, in *X* and *Y* coordinates. For the ship landing, the initial height was varied from 75 to 125 meters in increments of 25. For runway landing, the initial height was varied from 100 to 150 meters in increments of 25.

a. *Ideal Conditions: No Measurement Noise and No Wind*

Figure 44 shows the position of the center of gravity (CG) of the UAV with respect to the middle point of the net when the UAV crosses the *YZ* plane of the net.



(a)



(b)

Figure 44. *YZ* Plot of the Position of the UAV's CG When Crossing the Plane of the Recovering Net. (a) Shipboard Landing. (b) Runway Landing. No Measurement Noise and No Wind.

As seen in the above figure, the error of the algorithm, under ideal conditions is generally quite small. For the 324 experiments performed under these conditions for each type of landing (shipboard and runway) the mean and standard deviations of the errors were calculated and are shown in the following table:

LANDING TYPE	**MEAN (m)**		**STANDARD DEVIATION (m)**	
	Y	*Z*	*Y*	*Z*
Shipboard	0.1950	0.3689	1.8166	0.2794
Runway	0.3533	–0.5254	0.2753	0.0294

Table 3. Means and Standard Deviations (in meters) of the Landing Errors. No Measurement Noise and No Wind.

From the previous table and figure, it can be seen that even though the runway landing has higher values for the means of the errors, its standard deviations are almost a magnitude smaller than those of the shipboard case. This, potentially, means that a smaller net is required for recovering the UAV on the ground.

The following figures show a complete landing approach. Figure 45 shows the three-dimensional plot and Figure 46 shows the *XY* and *YZ* plots of the approaches. It is clear that ideal conditions allow the autonomous landing algorithm to have smooth trajectories and very small errors during the capture and tracking segments. The glide slope is only shown in the runway landing scenario because in shipboard landing, the glide slope moves with the ship. Therefore plotting only the last glide slope would have no meaning and plotting all of them would only clutter the figure.

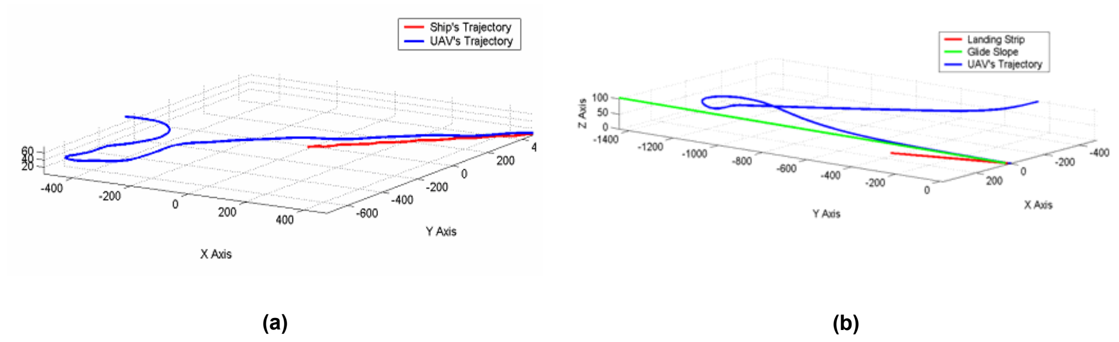
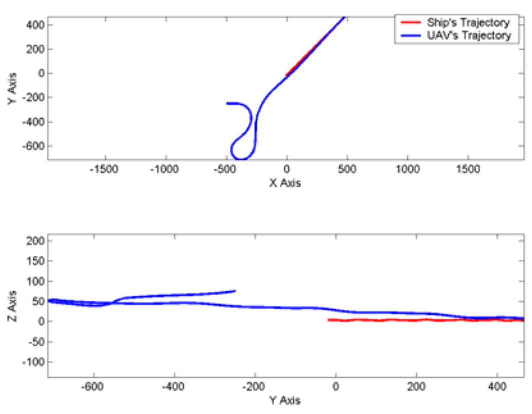


Figure 45. Three-Dimensional Plot of a Landing Approach. (a) Shipboard Landing. (b) Runway Landing. No Measurement Noise and No Wind.



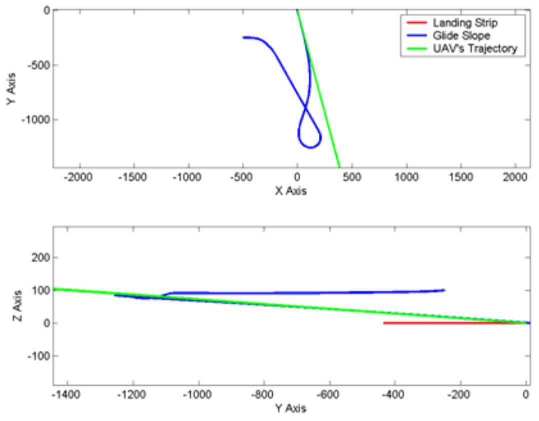
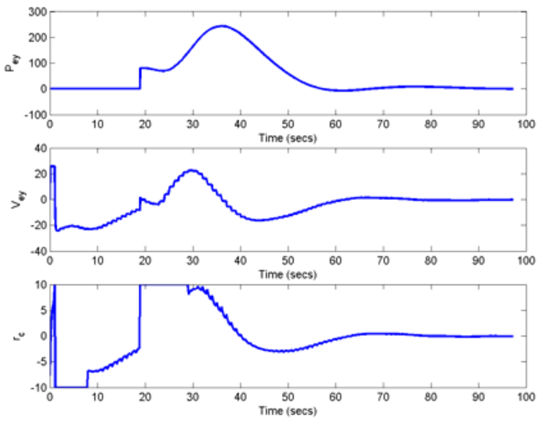


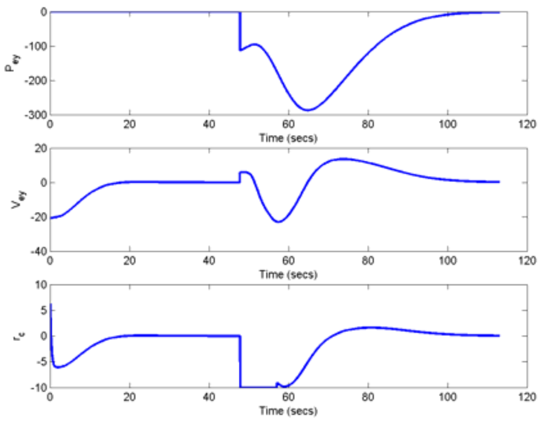
Figure 46. *XY* and *YZ* Plots of a Landing Approach. (a) Shipboard Landing. (b) Runway Landing. No Measurement Noise and No Wind.

From the above figures it is clear how, for the runway landing case, the UAV performs a perfect tracking of the glide slope as desired.

The following figures show how in the landing approaches shown above, first the UAV starts in capture mode and the position errors are zero. Once the UAV reaches the top of the glide slope, it switches to tracking mode and the position errors are no longer zero. After that, both errors are driven to zero by the outer loop control system. This is consistent with what was shown in Chapter II Section A. It is important to note how in the longitudinal channel, for the shipboard landing scenario, when the ship moves, the control system is always trying to compensate for that movement. Figures 47 and 48 show how the commanded pitch (q_c) and yaw (r_c) rates act to correct the position and velocity errors.

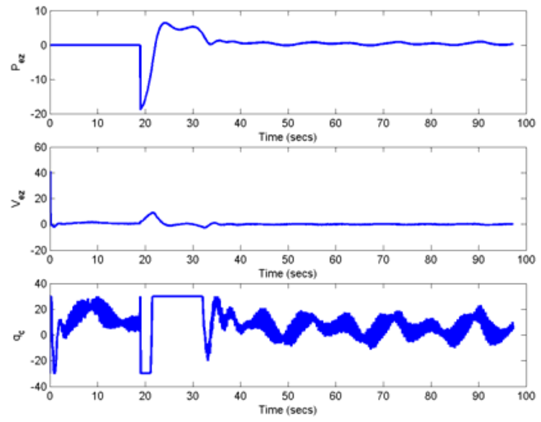


(a)

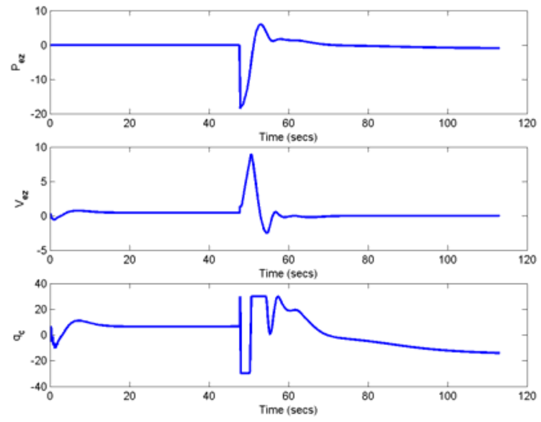


(b)

Figure 47. Position and Velocity Errors in Y Compared with the Commanded Yaw Rate. (a) Shipboard Landing. (b) Runway Landing. No Measurement Noise and No Wind.



(a)



(b)

Figure 48. Position and Velocity Errors in Z Compared with the Commanded Pitch Rate. (a) Shipboard Landing. (b) Runway Landing. No Measurement Noise and No Wind.

b. Measurement Noise and No Wind

The same set of experiments as in the previous subsection were performed but now measurement noise was added to the values *generated* by the Ship's and UAV's GPS. For all the cases additive white noise was assumed to be present. The following table shows the variances used for the different sensors.

SENSOR	READING	VARIANCE
UAV	Latitude and Longitude	3 m
	Height	1 m
	Velocities	0.5 m/s
Ship or Ground	Latitude, Longitude and Height	1 m
	Velocities	0.3 m/s

Table 4. Variances for the White Additive Noise Used in the Sensors for Simulation.

As expected, Figure 48 shows that introducing measurement noise negatively impacts the performance of the algorithm. It is important to note how the points on the plot are no longer clustered in a small regions and that the spread of these has noticeably increased.

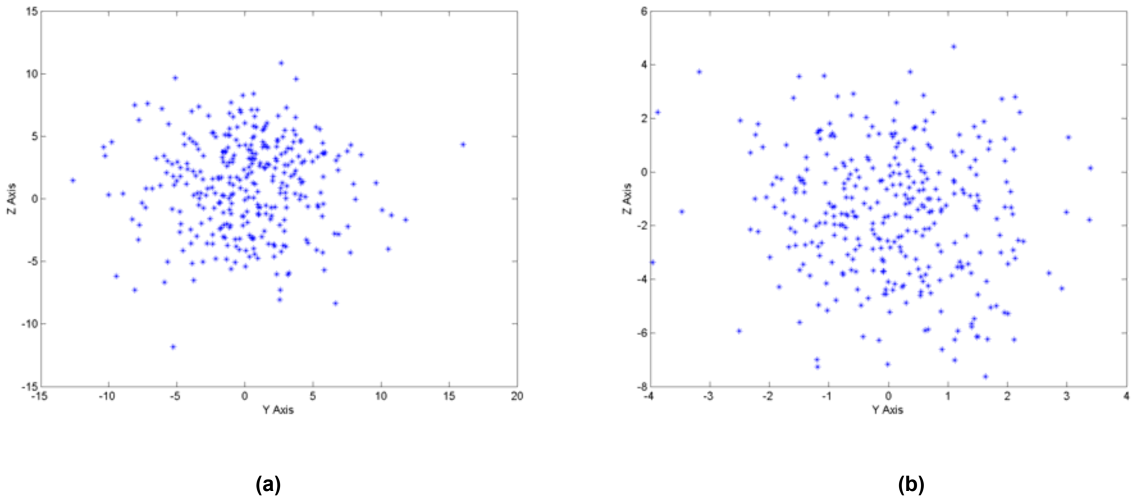


Figure 49. *YZ* Plot of the Position of the UAV's CG When Crossing the Plane of the Recovering Net. (a) Shipboard Landing. (b) Runway Landing. Measurement Noise and No Wind.

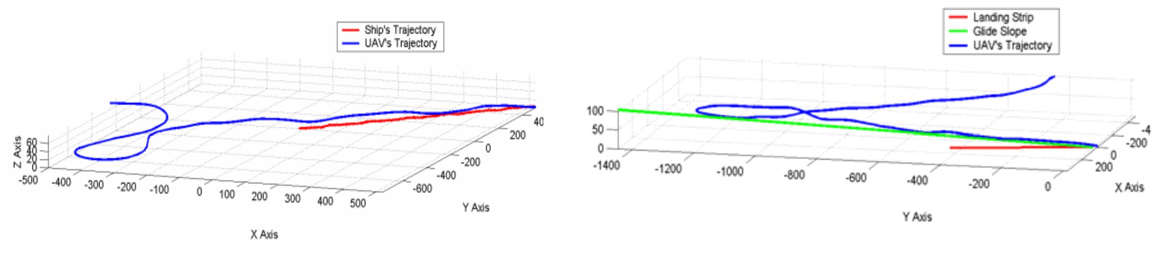
Table 5 shows the comparison of the means and standard deviations of the errors shown above.

LANDING TYPE	MEAN (m)		STANDARD DEVIATION (m)	
	Y	*Z*	*Y*	*Z*
Shipboard	0.1255	1.0930	4.1122	3.7121
Runway	0.0578	–1.5822	1.2455	2.3719

Table 5. Means and Standard Deviations (in meters) of the Landing Errors. Measurement Noise and No Wind.

As expected, due to the constant movement of the ship, the shipboard landing has larger standard deviations for the errors due to the uncertainty of the movement and the sensors.

The following figures show complete landing approaches for both scenarios. Figure 50 shows three-dimensional plots and Figure 51 shows the *XY* and *YZ* plots of the approaches. It can be seen that due to the uncertainty in the sensors, the UAV must perform a more complicated maneuver to position itself at the top of the glide slope. Also it is noticeable in the runway landing, that perfect tracking of the glide slope is no longer achievable.



(a) (b)

Figure 50. Three-Dimensional Plot of a Landing Approach. Measurement Noise and No Wind. (a) Shipboard Landing. (b) Runway Landing. Measurement Noise and No Wind.

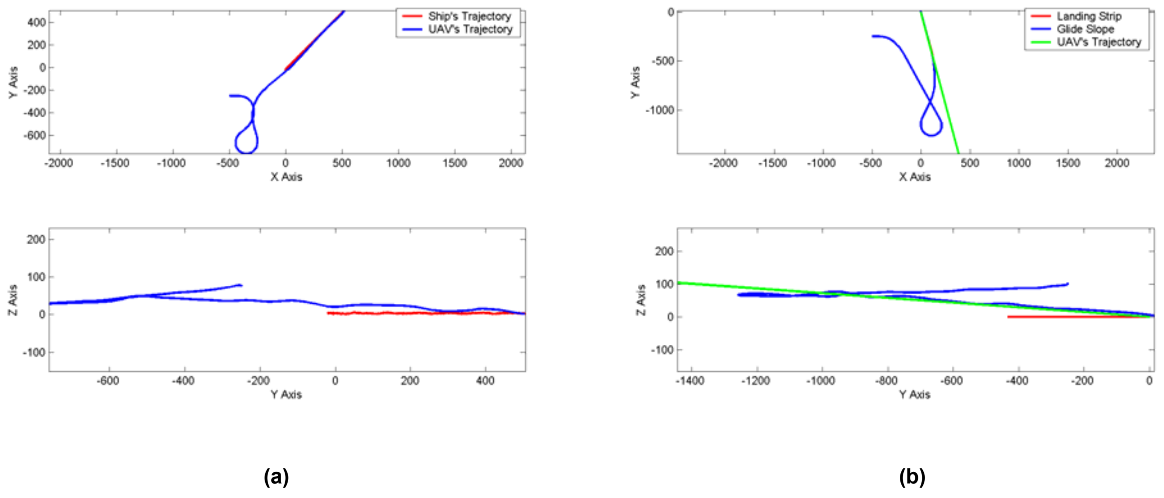


Figure 51. *XY* and *YZ* Plots of a Landing Approach. (a) Shipboard Landing. (b) Runway Landing. Measurement Noise and No Wind.

As in the previous case, the position and velocity errors in *Y* and *Z*, shown in Figure 52 and Figure 53, are driven to zero by the control system. It is important to note how the commanded pitch rate presents the effect known in control systems as *chattering.* Regardless of this effect, the control system still manages to steer the UAV into the recovering net correctly. The oscillations in the position error in *Z*, for the shipboard landing case, are due to the movement of the ship and the measurement noise. They are more noticeable than in the previous scenario.

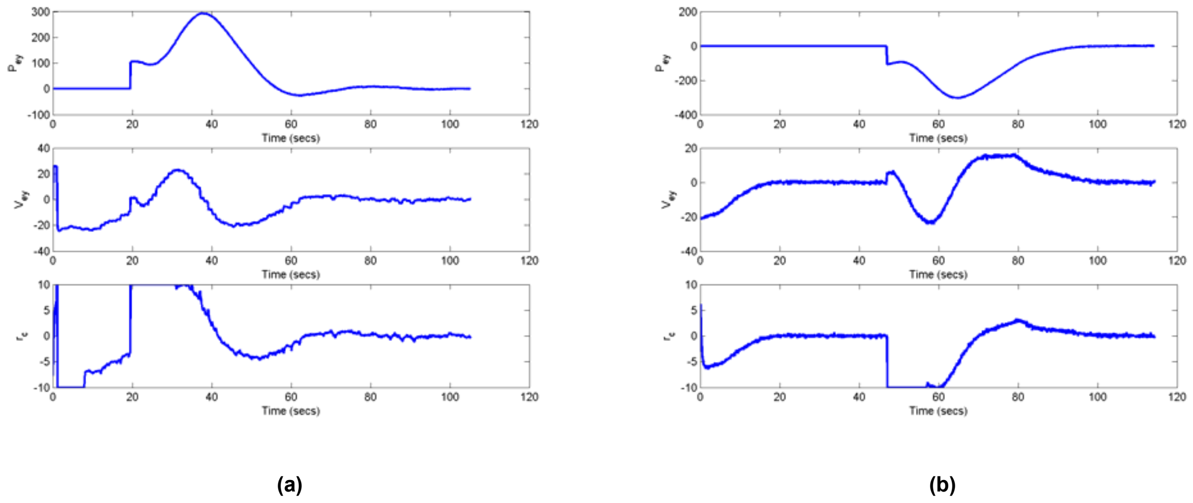


Figure 52. Position and Velocity Errors in Y Compared with the Commanded Yaw Rate. (a) Shipboard Landing. (b) Runway Landing. Measurement Noise and No Wind.

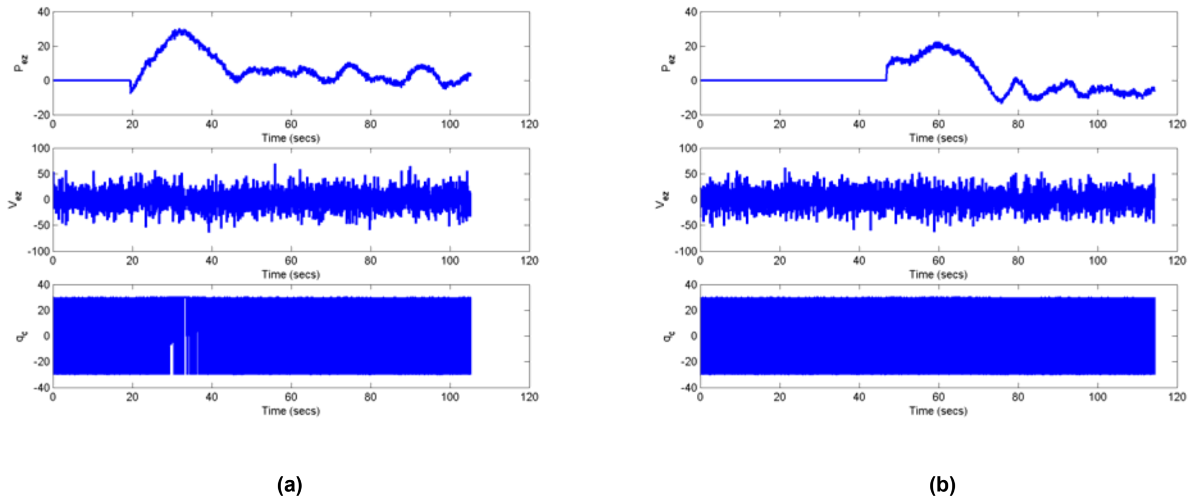


Figure 53. Position and Velocity Errors in Z Compared with the Commanded Pitch Rate. (a) Shipboard Landing. (b) Runway Landing. Measurement Noise and No Wind.

c. Measurement Noise and Wind Disturbances

In order to make the simulation more realistic, wind disturbances were introduced to the simulation. Real data was used, which was previously collected in the U.S. Army's Yuma Proving Grounds (YPG), Yuma, Arizona for the work presented in [12]. The wind was set up to run cyclically in the simulation, so if the data collected reached its end before the simulation was stopped, it would start again from the beginning in order to provide continuous wind for the simulation. The following figure presents the *X*, *Y,* and *Z* components of the wind.

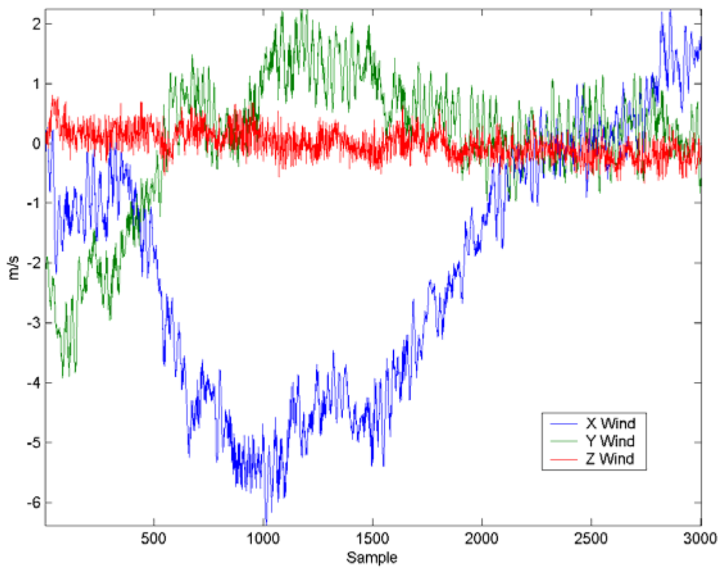
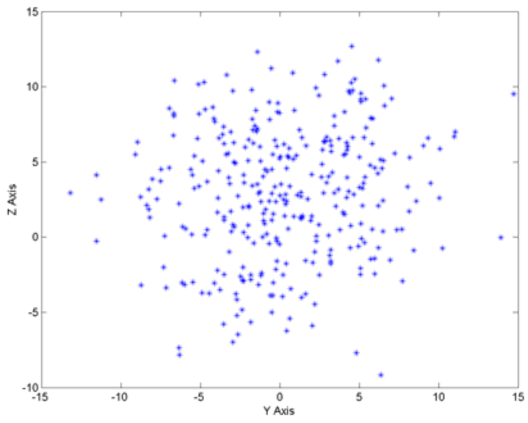


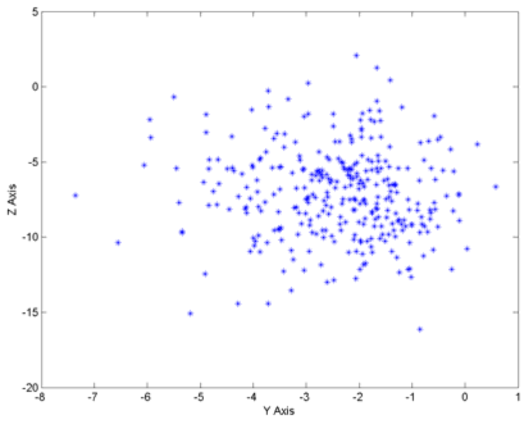
Figure 54. YPG's Wind Profile. (After Ref. [12].)

Once again, the same set of experiments as in the previous subsection were performed.

As expected, the following figure shows, that introducing wind disturbances negatively impacts the performance of the algorithm and downgrades the previously presented performance.



(a)



(b)

Figure 55. *YZ* Plot of the Position of the UAV's CG When Crossing the Plane of the Recovering Net. (a) Shipboard Landing. (b) Runway Landing. Measurement Noise and No Wind.

For shipboard landing, the error's means still remain small but the standard deviations grow even larger than before. On the contrary, for the runway landing, the means become significantly larger, but the standard deviations are barely changed.

LANDING TYPE	**MEAN (m)**		**STANDARD DEVIATION (m)**	
	Y	*Z*	*Y*	*Z*
Shipboard	0.3139	2.9194	4.6724	4.3451
Runway	–2.4227	–7.0919	1.2986	2.9819

Table 6. Means and Standard Deviations (in meters) of the Landing Errors. Measurement Noise and Wind.

The following figures show complete landing approaches. Figure 56 shows three-dimensional plots and Figure 57 shows the *XY* and *YZ* plots of the approaches. It can be seen that due to the wind disturbances, the UAV must perform more complicated maneuvers to position itself at the top of the glide slope.

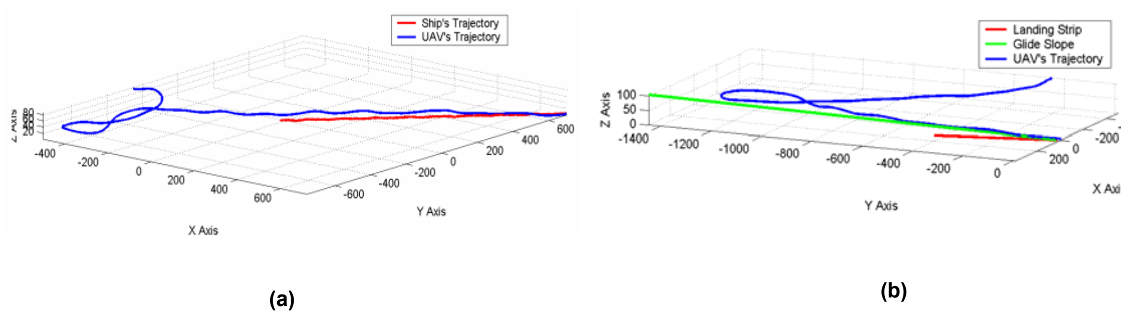


Figure 56. Three-Dimensional Plot of a Landing Approach. (a) Shipboard Landing. (b) Runway Landing. Measurement Noise and Wind.

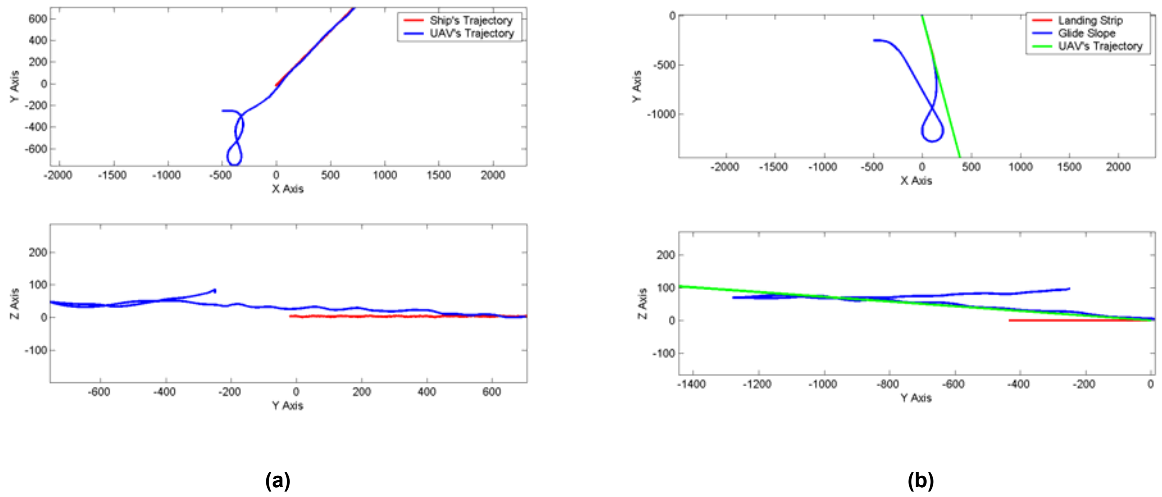
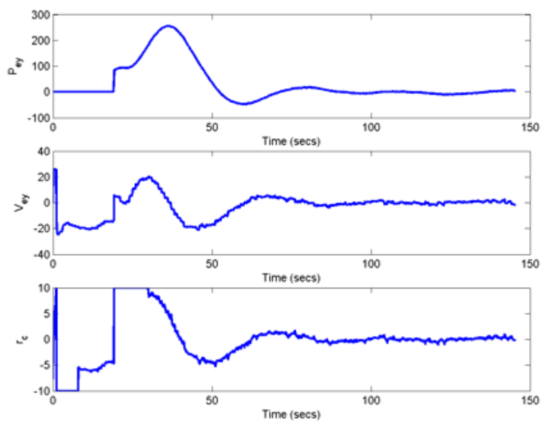


Figure 57. *XY* and *YZ* Plots of a Landing Approach. (a) Shipboard Landing. (b) Runway Landing. Measurement Noise and Wind.

As in the previous case, the position and velocity errors in *Y* and *Z*, shown in Figure 58 and Figure 59, are driven to zero by the control system. It is now noticeable how the position error in *Z* is greater than in the previous scenarios.



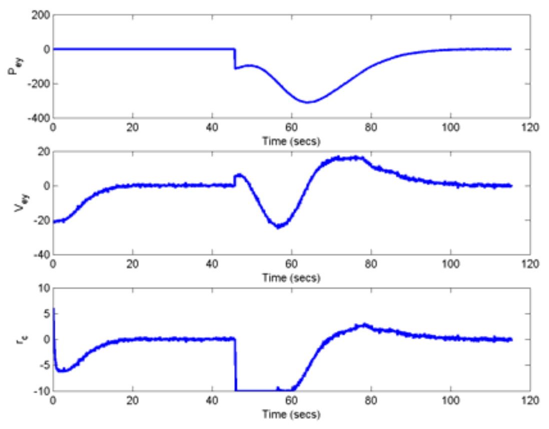
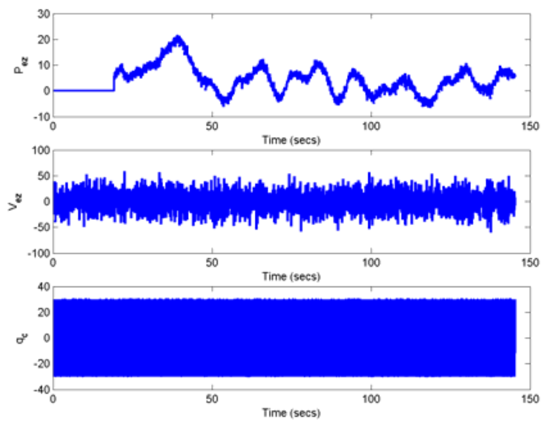


Figure 58. Position and Velocity Errors in Y Compared with the Commanded Yaw Rate. (a) Shipboard Landing. (b) Runway Landing. Measurement Noise and Wind.



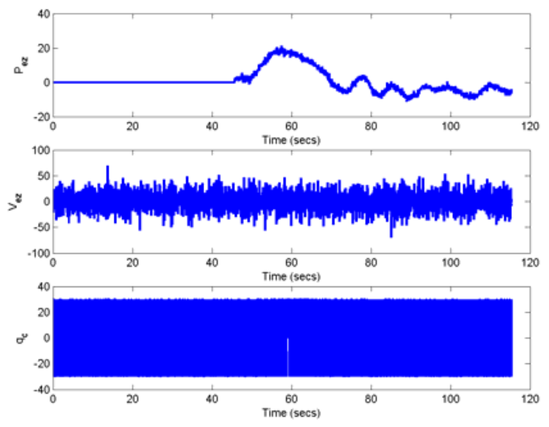


Figure 59. Position and Velocity Errors in Z Compared with the Commanded Pitch Rate. (a) Shipboard Landing. (b) Runway Landing. Measurement Noise and Wind.

d. Results Summary

Table 7 shows a summary of the results previously presented regarding the means and variances of the errors according to the simulation scenarios.

CONDITIONS		**MEAN (m)**		**STANDARD DEVIATION (m)**	
	Scenario	Y	Z	Y	Z
Ideal Conditions	*Ship*	0.1950	0.3689	1.8166	0.2794
	Runway	0.3533	–0.5254	0.2753	0.0294
Measurement Noise Only	*Ship*	0.1255	1.0930	4.1122	3.7121
	Runway	0.0578	–1.5822	1.2455	2.3719
Measurement Noise and Wind	*Ship*	0.3139	2.9194	4.6724	4.3451
	Runway	–2.4227	–7.0919	1.2986	2.9819

Table 7. Summary of Mean and Standard Deviations (in meters) for the Three Scenarios

Figure 60 and Figure 61 show the histograms of the occurrences of the position errors in *Y* and *Z*. These figures corroborate what was previously stated—even though there is not a significant change in the means as the simulation becomes more realistic, the spread of the data changes, thus decreasing the probability of the UAV hitting the recovering net. Nevertheless, it is worth noting that there is not a significant change in the means or in the standard deviations between the noise only and the noise and wind conditions. As expected, the algorithm has better results in the runway scenario.

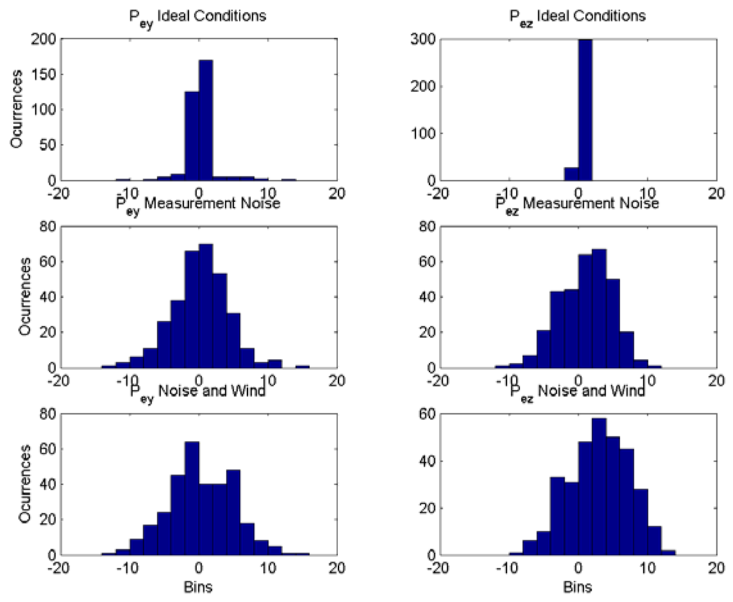


Figure 60. Histograms of the Occurrences of the Position Errors in *Y* and *Z* for Each of the Simulation Conditions. Shipboard Landing.

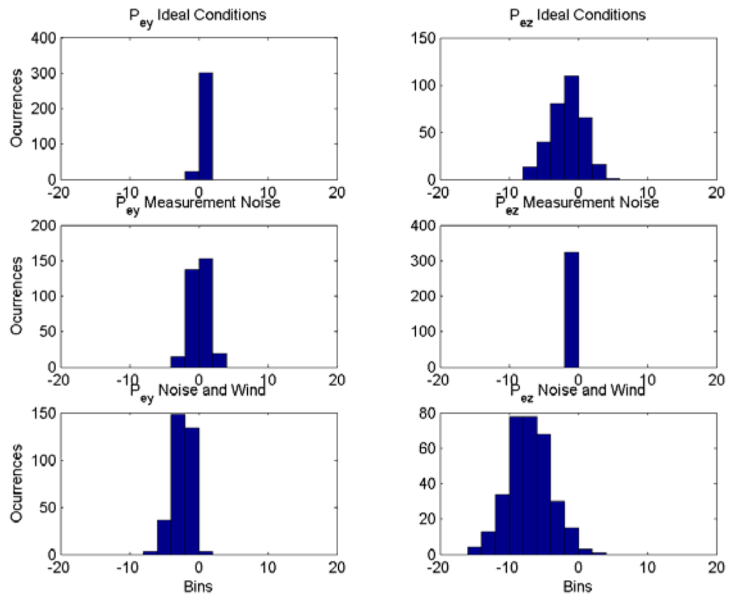


Figure 61. Histograms of the Occurrences of the Position Errors in *Y* and *Z* for Each of the Simulation Conditions. Runway Landing.

B. HARDWARE IN THE LOOP SIMULATION

With successful results in the computer simulation of the algorithm, the next step was to perform Hardware in the Loop (HIL) simulation. This was done in order to verify the following:

- The correct implementation of the Piccolo protocol reader/decoder and encoder/writer as well as the performance of the serial port 1 (COM1) in full duplex mode.
- How the latencies in the communications channel and drop outs in the data provided by the sensors affected the performance of the algorithm.
- The correct implementation of the autopilot in the Piccolo's firmware.
- The correct implementation of the GUI and the ability to switch between Piccolo's and NPS's autopilot effortlessly.
- The performance of the autonomous landing algorithm running in real-time in the Autonomous Landing Flight Control Computer.

Cloud Cap Technologies Inc. provided the software and hardware required to perform the HIL simulation. The setup and the interaction of the components will be described in the following subsection. The successful results of the HIL simulation are presented after that.

1. Setup

Included in the Piccolo system, Cloud Cap Technologies Inc, provides a set of software/hardware tools to perform HIL simulation. The software is an application that simulates the behavior of an aircraft in flight, thus its dynamics. It also simulates all the data that the aircraft's sensors would collect in real flight and provides a way to include wind disturbances in the simulations. All the data produced by this simulator is sent through an Universal Serial Bus to Controller Area Network (USB/CAN) bus adapter that connects the computer running the simulation and the Piccolo Avionics.

Figure 62 shows a diagram of the components involved in the HIL simulation and their interconnections.

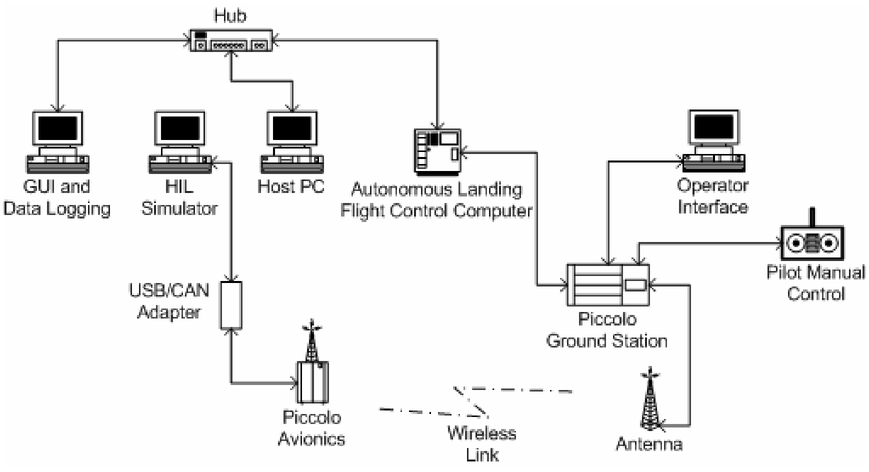


Figure 62. Diagram of the HIL Simulation.

The above HIL setup was done in the Unmanned Systems Lab at the NPS. The results presented in the following subsection were obtained in this lab. Figure 63 shows a picture of the setup.

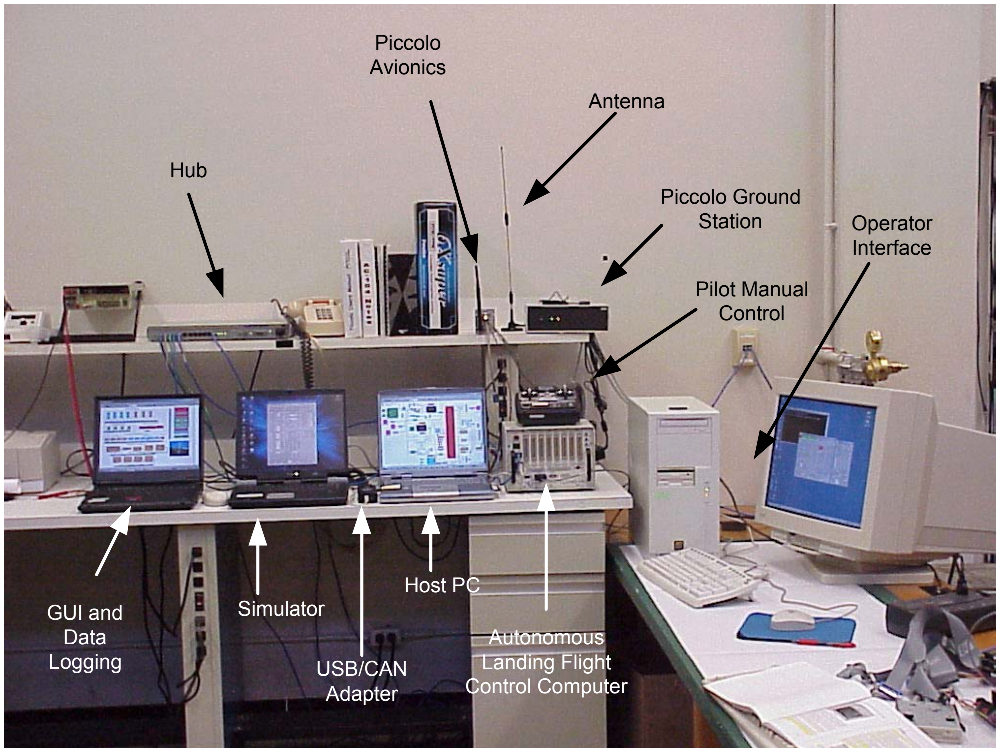


Figure 63. HIL Simulation Setup at the Naval Postgraduate School Unmanned Systems Lab.

For this simulation, a runway landing scenario was setup to prepare for the flight test, which will be conducted on a runway in Tucson, Arizona.

2. Results

The following figures show a complete landing approach. Figure 64 shows a three-dimensional plot, and Figure 65 shows the *XY* and *YZ* plots of the approach.

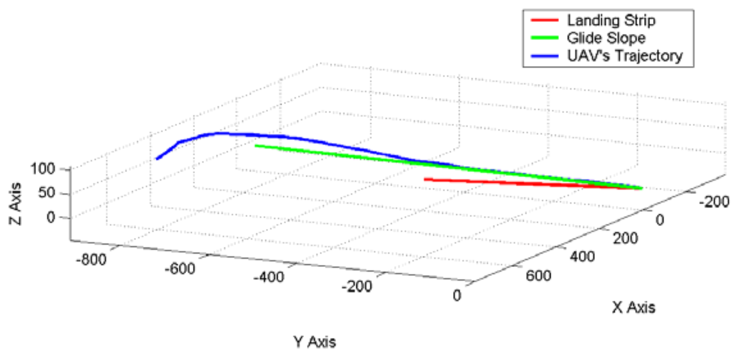


Figure 64. Three-Dimensional Plot of a Landing Approach in HIL Simulation. Noise and Wind .

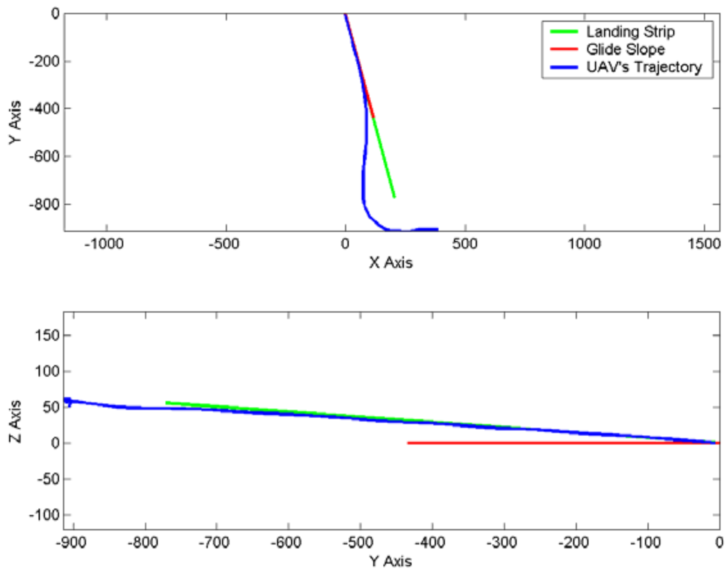


Figure 65. *XY* and *YZ* Plots of a Landing Approach in HIL Simulation. Noise and Wind.

The following figure shows the position of the center of gravity of the UAV (CG) with respect to the middle point of the net, when the UAV crosses the *YZ* plane of the net for the landing approach showed above. It can be seen that the position error in Z is less than 0.2 m and that the position error in Y is less than 4 m.

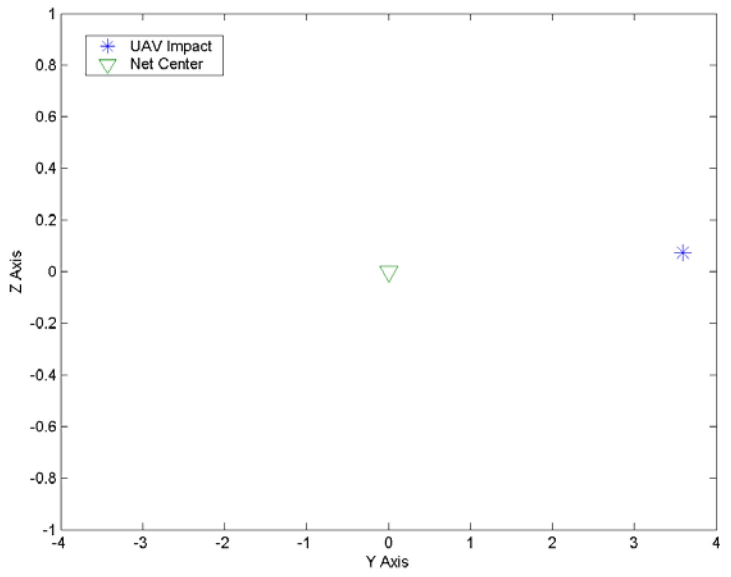


Figure 66. *YZ* Plot of the Position of the UAV's CG When Crossing the Plane of the Recovering Net. HIL Simulation.

From the above results, several of the items to be verified in the HIL were solved. Clearly, the implementation of the NPS autopilot in the Piccolo's firmware was done correctly. The parameters configured by the GUI were interpreted correctly by the autonomous landing algorithm and allowed to switch between Piccolo's and NPS's autopilot .

Regarding the position and velocity errors, shown in Figure 67 and Figure 68, they were all driven to zero by the control system. Therefore, the Piccolo protocol reader/decoder and the encoder/writer work correctly. Also, the use of serial port 1 (COM1) in full duplex mode did not show any signs of undermining the performance of the algorithm.

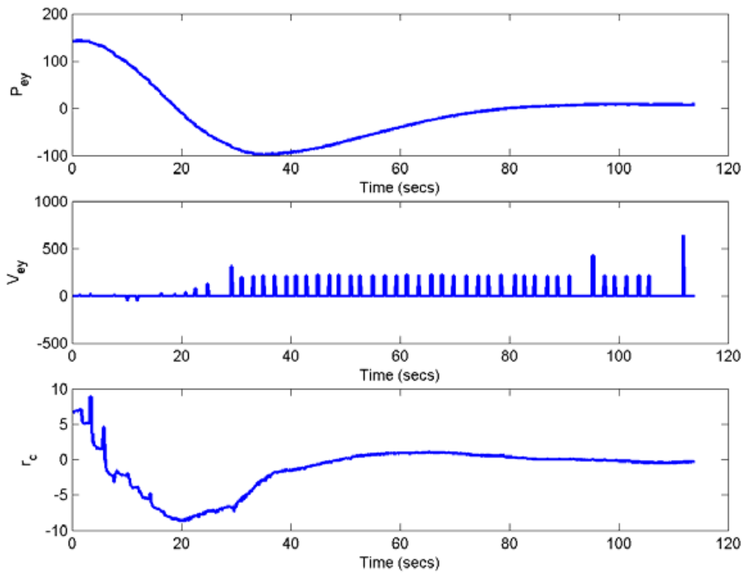


Figure 67. Position and Velocity Errors in Y Compared with the Commanded Yaw Rate in HIL Simulation. Noise and Wind.

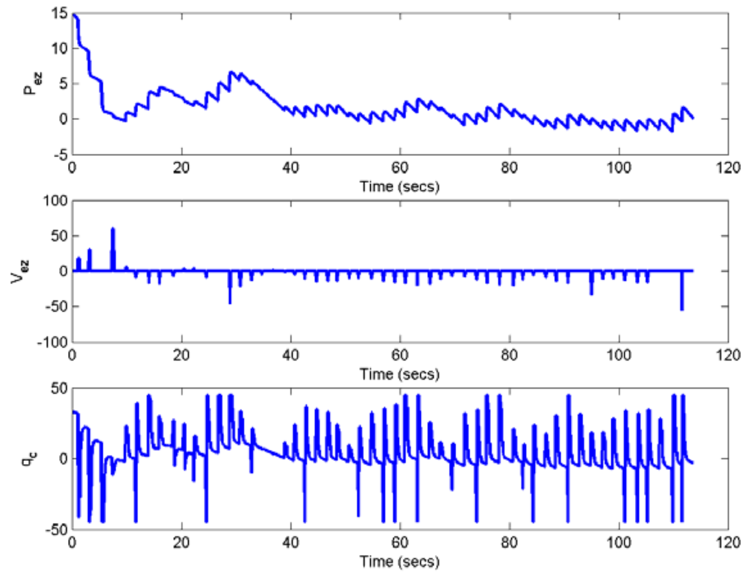


Figure 68. Position and Velocity Errors in Z Compared with the Commanded Pitch Rate in HIL Simulation. Noise and Wind.

It is noticeable in the above figures that the position and velocity errors, specially in the longitudinal channel (*Z*) have a kind of *jigsaw* effect that was not present in the computer simulations. This is a result of the position and velocity filter (discussed in Chapter II Section B Subsection 5) performing dead reckoning between fixes. As a byproduct of this effect, much of the chattering in the commanded pitch rate is eliminated in comparison with the previous results.

Finally, the latencies in the communications and/or the sensor data were studied. It was found, as shown in the following figure, that although most of the time the communications and the data provided by the sensors are reliable, there were several drop outs of considerable length. Figure 69 shows an extreme case, where a seven-second dropout in the last meters of the landing approach affected the performance of the algorithm. Other cases of small drop outs besides those happening in the last seconds of flight seemed to have little or no effect on the overall performance of the algorithm.

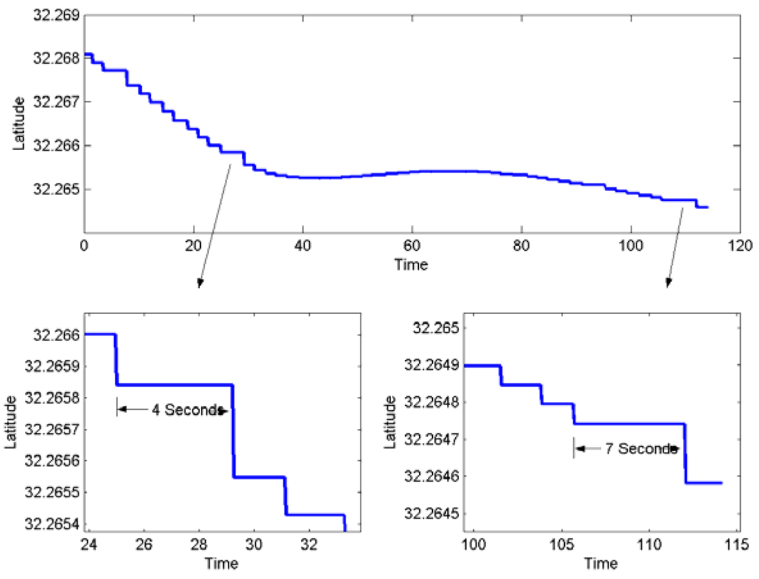


Figure 69. Dropouts in the Sensor Data during HIL Simulation.

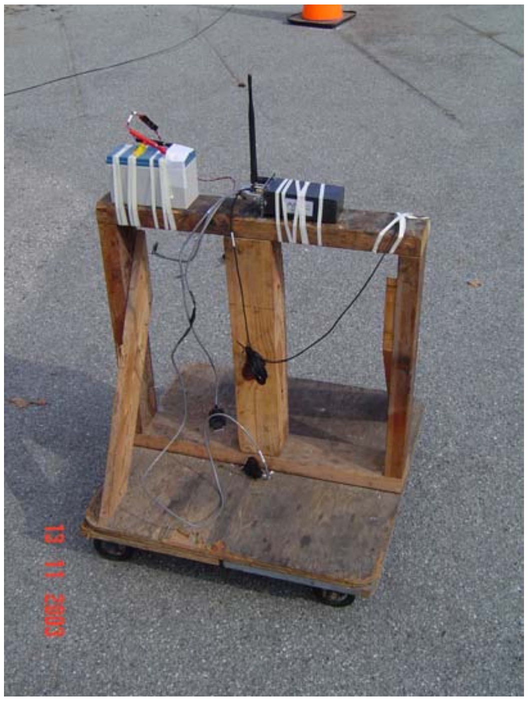
C. GROUND TEST

1. Setup

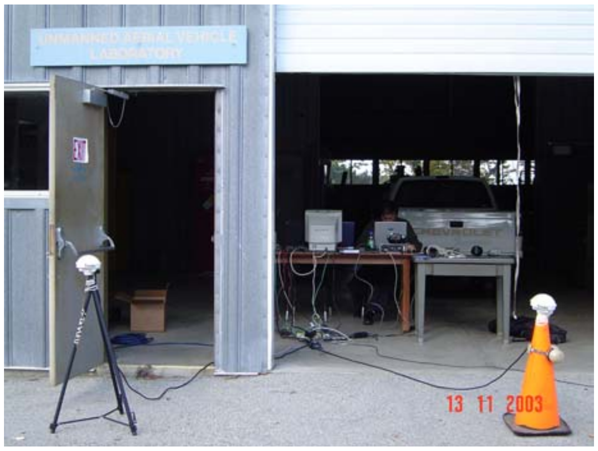
The last test performed before an actual flight was the ground test performed at the NPS's UAV lab at the Navy's Monterey Golf Course. This was done in order to verify the following:

- The duration and frequency of dropouts in the telemetry data provided by the Piccolo Ground Station.
- The existence of drifting in the sensors, if any.
- Once more, the correct implementation of the Piccolo protocol reader/decoder.

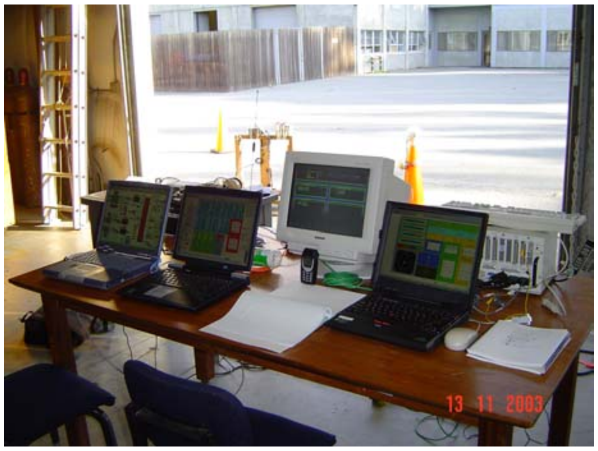
To achieve this, two different tests were performed. The first consisted of tying the Piccolo Avionics to a push cart and moving it along a triangular pattern established in the parking lot of the lab. The second consisted in leaving the Piccolo Avionics static and analyzing the drifting of the sensors. Figure 70 shows a picture of the actual set up at the lab.



(a)



(b)



(c)

Figure 70. Ground Test Setup at the NPS's UAV Lab. (a) Piccolo Avionics Tied to a Wheel Cart. (b) Ground Station GPS Receivers. (c) Ground Station Setup.

2. Results

The following figure shows the results of the first ground test. As illustrated, the triangular path followed by the Piccolo Avionics was correctly reconstructed by the collected data, thus proving once more that the Piccolo protocol reader/decoder had been implemented correctly.

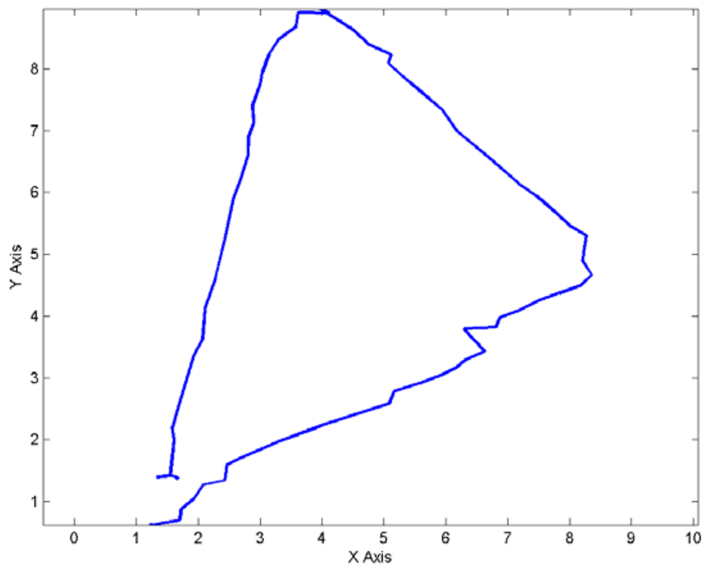


Figure 71. Triangular Path Followed by the Piccolo Avionics during the Ground Tests.

The second part of the ground test was performed by leaving static the avionics and collecting data for more than fifteen minutes. The following figure shows the drifting and variations of some of the data provided by the avionics. The GPS position (converted to the local reference frame *{L}*) drifted approximately one meter in each direction (*X* and *Y*) while the calculated Euler angles, roll, and pitch had a variation of $\pm 1.5^{\circ}$ and the yaw had a variation of $\pm 3^{\circ}$.

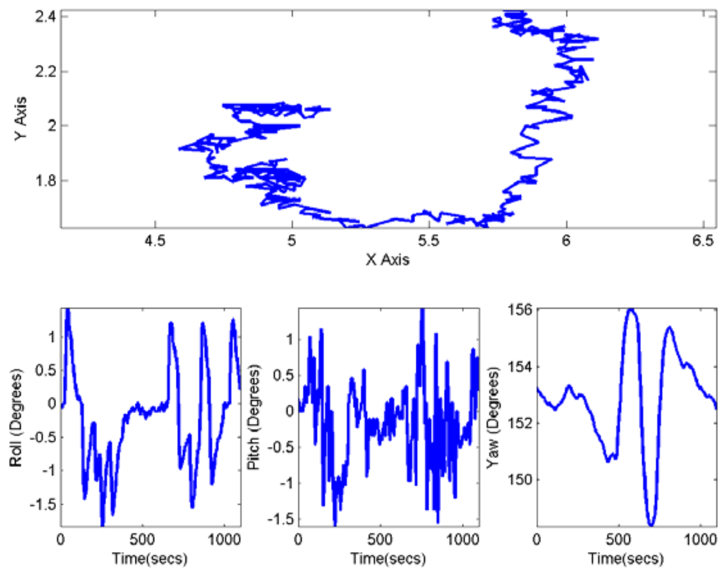


Figure 72. Drifting and Variation of the Signals Provided from the Sensors in the Ground Test.

The last test performed during the ground tests consisted in analyzing the frequency and duration of the dropouts in the GPS signals. For simplicity, the stream of data that was analyzed was the seconds of the GPS signal. This signal was differentiated and the results are shown in the following figure.

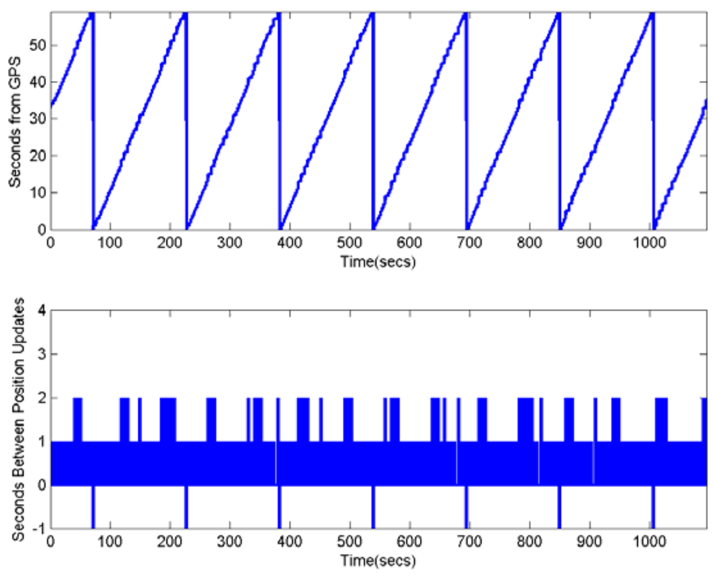


Figure 73. GPS Seconds from the Telemetry Message of the Piccolo Protocol and the Delays between Updates Obtained during the Ground Test.

No dropouts longer than two seconds appeared during the test. This result is noticeably better than that obtained during the HIL simulation in which the simulator produced drop outs of up to seven seconds, as shown in Figure 69.

D. SUMMARY

The results from different simulation scenarios as well as ground tests were presented in this chapter. These scenarios included ideal conditions, measurement noise only, and measurement noise and wind disturbances to the system. The following chapter presents the conclusion and makes some recommendations regarding follow-on work.

V. CONCLUSIONS AND FOLLOW-ON WORK

A. CONCLUSIONS

This thesis presented the design, development and implementation of an autonomous landing system for the Silver Fox UAV. The autonomous landing algorithm was developed in Simulink, a widely recognized tool for control systems and signal processing development. All the low-level device drivers were developed in ANSI C and integrated seamlessly into the Simulink models, providing the required link between hardware and software. The feasibility of the algorithm presented here was demonstrated in computer and HIL simulation, as well as ground tests. The system presented here is scheduled to have a flight test in the near future.

The fundamental idea on which this system was developed is mostly based on the work presented in [3]. The idea, although simple, is very effective.

The approach consisted of creating an imaginary glide slope and having the UAV track it as it approaches the landing point. Based on the position of the UAV, the algorithm creates a new reference frame that points either to the top of the glide slope, or to the landing point, depending on the previously explained parameters. Position and velocity errors are generated in this reference frame. The position errors indicate how far the UAV's CG is from the glide slope. The velocity errors indicate the misalignment of the UAV's velocity vector with that of the glide slope. Due to the introduction of this reference frame, a three-dimensional problem is simplified to two dimensions, making the control system design simpler.

The system presented in this work provides the following advantages:

- The system is not limited to day-time operations, as is generally the case for pilot-operated tactical UAVs.
- Since the Piccolo system is not tied to any particular UAV, it can be migrated to other UAVs with little effort.
- The algorithm works for both, shipboard and runway landing.
- It can easily be operated by one person.

B. FOLLOW-ON WORK

This work presents only an initial approach to solve the autonomous shipboard landing problem. There is certainly much to do, especially in the control system. Nevertheless, this work should only be done after obtaining real flight data.

The development of a more robust control algorithm is certainly a good area to pursue. Particularly, since this system is to perform in real-time, highly sophisticated control algorithms may be such a high burden to the Autonomous Landing Flight Control Computer that they may worsen, instead of improve, the overall performance of the system.

A predictor to allow the aircraft to come about and restart the approach in case it identifies that it will miss the recovering net is also an element that would add reliability to the overall system.

An observer or a predictor to have a better estimation of the current and future position of the ship would certainly improve the performance of the algorithm, again, only as long as it does not imply a heavy burden for the Autonomous Landing Flight Control Computer.

APPENDIX A: SOURCE CODE OF THE DRIVERS

This appendix presents the ANSI C source code of the S-Functions implemented in Simulink to be used as drivers. Each S-Function presented includes a header that briefly explains what it is used for.

```
//************************************************************
// File:        pic_rec_v4..c
// Author(s):   Mariano I Lizarraga and Vladimir Dobrokhodov
// Date:        August-23-2003
// Description: This is the mdlOutputs of the S-Function that
//              reads the Piccolo Protocol
//************************************************************
/* Function to compute outputs */
static void mdlOutputs(SimStruct *S, int_T tid)
{
//#ifndef MATLAB_MEX_FILE

       unsigned char  data[500];  /* temp char holder */
       int_T tmp=0;
       unsigned char *buf = (unsigned char *)ssGetDWork(S, 0);       /* uchar buffer bytes */
       unsigned char *remainsdata = (unsigned char *)ssGetDWork(S, 1);       /* buffer remain */
       int *current   = ssGetIWork(S);                         /* current = addr of current pointer in
       int *recLength = ssGetIWork(S) + 1;                     /* recLength = addr of receieved data */
       int *bufCount = ssGetIWork(S)+ 2;              /* count number of useful bytes in buf. */
       int *remainsSize = ssGetIWork(S)+ 3;           /* count size of the inner loop remains */
       int i=0, ii = 0, j=0, notenoughbytes_inner=0, notenoughbytes_outer=0,k=0;
       int crcreal=0,crc=0,crch=0, numBytesAvail=0, flag,packetSize=0 ;
       int serbufCount; //number of bytes available in the port
       int_T temp[1024];// make an aliase for the uPtrs
       int NETADR = (int)mxGetPr(NETADR_ARG)[0];
       unsigned __int8 temp_crc=0;

       InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);
       //real_T            *y0    = ssGetOutputPortRealSignal(S,1);


       // Get the number of bytes available in the port
       serbufCount = (int)mxGetPr(WIDTH_ARG)[0];


       for (i=0; i<serbufCount; i++){temp[i] = ( int_T)(*uPtrs[i]);}

       *bufCount = serbufCount + *current;
       // place new chunk of data at the end of previously unprocessed data
       // 'buf' also keeps the remains of previous step ending at the *(buf+current),
       // where 'current' is the position of last byte
       for (i=*current;i<*bufCount;i++)
       {
           *(buf+i)=(unsigned char)temp[i-*current];
       }

       i=0;
       // start main loop
       while(!notenoughbytes_outer && i<(*bufCount) )
       {
          // Iteration to find 4 bytes header,
          // so only complete(not fragmented) message is going to be processed
          while ( (*(buf+i) == 80 && *(buf+i+1) == 10  && *(buf+i+2) == 0
             && *(buf+i+3) == NETADR && *(buf+i+4) == MESSAGE_TYPE)==0 &&
             i< (*bufCount-5))  // Looking for header {50 0A 00 00//63 03}
          {  // Simply shifting along the *buf array to find a header
             i = i + 1;
          } // end of while

          // If we've found a top level header
          // {50 0A 00 00//63 03} do the rest of checking
          if (*(buf+i) == 80 && *(buf+i+1) == 10 && *(buf+i+2) == 0 &&
             *(buf+i+3) == NETADR && *(buf+i+4) == MESSAGE_TYPE)
          { // we've got a top level header
             flag=77;
          } // end of if

          // check availability of at least *(buf+i+6)
          // recall that *(buf+i+6) contains the size of the payload
          // in turn, the '*(buf+i+6)+9' is the length of outerlevel package
          if ( (i+6)>=*bufCount || (i+ *(buf+i+6)+9)>= *bufCount)
          {
             notenoughbytes_outer=1;
          }
          else// we do have enough bytes
          {
             // if we have a top level header and
             // enough bytes to proceed
             if (flag==77 && (i+*(buf+i+6)+9)<*bufCount )
             {  //=============================
                // Top Level HeaderChecksum Procedure
                //=============================
                // Read first 7 bytes into the array to get header CRC
               crch=0;//just in case
                for (k=0; k<7; k++)
                {// put the value in a temporary variable
                   temp_crc = ( unsigned __int8)(*(buf+i+k));
                   // Perform the CRC
                   crch = (unsigned __int16)(crch^temp_crc);
                   // clear the crc_temp to use it in the
                   // 2nd level crc chech procedure
                   temp_crc = 0;
                }//end of for

                //========================
                // End Top Level HeaderChecksum
                //========================
                // if the Top Level HeaderCRC is equal to that embedded in the
                // packet, thus *(buf+i+7)
               if (crch== *(buf+i+7))
                {
                   // Succeded with header check
                   flag=1;
                   //========================
                   // 1st Level CRC Procedure
                   //========================
                   // '*(buf+i+6)+7' is the amount of bytes used to get 1st Level CRC
                   // next 2 bytes (8,9) determine the actual CRC embedded into the stream

                   crc=0;//just in case
                   crc=(int)GetCRC((int)(*(buf+i+6)+7),(buf+i));

                   //========================
                   // End 1st Level CRC
                   //========================
                   // read the crc embedded in the message
                   crcreal=(*(buf+i+*(buf+i+6)+8))*(256) + *(buf+i+*(buf+i+6)+9);

                   // if the CRC is different then that embedded in the
                   // message

                   if (crc != crcreal)
                   {
                      // discard this chunk of data and shift to a new position
                      i=i+ *(buf+i+6)+9;
                      //flag=0;
                   }
                   else
                   {
                      // we have a valid 1st level CRC
                      flag=2;

                      //autopilot data=payload is located between i+8:i+8+Size{Size=temp(i+6)}
                      numBytesAvail= *(buf+i+6);

                      //Send packet out {crc}
                      ////////////////////////////////////////////////////////////////
                      //put valid bytes of payload to data - initial array to be splitted packets
                      FillDataRemains(data, *remainsSize, remainsdata,numBytesAvail, (buf+i));

                      notenoughbytes_inner = 0;
                      ii = 0;

                      // check, check, check: ii<(numBytesAvail + *remainsSize) &&
                      while(notenoughbytes_inner==0)
                      {
                         FindHeader(data, numBytesAvail, &ii);
                         // read size of first packet
```

```
                        // if there is enough bytes to obtain the size
                        if (ii<(numBytesAvail + *remainsSize - 3))
                        {
                        // obtain it
                           packetSize = data[ii+3];
                        }
                        // else
                        else
                        {
                        // make the packet size a very big number to
                        // force the next if to fail
                           packetSize = 500000;
                        }


                        ///////////////////////////////////////////********
                        if ((ii+packetSize+6) <= (numBytesAvail+*remainsSize))
                        {
                           // ProcessPacket();
                           ProcessPacket(data, &flag, &ii, S);
                           // ii is changed inside of ProcessPacket
                        }
                        else
                        {
                           //FillRemainsInnerLevel
                           FillRemainsInnerLevel(data, remainsSize, ii, remainsdata,
                                                 numBytesAvail);
                           notenoughbytes_inner=1;
                        }
                     } // end while notenoughbytes_inner==0

                     /////////////////////////////////////////////////////////////
                     i=i+(*(buf+i+6))+10;// shift to next byte of outer level
                  } // if else crc != crcreal      flag 2
               } // if crch==temp(i+7)             flag 1
               else
               {
                   i=i+7;
                }//
            } // if flag==77 &&(i+*(buf+i+6)+9)<*bufCount  flag 77

         } // end of notenoughbytes_outer

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

      }// end of while


      //Substitute remain bytes from this step at the beginning of "buf"
      //save remain bytes into the "buf" and shift current to the end of a new buf
      if (i<(*bufCount))
      {
         *bufCount=*bufCount-i;  //number of remain bytes
         // buf[j]=buf[i]
         for (j=0;j<*bufCount; j++)
         {
            *(buf+j)=*(buf+i);
            i++;
         }//  end of for
         *current=*bufCount;
      }

//#endif
}

//***************************************************************
// File:        dec_diag..cpp
// Author(s):   Mariano I Lizarraga and Vladimir Dobrokhodov
// Date:        April-15-2003
// Description: This is the mdlOutputs of the S-Function that
//              decodes the diagnostics message of the Piccolo Protocol
//***************************************************************

static void mdlOutputs(SimStruct *S, int_T tid)
{
      // Declare the variables to be used in the function
      // counters
      int_T       i=0;
      double      vr_temp=0;
      int_T temp[17]; // make an alias for the uPtrs
```

```
       // Incoming data stream,
       // *uPtrs[0]- determines the length of useful bytes{28 for the IMU}
       InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);
       real_T            *y    = ssGetOutputPortRealSignal(S,0);

       // read the whole incoming string and assign it to a temporary variable
       // in order to avoid mistakenly overwrting the incoming data
       for (i=0; i<(*uPtrs[0]); i++){temp[i] = ( int_T)(*uPtrs[i+1]);}


       // %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
       //   Generate the Output
       // %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

       // Output Channel 1: DeadMan Bit 7 (LSB)
       vr_temp = temp[16] % 2;
    *y=(real_T)vr_temp;



}


//*****************************************************************
// File:        dec_autopil..c
// Author(s):   Mariano I Lizarraga and Vladimir Dobrokhodov
// Date:        April-11-2003
// Description: This is the mdlOutputs of the S-Function that
//              decodes the autopilot message of the Piccolo Protocol
//*****************************************************************

static void mdlOutputs(SimStruct *S, int_T tid)
{
       // Declare the variables to be used in the function
       // counters
       int_T       i=0,j=0;
       double      vr_temp=0;
       int_T temp[56]; // make an alias for the uPtrs


       // Incoming data stream,
       // *uPtrs[0]- determines the length of useful bytes{28 for the IMU}
       InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);
       real_T            *y    = ssGetOutputPortRealSignal(S,0);

       // read the whole incoming string and assign it to a temporary variable
       // in order to avoid mistakenly overwrting the incoming data
       for (i=0; i<(*uPtrs[0]); i++){temp[i] = ( int_T)(*uPtrs[i+1]);}


       // %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
       //   Start the Decoding Procedure
       // %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

          // Output Channel 1: Barometric Altitude Command (in meters)
       vr_temp = temp[0]*16777216 + temp[1]*65536 + temp[2]*256 + temp[3];
       if (vr_temp>2147483647) vr_temp =vr_temp-4294967295;
       *(y+0)=(real_T)(vr_temp/100);

       // Output Channel 2: Dynamic Pressure Command (in Pa)
       vr_temp =0;
       vr_temp = temp[4]*256 + temp[5];
       *(y+1)=(real_T)vr_temp;

       // Output Channel 3: Turn Rate Command (in rad/sec)
       vr_temp =0;
       vr_temp = temp[6]*256 + temp[7];
       *(y+2)=(real_T)(vr_temp/1000);

       // Output Channel 4: Waypoint Number
       *(y+3)=(real_T)temp[8];

       // Output Channel 5: Flap Deflection (in rads)
       vr_temp =0;
       vr_temp = temp[9];
       if (vr_temp>127) vr_temp = vr_temp - 128;
       *(y+4)=(real_T)vr_temp*0.01745329251994;

       // Output Channel 6: Altitude Loop Control
       // 0 -> Off
       // 1 -> Use the Passed Command
       // 2 -> Use the Command from the Navigator
       vr_temp =0;
       vr_temp = temp[10] >> 6;
       vr_temp = (int)vr_temp & 0x03;
       *(y+5)=(double)(vr_temp);

       // Output Channel 7: Dynamic Pressure Loop Control
       // 0 -> Off
       // 1 -> Use the Passed Command
       // 2 -> Use the Command from the Config Manager
       vr_temp =0;
       vr_temp = temp[10] >> 4;
       vr_temp = (int)vr_temp & 0x03;
       *(y+6)=(double)(vr_temp);

       // Output Channel 8: Engine Kill On
       vr_temp =0;
       vr_temp = temp[10] >> 3;
       vr_temp = (int)vr_temp & 0x01;
       *(y+7)=(double)(vr_temp);

       // Output Channel 9: Deadman Engine Kill On
       vr_temp =0;
       vr_temp = temp[10] >> 2;
       vr_temp = (int)vr_temp & 0x01;
       *(y+8)=(double)(vr_temp);

       // Output Channel 10: Turn Rate Loop Control
       // 0 -> Off
       // 1 -> Use the Passed Command
       // 2 -> Use the Command from the Navigator
       vr_temp =0;
       vr_temp = (int)temp[10] & 0x03;
       *(y+9)=(double)(vr_temp);

       // Output Channel 11: Flaps Loop Control
       // 0 -> Neutral
       // 1 -> Use the Passed Command
       // 2 -> Use the Command from the Config Manager
       vr_temp =0;
       vr_temp = temp[11] >> 6;
       vr_temp = (int)vr_temp & 0x03;
       *(y+10)=(double)(vr_temp);

       // Output Channel 12: Tracker Loop Control
       // 0 -> Off
       // 1 -> Tracker Enabled
       vr_temp =0;
       vr_temp = temp[11] >> 4;
       vr_temp = (int)vr_temp & 0x03;
       *(y+11)=(double)(vr_temp);

       // Output Channel 13: Manual Steering On
       vr_temp =0;
       vr_temp = temp[11] >> 2;
       vr_temp = (int)vr_temp & 0x01;
       *(y+12)=(double)(vr_temp);


       // Output Channel 14: Autolaunch On
       vr_temp =0;
       vr_temp = temp[11] >> 1;
       vr_temp = (int)vr_temp & 0x01;
       *(y+13)=(double)(vr_temp);

       // Output Channel 15: Global On
       vr_temp =0;
       vr_temp = (int)temp[11] & 0x01;
       *(y+14)=(double)(vr_temp);

}

//*******************************************************************
// File:        dec_telemtry..c
// Author(s):   Mariano I Lizarraga and Vladimir Dobrokhodov
// Date:        April-5-2003
// Description: This is the mdlOutputs of the S-Function that
//              decodes the telemetry message of the Piccolo Protocol
//*******************************************************************
static void mdlOutputs(SimStruct *S, int_T tid)
```

```
{
      // Declare the variables to be used in the function
      // counters
      int_T       i=0,j=0;
      double      vr_temp=0;
      int_T temp[60]; // make an alias for the uPtrs
      char ext[]="\0";
      int count=0,len_in;


      // Incoming data stream,
      // *uPtrs[0]- determines the length of useful bytes{28 for the IMU}
      InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);
      real_T            *y    = ssGetOutputPortRealSignal(S,0);

      // read the whole incoming string and assign it to a temporary variable
      // in order to avoid mistakenly overwrting the incoming data
      for (i=0; i<(*uPtrs[0]); i++){temp[i] = ( int_T)(*uPtrs[i+1]);}


      // %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      //   Start the Decoding Procedure
      // %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

      // Output Channel 1: Latitude
      vr_temp = temp[0]*16777216 + temp[1]*65536 + temp[2]*256 + temp[3];
      if (vr_temp>2147483647) vr_temp =vr_temp-4294967295;
      *(y+0)=(real_T)vr_temp*MSEC2RAD;

      // Output Channel 2: Longitude
      vr_temp =0;
      vr_temp = temp[4]*16777216 + temp[5]*65536 + temp[6]*256 + temp[7];
      if (vr_temp>2147483647) vr_temp=vr_temp-4294967295;
      *(y+1)=(real_T)vr_temp*MSEC2RAD;

      // Output Channel 3: Height (in meters)
      vr_temp =0;
      vr_temp = temp[8]*16777216 + temp[9]*65536 + temp[10]*256 + temp[11];
      if (vr_temp>2147483647) vr_temp=vr_temp-4294967295;
      *(y+2)=(real_T)vr_temp/100;

      // Output Channel 4: V North (in meters/sec)
      vr_temp =0;
      vr_temp = temp[12]*256 + temp[13];
      if (vr_temp>32768) vr_temp=vr_temp-65536;
      *(y+3)=(real_T)vr_temp/100;

       // Output Channel 5: V East (in meters/sec)
      vr_temp =0;
      vr_temp = temp[14]*256 + temp[15];
      if (vr_temp>32768) vr_temp=vr_temp-65536;
      *(y+4)=(real_T)vr_temp/100;

      // Output Channel 6: Number of Satellites
      *(y+5)=temp[16];

      // Output Channel 7: Number of Visible Satellites
      *(y+6)=temp[17];

      // Output Channel 8: Dop
      vr_temp =0;
      vr_temp = temp[18]*256 + temp[19];
      *(y+7)=(real_T)vr_temp;

      // Output Channel 9: Status
      vr_temp =0;
      vr_temp = temp[20]*256 + temp[21];
      *(y+8)=(real_T)vr_temp;

      // Output Channel 10: Year
      vr_temp =0;
      vr_temp = temp[22]*256 + temp[23];
      *(y+9)=(real_T)vr_temp;

      // Output Channel 11: Month
      *(y+10)=(real_T)temp[24];

      // Output Channel 12: Day
      *(y+11)=(real_T)temp[25];

      // Output Channel 13: Hour
```

```
*(y+12)=(real_T)temp[26];

// Output Channel 14: Minute
*(y+13)=(real_T)temp[27];

// Output Channel 15: Seconds
*(y+14)=(real_T)temp[28];

// Output Channel 16: TmpOat
vr_temp =0;
if(temp[29] >128) {vr_temp=temp[29]-256;}
else {vr_temp=temp[29];}
*(y+15)=(real_T)vr_temp;

// Output Channel 17: Altitude (in meters)
vr_temp =0;
vr_temp = temp[30]*16777216 + temp[31]*65536 + temp[32]*256 + temp[33];
if (vr_temp>2147483647) vr_temp=vr_temp-4294967295;
*(y+16)=(real_T)vr_temp/100;

// Output Channel 18: Tas
vr_temp =0;
vr_temp = temp[34]*256 + temp[35];
*(y+17)=(real_T)vr_temp/100;

    // Output Channel 19: X Track (in meters)
vr_temp =0;
vr_temp = temp[36]*16777216 + temp[37]*65536 + temp[38]*256 + temp[39];
if (vr_temp>2147483647) vr_temp=vr_temp-4294967295;
*(y+18)=(real_T)vr_temp/100;

// Output Channel 20: Y Track (in meters)
vr_temp =0;
vr_temp = temp[40]*16777216 + temp[41]*65536 + temp[42]*256 + temp[43];
if (vr_temp>2147483647) vr_temp=vr_temp-4294967295;
*(y+19)=(real_T)vr_temp/100;

 // Output Channel 21: Velocity in X (in meters/sec)
vr_temp =0;
vr_temp = temp[44]*256 + temp[45];
if (vr_temp>32768) vr_temp=vr_temp-65536;
*(y+20)=(real_T)vr_temp/100;

 // Output Channel 22: Velocity in y (in meters/sec)
vr_temp =0;
vr_temp = temp[46]*256 + temp[47];
if (vr_temp>32768) vr_temp=vr_temp-65536;
*(y+21)=(real_T)vr_temp/100;

// Output Channel 23: W South (in meters/sec)
vr_temp =0;
vr_temp = temp[48]*256 + temp[49];
if (vr_temp>32768)vr_temp=vr_temp-65536;
*(y+22)=(real_T)vr_temp/100;

// Output Channel 24: W West (in meters/sec)
vr_temp =0;
vr_temp = temp[50]*256 + temp[51];
if (vr_temp>32768)vr_temp=vr_temp-65536;
*(y+23)=(real_T)vr_temp/100;

// Output Channel 25: W Error (in meters/sec)
vr_temp =0;
vr_temp = temp[52]*256 + temp[53];
if (vr_temp>32768)vr_temp=vr_temp-65536;
*(y+24)=(real_T)vr_temp/100;

// Output Channel 26: Residue 0
*(y+25)=(real_T)temp[54];

// Output Channel 27: Residue 1
*(y+26)=(real_T)temp[55];

// Output Channel 28: Iterations
*(y+27)=(real_T)temp[56];

// Output Channel 29: Number of Points
*(y+28)=(real_T)temp[57];

// Output Channel 30: Wind Hour
*(y+29)=(real_T)temp[58];
```

```
        // Output Channel 31: Wind Minute
        *(y+30)=(real_T)temp[59];


}

//******************************************************************
// File:        dec_control.c
// Author(s):   Mariano I Lizarraga and Vladimir Dobrokhodov
// Date:        April-1-2003
// Description: This is the mdlOutputs of the S-Function that
//              decodes the control message of the Piccolo Protocol
//******************************************************************

static void mdlOutputs(SimStruct *S, int_T tid)
{
        // Declare the variables to be used in the function
        // counters
        int_T       i=0,j=0;
        double      vr_temp=0;
        int_T temp[56]; // make an alias for the uPtrs


        // Incoming data stream,
        // *uPtrs[0]- determines the length of useful bytes{28 for the IMU}
        InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);
        real_T            *y    = ssGetOutputPortRealSignal(S,0);

        // read the whole incoming string and assign it to a temporary variable
        // in order to avoid mistakenly overwrting the incoming data
        for (i=0; i<(*uPtrs[0]); i++){temp[i] = ( int_T)(*uPtrs[i+1]);}


        // %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        //   Start the Decoding Procedure
        // %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        // Output Channel 1: Number of Milliseconds since the system has been on
        vr_temp = temp[0]*16777216 + temp[1]*65536 + temp[2]*256 + temp[3];
        //if (vr_temp>2147483647) vr_temp =vr_temp-4294967295;
         *(y+0)=(real_T)vr_temp;

        // Output Channel 2: Static Pressure (in Pa)
        vr_temp =0;
        vr_temp = temp[4]*16777216 + temp[5]*65536 + temp[6]*256 + temp[7];
        *(y+1)=(real_T)vr_temp;

        // Output Channel 3: Dynamic Pressure (in Pa)
        vr_temp =0;
        vr_temp = temp[8]*256 + temp[9];
        *(y+2)=(real_T)vr_temp;

        // Output Channel 4: Outside Temperature (in Celsius)
        vr_temp =0;
        vr_temp = temp[10];
        if (vr_temp>127) vr_temp=vr_temp-256;
        *(y+3)=(real_T)vr_temp;

        // note that temp[11] is not used.

        // Output Channel 5: Roll Rate(in rads/sec)
        vr_temp =0;
        vr_temp = temp[12]*256 + temp[13];
        if (vr_temp>32768) vr_temp=vr_temp-65536;
        *(y+4)=(real_T)(vr_temp/1000);

        // Output Channel 6: Pitch Rate(in rads/sec)
        vr_temp =0;
        vr_temp = temp[14]*256 + temp[15];
        if (vr_temp>32768) vr_temp=vr_temp-65536;
        *(y+5)=(real_T)(vr_temp/1000);

        // Output Channel 7: Yaw Rate(in rads/sec)
        vr_temp =0;
        vr_temp = temp[16]*256 + temp[17];
        if (vr_temp>32768) vr_temp=vr_temp-65536;
        *(y+6)=(real_T)(vr_temp/1000);

        // Output Channel 8: Roll Rate Bias (in rads/sec)
        vr_temp =0;
vr_temp = temp[18]*256 + temp[19];
if (vr_temp>32768) vr_temp=vr_temp-65536;
*(y+7)=(real_T)(vr_temp/1000);

// Output Channel 9: Pitch Rate Bias (in rads/sec)
vr_temp =0;
vr_temp = temp[20]*256 + temp[21];
if (vr_temp>32768) vr_temp=vr_temp-65536;
*(y+8)=(real_T)(vr_temp/1000);

// Output Channel 10: Yaw Rate Bias (in rads/sec)
vr_temp =0;
vr_temp = temp[22]*256 + temp[23];
if (vr_temp>32768) vr_temp=vr_temp-65536;
*(y+9)=(real_T)(vr_temp/1000);

// Output Channel 11: X Axis Accel (in meters/sec^2)
vr_temp =0;
vr_temp = temp[24]*256 + temp[25];
if (vr_temp>32768) vr_temp=vr_temp-65536;
*(y+10)=(real_T)(vr_temp/100);

// Output Channel 12: Y Axis Accel (in meters/sec^2)
vr_temp =0;
vr_temp = temp[26]*256 + temp[27];
if (vr_temp>32768) vr_temp=vr_temp-65536;
*(y+11)=(real_T)(vr_temp/100);

// Output Channel 13: Z Axis Accel (in meters/sec^2)
vr_temp =0;
vr_temp = temp[28]*256 + temp[29];
if (vr_temp>32768) vr_temp=vr_temp-65536;
*(y+12)=(real_T)(vr_temp/100);


// Output Channel 14: Aileron 1 Command (in rad)
vr_temp =0;
vr_temp = temp[30]*256 + temp[31];
if (vr_temp>32768) vr_temp=vr_temp-65536;
*(y+13)=(real_T)(vr_temp/1000);

// Output Channel 15: Elevator 1 Command (in rad)
vr_temp =0;
vr_temp = temp[32]*256 + temp[33];
if (vr_temp>32768) vr_temp=vr_temp-65536;
*(y+14)=(real_T)(vr_temp/1000);

// Output Channel 16: Rudder 1 Command (in rad)
vr_temp =0;
vr_temp = temp[34]*256 + temp[35];
if (vr_temp>32768) vr_temp=vr_temp-65536;
*(y+15)=(real_T)(vr_temp/1000);

// Output Channel 17: Throttle 1 Command (in % of Full Throw)
vr_temp =0;
vr_temp = temp[36]*256 + temp[37];
if (vr_temp>32768) vr_temp=vr_temp-65536;
*(y+16)=(real_T)(vr_temp/100);

// Output Channel 18: Flap 1 Command (in rad)
vr_temp =0;
vr_temp = temp[38]*256 + temp[39];
if (vr_temp>32768) vr_temp=vr_temp-65536;
*(y+17)=(real_T)(vr_temp/100);

// Output Channel 19: Aileron 2 Command (in rad)
vr_temp =0;
vr_temp = temp[40]*256 + temp[41];
if (vr_temp>32768) vr_temp=vr_temp-65536;
*(y+18)=(real_T)(vr_temp/1000);

// Output Channel 20: Elevator 2 Command (in rad)
vr_temp =0;
vr_temp = temp[42]*256 + temp[43];
if (vr_temp>32768) vr_temp=vr_temp-65536;
*(y+19)=(real_T)(vr_temp/1000);

// Output Channel 21: Rudder 2 Command (in rad)
vr_temp =0;
vr_temp = temp[44]*256 + temp[45];
if (vr_temp>32768) vr_temp=vr_temp-65536;
```

```
       *(y+20)=(real_T)(vr_temp/1000);

       // Output Channel 22: Throttle 1 Command (in % of Full Throw)
       vr_temp =0;
       vr_temp = temp[46]*256 + temp[47];
       if (vr_temp>32768) vr_temp=vr_temp-65536;
       *(y+21)=(real_T)(vr_temp/100);

       // Output Channel 23: Flap 1 Command (in rad)
       vr_temp =0;
       vr_temp = temp[48]*256 + temp[49];
       if (vr_temp>32768) vr_temp=vr_temp-65536;
       *(y+22)=(real_T)(vr_temp/100);

       // Output Channel 24: Estimated Euler Roll Angle (in rad)
       vr_temp =0;
       vr_temp = temp[50]*256 + temp[51];
       if (vr_temp>32768)vr_temp=vr_temp-65536;
       *(y+23)=(real_T)vr_temp/1000;

       // Output Channel 25: Estimated Euler Pitch Angle (in rad)
       vr_temp =0;
       vr_temp = temp[52]*256 + temp[53];
       if (vr_temp>32768)vr_temp=vr_temp-65536;
       *(y+24)=(real_T)vr_temp/1000;

       // Output Channel 26: Estimated Euler Yaw Angle (in rad)
       vr_temp =0;
       vr_temp = temp[54]*256 + temp[55];
       if (vr_temp>32768)vr_temp=vr_temp-65536;
       *(y+25)=(real_T)vr_temp/1000;



}

//*****************************************************************
// File:        toplevelcrc.c
// Author(s):   Mariano I Lizarraga and Vladimir Dobrokhodov
// Date:        March-20-2003
// Description: This is the mdlOutputs of the S-Function that
//              computes the top level checksum used to wrap data
//              in the Piccolo Protocol
//*****************************************************************

static void mdlOutputs(SimStruct *S, int_T tid)
{
       int_T             i=0,j=0;
       InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);/* Incoming stream*/
       real_T            *y    = ssGetOutputPortRealSignal(S,0);


       unsigned __int8 temp[100]; // make an aliase for the uPtrs
       unsigned __int8 crc = 0,tmp, tmp1;
       int count=0;
       int len_in=*uPtrs[0];

       for (i=0; i<len_in; i++)
         {
            temp[i] = ( unsigned __int8)(*uPtrs[i+1]);
         }


       for (i=0;i<len_in;i++)
       {
          crc = (unsigned __int16)(crc^temp[i]);
       }
       *y=(real_T)crc;
}

//*****************************************************************
// File:        enc_apilot_loop_fix.c
// Author(s):   Mariano I Lizarraga and Vladimir Dobrokhodov
// Date:        June-10-2003
// Description: This is the mdlOutputs of the S-Function that
//              recieves the inner level stream, calculates the
//              CRC, appends it to the end and sends the data out
//              in the Piccolo Protocol inner level form
//*****************************************************************
static void mdlOutputs(SimStruct *S, int_T tid)
{
       // Declare the variables to be used in the function
       // counters
       int_T       i=0,j=0;
       int_T     temp[50]={0}; // make an alias for the uPtrs
       char      ext[]="\0";
       int       count=0,temp_crc=0, crc=0, CRCHi=0, CRCLo=0;
       float       vr_temp = 0.0;


       // Incoming data stream,
       // *uPtrs[0]- determines the length of useful bytes{15 in this case}
       InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);
       real_T            *y    = ssGetOutputPortRealSignal(S,1);

       // read the whole incoming string and assign it to a temporary variable
       // in order to avoid mistakenly overwrting the incoming data
       for (i=0; i<(*uPtrs[0]); i++){temp[i] = ( int_T)(*uPtrs[i+1]);}


       // %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
       //   Start the Decoding Procedure
       // %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

       // ===============
       // Compute the CRC
       // ===============
       crc=0;
       for (j=0;j<(*uPtrs[0]);j++)
       {
          // read the value to a temp variable
          temp_crc = temp[j];

          // perform the crc calculation
          crc = (unsigned __int16)(crctable2[(temp_crc^crc) & 0xff] ^ (crc >> 8));
       }

       // ======================
       // Split CRC in two bytes
       // ======================

       vr_temp = crc/256.0;
       CRCHi= (int)floor(vr_temp);

       vr_temp = vr_temp-CRCHi;
       CRCLo = (int)floor(vr_temp*256);


       // ===============
       // Output the data
       // ===============

       // Pass the input directly to the output
       // SYNC
       // SYNC1
       // Packet Type
       // Size of Payload
       // Payload
       for (j=0;j<(*uPtrs[0]);j++)
       {
          *(y+j)=(real_T)temp[j];
       }

       // Write the CRC bytes

       *(y+j) = (real_T)CRCHi;
       *(y+j+1) = (real_T)CRCLo;


       // Issue done pulse to outport 0
       ssCallSystemWithTid(S, 0, 0);


}

//*****************************************************************
// File:        enc_top_level_26.c
// Author(s):   Mariano I Lizarraga and Vladimir Dobrokhodov
// Date:        June-10-2003
// Description: This is the mdlOutputs of the S-Function that
//              recieves the inner level packet, calculates the
//              CRC, appends it to the end and sends the data out
//              in the Piccolo Protocol form
//*********************************************************************
static void mdlOutputs(SimStruct *S, int_T tid)
{
       // Declare the variables to be used in the function
       // counters
       int_T       i=0,j=0;
       int_T     temp[100]={0}; // make an alias for the uPtrs
       char      ext[]="\0";
       int       count=0,temp_crc=0, crc=0, CRCHi=0, CRCLo=0;
       float       vr_temp = 0.0;


       // Incoming data stream,
       // *uPtrs[0]- determines the length of useful bytes{15 in this case}
       InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);
       real_T            *y    = ssGetOutputPortRealSignal(S,1);

       // read the whole incoming string and assign it to a temporary variable
       // in order to avoid mistakenly overwrting the incoming data
       for (i=0; i<(*uPtrs[0]); i++)
       {temp[i] = ( int_T)(*uPtrs[i+1]);}


       // %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
       //   Start the Decoding Procedure
       // %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

       // ===============
       // Compute the CRC
       // ===============
       crc=0;
       for (j=0;j<(*uPtrs[0]);j++)
       {
          // read the value to a temp variable
          temp_crc = temp[j];

          // perform the crc calculation
          crc = (unsigned __int16)(crctable3[(temp_crc^crc) & 0xff] ^ (crc >> 8));
       }

       // ======================
       // Split CRC in two bytes
       // ======================

       vr_temp = crc/256.0;
       CRCHi= (int)floor(vr_temp);

       vr_temp = vr_temp-CRCHi;
       CRCLo = (int)floor(vr_temp*256);


       // ===============
       // Output the data
       // ===============

       // Pass the input directly to the output
       // SYNC
       // SYNC1
       // Packet Type
       // Size of Payload
       // Payload
       for (j=0;j<(*uPtrs[0]);j++)
       {
          *(y+j)=(real_T)temp[j];
       }

       // Write the CRC bytes

       *(y+j) = (real_T)CRCHi;
       *(y+j+1) = (real_T)CRCLo;


       // Issue done pulse to outport 0
       ssCallSystemWithTid(S, 0, 0);


}

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

```
// File:        gpsrcv_v2.c
// Author(s):   Mariano I Lizarraga and Vladimir Dobrokhodov
// Date:        August-30-2003
// Description: This is the mdlOutputs of the S-Function that
//              recieves data from the GPS and sends to the output
//              the GGA and RMC messages
//***************************************************************
/* Function to compute outputs */
static void mdlOutputs(SimStruct *S, int_T tid)
{
//#ifndef MATLAB_MEX_FILE
       int width   = (int)mxGetPr(OUT_WIDTH_ARG)[0];

       unsigned char tmp;                               /* temp char holder */
       unsigned char *buf = (unsigned char *)ssGetDWork(S, 0);       // uchar buffer to contain bytes
       int *current   = ssGetIWork(S);      /* current = addr of current position pointer in buf */
       int *recLength = ssGetIWork(S) + 1;  /* recLength = addr of receieved data length */
       int *bufCount = ssGetIWork(S)+ 2;    /* count number of useful bytes in buf. */
       int  serbufCount;             /* count number of useful bytes collected in Serial buf*/
       int  HeaderFound, i, j, bufStop, chksum=0, nextbytetoprocess=0, lastHeaderPos=0, EOB;
       int  GGA=358,RMC=377;// checksum of 'GPGGA','GPRMC' sentences's header
       int  bl_header;// boolean values for GGA=1 and RMC=2 sentences, 0=nothing found
       int  headwidth=5;// length of GPS header except '$'
       int  ggalng=87,rmclng=73, enough_bytes =1;// initialize length of GGA (87) and RMC(73)
       int_T temp[1024]={0};
       int *head_ptr = &bl_header;

       InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);

       // Get the number of bytes available from the port
       serbufCount = (int)mxGetPr(IN_WIDTH_ARG)[0];

       for (i=0; i<serbufCount; i++){temp[i] = ( int_T)(*uPtrs[i]);}

       *bufCount = serbufCount + *current;
       // place new chunk of data at the end of previously unprocessed data
       // 'buf' also keeps the remains of previous step ending at the *(buf+current),
       // where 'current' is the position of last byte
       for (i=*current;i<*bufCount;i++)
       {
           *(buf+i)=(unsigned char)temp[i-*current];
       }

       i=0;


       /*Initialize logical flags*/
       HeaderFound = 0;
       i = 0;
       EOB = 0;//end of buffer

       //if (*bufCount<width) return;    // Not enough bytes to decode, output old value

       // while no header has been found
       while ((HeaderFound==0) && (EOB !=1) && (enough_bytes ==1))
       {         /* find Header byte = '$'=36 */

               chksum =0;
              //printf("\n%d %d %d" , *bufCount,i, EOB);
              // while no header has been found
              while ((*(buf+i) != HEADER) && (EOB != 1))
               {

                  // if you have enough bytes
                  if (i < *bufCount)
                  {
                     // increment position indx
                     i++;
                  }
                  // if you ran out of bytes
                  else
                  {
                     // turn on the flag
                     EOB = 1;
                  }
              }

              //printf("\n%d %d %d" , *bufCount,i, EOB);
              // if the remaining amount of data read from the port is less
              // then the amount of data I want to write to the output
              if (*bufCount - i < width-1)
```

```
      {
         // turn the flag
         enough_bytes = 0;
      }

      // if I did not reach the End Of Buffer and have enough bytes
      // to write
      if ((EOB!=1) && (enough_bytes == 1) )
      {
         //printf("\n Header Found i = %d" , i);
         // initialize the chechsum
         chksum = 0;

         // grab GPXXX
         for (j=1;j<headwidth+1;j++)
         {
            chksum = chksum + *(buf+i+j);
         }

         //printf("\n Checksum = %d" , chksum);
          if  (chksum == GGA)
         {
            // turn on the flag
            HeaderFound = 1;

            // turn on the boolean flag
            bl_header=1;

            // intialize the length of the GGA message
            ggalng=0;

            // while we do not find a 13 (CR)
            //"0D"=13(ASCII),
            while(*(buf+i+headwidth+1+ggalng) != 13)
            {
               // increment the counter
               ggalng++;
            }

            // once we found the CR
            // copy to the outputport ggalng bytes
            memcpy(ssGetOutputPortSignal(S,1),buf+i+headwidth+1,ggalng);

            // increment the position index to reflect all the bytes read
            i = i + ggalng;
         } // if checksum tally
         else
         {
            if (chksum == RMC)
            {
               HeaderFound = 1;
               bl_header=2;
               rmclng = 0;
               while(*(buf+i+headwidth+1+rmclng) != 13)
               {
                  rmclng++;//"0D"=13(ASCII),
               }
               //              printf("rmc length = %d", rmclng);
               memcpy(ssGetOutputPortSignal(S,1),buf+i+headwidth+1,rmclng);
               i = i + rmclng;
            } // if checksum tally
            else
            {
               // checksum doesn't tally
               i=i+5;
               bl_header=0;
            }// end of checksum searching
         }

         memcpy(ssGetOutputPortSignal(S,2),head_ptr,1);// ? chek to be ","=44

      }// if not EOB and there is enough bytes

  } /* while HeaderFound = 0*/



// if I did not reach the EOB looking for a header but I ran out of bytes to send
// then copy the remains to the beggining of buffer
if ((EOB != 1) || (enough_bytes == 0))
{
```

```
         *bufCount=*bufCount-i;  //number of remain bytes
         // buf[j]=buf[i]
         for (j=0;j<*bufCount; j++)
         {
            *(buf+j)=*(buf+i);
            i++;
         }//  end of for
         *current=*bufCount;
      }

      ssCallSystemWithTid(S, 0, 0); /* issue done pulse to outport 0 */
    return;

//#endif
}

//*****************************************************************
// File:        gpgga.c
// Author(s):   Mariano I Lizarraga and Vladimir Dobrokhodov
// Date:        August-30-2003
// Description: This is the mdlOutputs of the S-Function that
//              recieves data from the gpsrcv_v2 S-Function and
//              decodes the GGA message
//*****************************************************************
static void mdlOutputs(SimStruct *S, int_T tid)
{
    int_T            i=0,j=0;
    InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);/* Incoming data*/
     real_T            *y    = ssGetOutputPortRealSignal(S,0);
    /*int_T             width = ssGetOutputPortWidth(S,0);*/


      int_T tempbuf[100]; //here we make an aliase for the uPtrs
      char ext[]="\0";
      int count=0,len_in;
      double sys[20];//output array of decoded data
      double res[1]={0}; //output piece of decoded data
      real_T tmp;

      for (i=0; i<(*uPtrs[0]); i++){tempbuf[i] = ( int_T)(*uPtrs[i+1]);}


     //GPGGA sentence of GPS message
     i=0;
     if ((real_T) tempbuf[i] != (real_T) 44)
     { /*printf("\n Error! 44 expected, received %d",*tempbuf);*/
        return;}
     else count++; /*miss first comma sign and define shift*/

     /* UTC & Latitude -1,2*/
     for(i=0;i<2;i++)
     {
     len_in=0;/*initialize it again*/
     while (tempbuf[count+len_in] != 44)
     {++len_in;}//end of while to count the length of GPS field
     bin2ascii((tempbuf+count),len_in,ext);
        *y++=(real_T)atof(ext);
     *ext=NULL;/*initialize it again*/
     count+=len_in+1;//miss next comma sign and define new shift
     }/*end of for*/


     /*Direction of latitude -3*/
     len_in=0;/*initialize it again*/
     while (tempbuf[count+len_in] != 44)
     {++len_in;}//end of while to count the length of GPS field
     *y++=gpssmbl((tempbuf+count),len_in);
     count+=len_in+1;//miss next comma sign and define new shift

     /*Longitude -4*/
     len_in=0;/*initialize it again*/
     while (tempbuf[count+len_in] != 44)
     {++len_in;}//end of while to count the length of GPS field
     bin2ascii((tempbuf+count),len_in,ext);
     *y++=(real_T)atof(ext);
     *ext=NULL;/*initialize it again*/
     count+=len_in+1;//miss next comma sign and define new shift

     /*Direction of longitude -5*/
     len_in=0;/*initialize it again*/
     while (tempbuf[count+len_in] != 44)
     {++len_in;}//end of while to count the length of GPS field
     *y++=gpssmbl((tempbuf+count),len_in);
     count+=len_in+1;//miss next comma sign and define new shift

     /*GPS quality indicator -6;
     Number of SVs      -7;
     HDOP           -8;
     Antenna height     -9*/

     for(i=0;i<4;i++)
     {
     len_in=0;/*initialize it again*/
     while (tempbuf[count+len_in] != 44)
     {++len_in;}//end of while to count the length of GPS field
     bin2ascii((tempbuf+count),len_in,ext);
     *y++=(real_T)atof(ext);
     *ext=NULL;/*initialize it again*/
     count+=len_in+1;//miss next comma sign and define new shift
     }/* end of for*/

     /*Altitude in meters -10*/
     len_in=0;/*initialize it again*/
     while (tempbuf[count+len_in] != 44)
     {++len_in;}//end of while to count the length of GPS field
     *y++=gpssmbl((tempbuf+count),len_in);
     count+=len_in+1;//miss next comma sign and define new shift

     /*Geoidal separation -11*/
     len_in=0;/*initialize it again*/
     while (tempbuf[count+len_in] != 44)
     {++len_in;}//end of while to count the length of GPS field
     bin2ascii((tempbuf+count),len_in,ext);
     *y++=(real_T)atof(ext);
     *ext=NULL;/*initialize it again*/
     count+=len_in+1;//miss next comma sign and define new shift

     /*Geoidal separation in meters -12*/
     len_in=0;/*initialize it again*/
     while (tempbuf[count+len_in] != 44)
     {++len_in;}//end of while to count the length of GPS field
     *y++=gpssmbl((tempbuf+count),len_in);
     count+=len_in+1;//miss next comma sign and define new shift

     /*Age of DGPS data -13*/
     len_in=0;/*initialize it again*/
     while (tempbuf[count+len_in] != 44)
     {++len_in;}//end of while to count the length of GPS field
     bin2ascii((tempbuf+count),len_in,ext);
     *y++=(real_T)atof(ext);
     *ext=NULL;/*initialize it again*/
     count+=len_in+1;//miss next comma sign and define new shift

     /*Base station ID-14*/
     len_in=0;/*initialize it again*/
     while (tempbuf[count+len_in] != 42)//42 ='*' It's the beginning
     {++len_in;}//end of while to count the length of GPS field
     bin2ascii((tempbuf+count),len_in,ext);
     *y++=(real_T)atof(ext);
     *ext=NULL;/*initialize it again*/
     count+=len_in+1;//miss next comma sign and define new shift

}

//*******************************************************************
// File:        gprmc.c
// Author(s):   Mariano I Lizarraga and Vladimir Dobrokhodov
// Date:        August-30-2003
// Description: This is the mdlOutputs of the S-Function that
//              recieves data from the gpsrcv_v2 S-Function and
//              decodes the RMC message
//*******************************************************************
static void mdlOutputs(SimStruct *S, int_T tid)
{
    int_T             i=0,j=0;
    InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);/* Incoming stream*/
     real_T            *y    = ssGetOutputPortRealSignal(S,0);
    /*int_T             width = ssGetOutputPortWidth(S,0);*/


       int_T tempbuf[100]; //here we make an aliase for the uPtrs
       char ext[]="\0";
```

```
  int count=0,len_in;
  double sys[20];//output array of decoded data
  double res[1]={0}; //output piece of decoded data
  real_T tmp;

 for (i=0; i<(*uPtrs[0]); i++){tempbuf[i] = ( int_T)(*uPtrs[i+1]);;}


//GPRMC sentence of GPS message
i=0;
if ((real_T) tempbuf[i] != (real_T) 44) return;
else count++; /*miss first comma sign and define shift*/

/* UTC -1*/
len_in=0;/*initialize it again*/
while (tempbuf[count+len_in] != 44)
{++len_in;}//end of while to count the length of GPS field
bin2_ascii((tempbuf+count),len_in,ext);
*y++=(real_T)atof(ext);
*ext=NULL;/*initialize it again*/
count+=len_in+1;//miss next comma sign and define new shift


/*Status -2*/
len_in=0;/*initialize it again*/
while (tempbuf[count+len_in] != 44)
{++len_in;}//end of while to count the length of GPS field
*y++=gps_smbl((tempbuf+count),len_in);
count+=len_in+1;//miss next comma sign and define new shift

/*Latitude -3*/
len_in=0;/*initialize it again*/
while (tempbuf[count+len_in] != 44)
{++len_in;}//end of while to count the length of GPS field
bin2_ascii((tempbuf+count),len_in,ext);
*y++=(real_T)atof(ext);
*ext=NULL;/*initialize it again*/
count+=len_in+1;//miss next comma sign and define new shift

/*Latitude direction -4*/
len_in=0;/*initialize it again*/
while (tempbuf[count+len_in] != 44)
{++len_in;}//end of while to count the length of GPS field
*y++=gps_smbl((tempbuf+count),len_in);
count+=len_in+1;//miss next comma sign and define new shift

   /*Longitude -5*/
len_in=0;/*initialize it again*/
while (tempbuf[count+len_in] != 44)
{++len_in;}//end of while to count the length of GPS field
bin2_ascii((tempbuf+count),len_in,ext);
*y++=(real_T)atof(ext);
*ext=NULL;/*initialize it again*/
count+=len_in+1;//miss next comma sign and define new shift

/*Direction of logitude -6*/
len_in=0;/*initialize it again*/
while (tempbuf[count+len_in] != 44)
{++len_in;}//end of while to count the length of GPS field
*y++=gps_smbl((tempbuf+count),len_in);
count+=len_in+1;//miss next comma sign and define new shift

/*Speed over ground[knots] -7*/
len_in=0;/*initialize it again*/
while (tempbuf[count+len_in] != 44)
{++len_in;}//end of while to count the length of GPS field
bin2_ascii((tempbuf+count),len_in,ext);
*y++=(real_T)atof(ext);
*ext=NULL;/*initialize it again*/
count+=len_in+1;//miss next comma sign and define new shift

/*Track made good,True[degree] -8*/
len_in=0;/*initialize it again*/
while (tempbuf[count+len_in] != 44)
{++len_in;}//end of while to count the length of GPS field
bin2_ascii((tempbuf+count),len_in,ext);
*y++=(real_T)atof(ext);
*ext=NULL;/*initialize it again*/
count+=len_in+1;//miss next comma sign and define new shift

   /*Date in  dd/mm/yy       -9;
```

```
    Manetic variation [degree]      -10;*/

    for(i=0;i<2;i++)
    {
    len_in=0;/*initialize it again*/
    while (tempbuf[count+len_in] != 44)
    {++len_in;}//end of while to count the length of GPS field
    bin2_ascii((tempbuf+count),len_in,ext);
    *y++=(real_T)atof(ext);
    *ext=NULL;/*initialize it again*/
    count+=len_in+1;//miss next comma sign and define new shift
    }/* end of for*/

    /*Direction of Magnetic variation-11*/
    len_in=0;/*initialize it again*/
    while (tempbuf[count+len_in] != 44)
    {++len_in;}//end of while to count the length of GPS field
    *y++=gps_smbl((tempbuf+count),len_in);
    count+=len_in+1;//miss next comma sign and define new shift


    /*Mode indicator(A(4)-autonomous;D(5)-differencial;N(0)-not valid) -12*/
    len_in=0;/*initialize it again*/
    while (tempbuf[count+len_in] != 42)/*=42*/
    {++len_in;}//end of while to count the length of GPS field
    *y++=gps_smbl((tempbuf+count),len_in);
    count+=len_in+1;//miss next comma sign and define new shift


}
```

APPENDIX B: DERIVATION OF THE 6-DOF AERODDYNAMIC MODEL OF THE SILVER FOX UAV

This appendix presents a detailed derivation of the 6-DOF model of the Silver Fox UAV.

A. 6-DOF AIRCRAFT MODEL

The derivation of the equations of motion for a 6-DOF aircraft model to be used as a plant in the control system design was done in two steps. The first step was formulating the equations of motion for the aircraft, treated as a rigid body. The second step was the computation of aerodynamic, gravitational and propulsive forces that act upon the aircraft's body. The aerodynamic and propulsive forces are specific to the aircraft to be modeled and depend directly on the airfoil, aircraft geometry, mass dispersion, and engine characteristics.

1. Equations of Motion

In the following subsections, the development of the equations of motion for the aircraft's linear and angular dynamics in the local reference frame *{L}* are developed.

a. Linear Dynamics Equations

The equations for linear motion are governed by Newton's second law, which states that the net force applied to the center of mass of a body is equal to the product of the mass times the acceleration. However, aircraft velocities, accelerations, linear forces and attitude angles are usually measured with respect to the aircraft's body axis coordinate system since the sensors are strapped down to the aircraft's body (fuselage).

Because of this, a new reference frame is required, a frame that is attached to the aircraft's body center of gravity, hence called the *Body* frame *{B}*. This frame has its *X* axis defined as coming out of the nose of the airplane; its *Y* axis as pointing to the right wing; and the *Z* axis as orthogonal to these two, thus pointing straight down the fuselage.

It is known from elementary kinematics that:

$$\vec{v}_{UAV} = \dot{\vec{p}}_{uav}. \tag{B1}$$

Equation (B1) can be expressed in the local reference frame *{L}* simply by premultiplying it by a rotation matrix, as previously described,

$$\dot{\vec{p}}_{uav} = {}^{L}_{B}R \; {}^{B}\vec{v}_{UAV}, \tag{B2}$$

where the rotation matrix ${}^{B}_{L}R$ is derived in [2], written below for reference:

$${}^{B}_{L}R = \begin{bmatrix} \cos\theta\cos\psi & \cos\theta\sin\psi & -\sin\theta \\ \sin\phi\sin\theta\cos\psi - \cos\phi\sin\psi & \sin\phi\sin\theta\sin\psi + \cos\phi\cos\psi & \sin\phi\cos\theta \\ \cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi & \cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi & \cos\phi\cos\theta \end{bmatrix}, \tag{B3}$$

and

$${}^{L}_{B}R = \left({}^{B}_{L}R\right)^{T}. \tag{B4}$$

The total derivative of a vector ${}^{B}\vec{v}$, which is rotating with angular velocity ${}^{B}\vec{\omega}$ with respect to *{L}*, is given by Coriolis' theorem [8] by:

$$\frac{d}{dt_L}\,{}^{B}\vec{v} = {}^{B}\dot{\vec{v}} + {}^{B}\vec{\omega} \times {}^{B}\vec{v}, \tag{B5}$$

where $\frac{d}{dt_L}$ denotes the total derivative in *{L}*. Therefore, the derivative of vector ${}^{B}\vec{v}$ with respect to *{L}* has two components; the total derivative in its frame and a component that is nonzero when it is rotating.

Hence, the rigid body's linear acceleration is given by:

$$\frac{d}{dt_L}\,{}^{B}\vec{v}_{UAV} = {}^{B}\dot{\vec{v}}_{UAV} + {}^{B}\vec{\omega}_{UAV} \times {}^{B}\vec{v}_{UAV}. \tag{B6}$$

Therefore, Newton's second law applied to the aircraft's center of gravity can be written as:

$$^{B}\vec{F}_{UAV} = m\frac{d}{dt_L}\,{}^{B}\vec{v}_{UAV}. \tag{B7}$$

The above equation can be rewritten by substituting Equation (B6) into Equation (B7) as:

$$^{B}\vec{F}_{UAV} \quad = \quad m\left({}^{B}\dot{\vec{v}}_{UAV} + {}^{B}\vec{\omega}_{UAV} \times {}^{B}\vec{v}_{UAV}\right). \tag{B8}$$

Note that equations (B7) and (B8) consider the mass to be constant. Generally this is not the case. However, since a typical landing approach for a tactical UAV such as the Silver Fox only lasts about two minutes, the change on mass in that time is considered to be negligible.

b. Angular Dynamics Equations

The angular equations of motion are derived using Euler's law for conservation of angular momentum in *{L}*. Let ${}^{L}\vec{G}_{UAV}$ denote the moment imparted to the rigid body and $\frac{d}{dt}\dot{\vec{H}}_{UAV}$ denote the rate of change of the angular momentum. Therefore [2]:

$$\vec{G}_{UAV} = \frac{d}{dt}\vec{H}_{UAV}. \tag{B9}$$

Using equations (B3) and (B5), Equation (B9) can be rewritten as:

$$\frac{d}{dt_L}\,{}^{B}\vec{H}_{UAV} = {}^{B}\vec{G}_{UAV} = {}^{B}\dot{\vec{H}}_{UAV} + {}^{B}\vec{\omega}_{UAV} \times {}^{B}\vec{H}_{UAV}, \tag{B10}$$

where the angular momentum can be rewritten in terms of the moments and cross products of inertia as [8]:

$${}^{B}\vec{H}_{UAV} = \begin{bmatrix} J_{xx} & -J_{xy} & -J_{xz} \\ -J_{xy} & J_{yy} & -J_{yz} \\ -J_{xz} & -J_{yz} & J_{zz} \end{bmatrix} {}^{B}\vec{\omega}_{UAV} = J\,{}^{B}\vec{\omega}_{UAV}. \tag{B11}$$

However, since the UAV is symmetric with respect to its *XZ* plane, all the *YZ* and *XY* cross products of inertia are cancelled, simplifying the inertia matrix *J* to:

$$J = \begin{bmatrix} J_{xx} & 0 & -J_{xz} \\ 0 & J_{yy} & 0 \\ -J_{xz} & 0 & J_{zz} \end{bmatrix}. \qquad \text{(B12)}$$

Equation (B10) can now be rewritten as:

$${}^{B}\vec{G}_{UAV} = J\,{}^{B}\dot{\vec{\omega}}_{UAV} + {}^{B}\vec{\omega}_{UAV} \times \left(J\,{}^{B}\vec{\omega}_{UAV}\right). \qquad \text{(B13)}$$

c. Attitude (Euler) Equations

A vector resolved in *{L}* can be expressed in *{B}* simply by premultiplying it by the rotation matrix shown in Equation (B3) as:

$${}^{B}\vec{v} = {}^{B}_{L}R\vec{v}. \qquad \text{(B14)}$$

If, instead of using Coriolis' law to obtain the total derivative of the vector expressed in Equation (B14), the rules of calculus are used, then the vector can be expressed as:

$$\frac{d}{dt_L}\left({}^{B}\vec{v}\right) = {}^{B}_{L}R\dot{\vec{v}}, \qquad \text{(B15)}$$

but, if the total derivative is taken in *{B},* then the product rule for differentiation must be applied to obtain:

$$\frac{d}{dt_B}\left({}^{B}\vec{v}\right) = {}^{B}_{L}\dot{R}\vec{v} + {}^{B}_{L}R\dot{\vec{v}}, \qquad \text{(B16)}$$

which, if it is solved for ${}^{B}_{L}R\dot{\vec{v}}$ and substituted in Equation (B15), gives:

$$\frac{d}{dt_L}\left({}^{B}\vec{v}\right) = {}^{B}\dot{\vec{v}} - {}^{B}_{L}\dot{R}\vec{v}. \qquad \text{(B17)}$$

Comparing the above equation with Equation (B5) suggests that a relation between the rate of change of the rotation matrix ${}^{B}_{L}R$ and the angular velocity vector ${}^{B}\vec{\omega}$ exists. To find this relationship, let the vector cross product be expressed as a product of a matrix and a vector as:

$$
{}^{B}\vec{\omega}\times{}^{B}\vec{v}=\begin{bmatrix}0 & -r & q\\ r & 0 & -p\\ -q & p & 0\end{bmatrix}{}^{B}\vec{v}={}^{B}\Omega^{B}\vec{v}, \tag{B18}
$$

where

$$
{}^{B}\vec{\omega}=\begin{bmatrix}p\\ q\\ r\end{bmatrix}. \tag{B19}
$$

By substituting Equation (B14) into Equation (B18) and comparing the second terms of equations (B5) and (B17), the following equality can be established:

$$
-{}^{B}\Omega{}^{B}_{L}R\vec{v}={}^{B}_{L}\dot{R}\vec{v}, \tag{B20}
$$

or,

$$
{}^{B}_{L}\dot{R}=-{}^{B}\Omega{}^{B}_{L}R. \tag{B21}
$$

The above equation is known as the *strapdown equation* [8]. Equating elements (1,2), (1,3), and (2,3) of the above matrix equation (where (*i* ,*j*) represents the element in the ith row and the jth column) gives a set of three equations that allows one to compute the rate of change of the Euler angles as follows:

$$
\begin{aligned}
\dot{\varphi}&=p+q\left(\sin\phi+r\cos\phi\right)\tan\theta\\
\dot{\theta}&=q\cos\phi-r\sin\phi\\
\dot{\psi}&=\sec\theta\left(q\sin\phi+r\cos\theta\right).
\end{aligned} \tag{B22}
$$

In summary, the equations to be implemented in the rigid body 6-DOF model of the Silver Fox are given in Equations (B8), (B13), and (B22).

2. Forces and Moments on Aircraft

The forces and the moments exerted on the aircraft are due to the aerodynamic, propulsion, and gravitational effects on the body. They can be expressed as:

$$\begin{bmatrix} {}^B\vec{F}_{UAV} \\ {}^B\vec{G}_{UAV} \end{bmatrix} = \begin{bmatrix} {}^B\vec{F}_{aero} + {}^B\vec{F}_{prop} + {}^B\vec{F}_{grav} \\ {}^B\vec{G}_{aero} + {}^B\vec{G}_{prop} \end{bmatrix}. \tag{B23}$$

a. Aerodynamic Forces and Moments

The aerodynamic forces and moments that act upon an aircraft are produced by the relative motion of the aircraft with respect to the air mass (the airspeed). Mathematically, the aerodynamic forces and moments terms are determined by using a first-order Taylor series expansion around the aircraft trimmed operating point.

Each term in the series is a partial derivative of ${}^B\vec{F}_{UAV}$ and ${}^B\vec{G}_{UAV}$ with respect to the aerodynamic variables. These aerodynamic variables are given by the deflection of the elevator, aileron, and rudder.

Let the angle of attack α be the angle between the projection of the airspeed vector to the *XZ* body plane and the *X* body axis. Also let the sideslip angle β, be the angle between the projection of the airspeed vector to the *XY* body plane and the *X* body axis.

Since the forces and moments depend on the airspeed, a new coordinate frame must be introduced. In this new coordinate frame, denoted as *{W}*, the *X* axis is aligned with the wind's velocity vector ($\vec{W}$). The rotation matrix ${}^B_W R$, which rotates a vector from *{W}* to *{B},* is given in [8]:

$${}^B_W R = \begin{bmatrix} \cos\alpha\cos\beta & -\cos\alpha\sin\beta & -\sin\alpha \\ \sin\beta & \cos\beta & 0 \\ \sin\alpha\cos\beta & -\sin\alpha\sin\beta & \cos\alpha \end{bmatrix}. \tag{B24}$$

Equation (B23) can now be rewritten as:

$$\begin{bmatrix} {}^{B}\vec{F}_{UAV} \\ {}^{B}\vec{G}_{UAV} \end{bmatrix} = \begin{bmatrix} {}_{W}^{B}R\,{}^{W}\vec{F}_{aero} + {}^{B}\vec{F}_{prop} + {}^{B}\vec{F}_{grav} \\ {}_{W}^{B}R\,{}^{W}\vec{G}_{aero} + {}^{B}\vec{G}_{prop} \end{bmatrix}. \tag{B25}$$

For simplicity, the forces and moments acting on the aircraft are defined in terms of dimensionless aerodynamic coefficients that, as expected, depend on the control surfaces deflection, the aerodynamic angles α and β, and the angular rates. The forces and moments expressed in terms of the dimensionless aerodynamic coefficients are derived in [8] and shown in Equation (B26) for easy reference.

$$\begin{bmatrix} {}^{W}\vec{F}_{UAV} \\ {}^{W}\vec{G}_{UAV} \end{bmatrix} = qS \begin{bmatrix} \begin{bmatrix} C_D \\ C_Y \\ C_L \end{bmatrix} \\ \begin{bmatrix} C_l b \\ C_m c \\ C_n b \end{bmatrix} \end{bmatrix}, \tag{B26}$$

where *q* is the dynamic pressure given by:

$$q = \frac{\rho \left| {}_{W}^{B}R\left(\vec{v}_{UAV} - \vec{W}\right)\right|^{2}}{2}, \tag{B27}$$

the variable S is the wing reference area, given by:

$$S = cb, \tag{B28}$$

the variable ρ is the air density, *c* is the wing chord, and *b* is the wingspan.

The numerical values for the dimensionless aerodynamic coefficients required by Equation (B26) were obtained using LinAir Pro, developed by Desktop Aeronautics Inc., which is a software tool that employs a panel or a multi-element method to compute the aircraft's aerodynamic characteristics. Later this chapter briefly explains how LinAir Pro produced the above coefficients.

b. Gravitational and Propulsive Forces and Moments

Gravitational forces act on the rigid body but generate no moments since the forces are assumed to act at the center of gravity. The gravitational forces are resolved in *{L}* and are given by:

$$\vec{F}_{grav} = \begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix}, \tag{B29}$$

where *m* is the mass of the airplane and *g* is the acceleration of gravity. Equation (B29) can be resolved in *{B}* simply by premultiplying it by the appropriate rotation matrix:

$$^{B}\vec{F}_{grav} = {}^{B}_{L}R \begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix}. \tag{B30}$$

Propulsive forces and moments are exerted in *{B}* and can be stated as:

$$^{B}\vec{F}_{prop} = \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix}, \tag{B31}$$

and

$$^{B}\vec{G}_{prop} = \begin{bmatrix} P_l \\ P_m \\ P_n \end{bmatrix}, \tag{B32}$$

where each of the scalars *P* in Equations (B31) and (B32) represent forces or moments due to the aircraft's thrust.

Since in the aircraft being modeled, the Silver Fox UAV, the engine thrust axis coincides with the *X* axis of *{B}*, then the thrust projections P_y and P_z are zero.

In summary, Equation (B25) can be rewritten as:

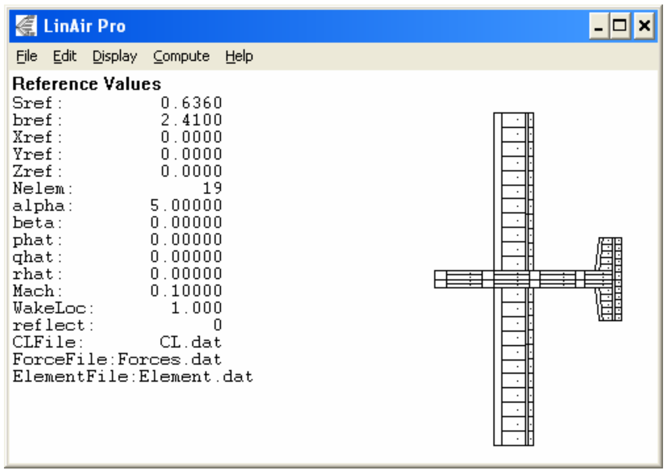
$$ {}^{B}\vec{F}_{UAV} = qS\,{}^{B}_{W}R\begin{bmatrix} C_D \\ C_Y \\ C_L \end{bmatrix} + \begin{bmatrix} P_x \\ 0 \\ 0 \end{bmatrix} + {}^{B}_{L}R\begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix}, \tag{B33}$$

$$ {}^{B}\vec{G}_{UAV} = qS\,{}^{B}_{W}R\begin{bmatrix} C_l b \\ C_m c \\ C_n b \end{bmatrix}. \tag{B34}$$

3. Dimensionless Aerodynamic Coefficients

As was previously mentioned, the values of the dimensionless aerodynamic coefficients are obtained from a Taylor series expansion about a trim condition. Typically these values are experimentally obtained in a wind tunnel. Nevertheless, software packages exist that allow one to obtain reasonably good approximations of wind-tunnel experiments. LinAir Pro is one of these software packages. LinAir Pro computes the aerodynamic characteristics of multi-element, nonplanar lifting surfaces.

In order to have LinAir Pro produce the desired results, first one must create an *input file*, which contains the geometry of the airplane divided in panels, shown in Figure B 1.



```
Reference Values
Sref :          0.6360
bref :          2.4100
Xref :          0.0000
Yref :          0.0000
Zref :          0.0000
Nelem:              19
alpha:         5.00000
beta :         0.00000
phat :         0.00000
qhat :         0.00000
rhat :         0.00000
Mach :         0.10000
WakeLoc:         1.000
reflect:             0
CLFile:         CL.dat
ForceFile:Forces.dat
ElementFile:Element.dat
```

Figure B 1. Main Configuration Window of LinAir Pro. Top View of the Modeled Silver Fox UAV.

Once the input file is correctly configured, the Prandtl-Glauert equation is internally solved, and the force and moment contribution of each panel is computed [9]. Figure B 2 shows one of the result screens produced by LinAir Pro. It shows how each panel contributes to the lift of the aircraft.

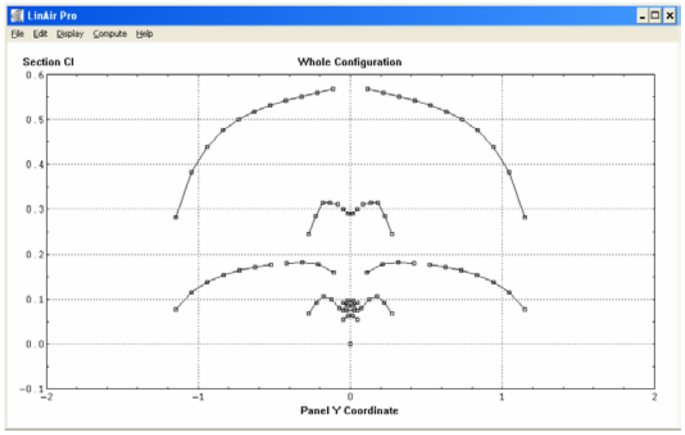


Figure B 2. Nondimensionalized Section Lift Coefficient vs. Panel Y Coordinate. LinAir Pro.

LinAir Pro was ran with different scenarios, in which the deflection of the control surfaces were first varied from –15° to 15° and then different values for α and β were used for each run. All the data were collected and analyzed with Microsoft Excel to produce curve fittings of the data.

The following equations were used to compute the value of the dimensionless aerodynamic coefficients required by Equations (B33) and (B34) . The numerical values of the intermediate coefficients, which were obtained with LinAir Pro, are shown in the MATLAB script file listed in Appendix C.

$$\begin{aligned} C_D &= C_{D0} + A_{polar} C_L + A_2 C_L{}^2 \\ C_Y &= C_{Y\beta}\beta + C_{Y\delta_r}\delta_r \qquad \text{(B35)} \\ C_L &= C_{L0} + C_{L\alpha}\alpha + C_{L\dot{\alpha}}\dot{\alpha} + C_{L\delta_e}\delta_e + \frac{C_{Lq}q}{\left\| {}^B\vec{V}_{UAV} \right\|}, \end{aligned}$$

$$
\begin{aligned}
C_l &= C_{l\beta}\beta + C_{l\delta_a}\delta_a + C_{l\delta_r}\delta_r + \frac{b}{2\left\| {}^B\vec{v}_{UAV} \right\|}\left(C_{lp}p + C_{lr}r\right) \\
C_m &= C_{m0} + C_{m\alpha}\alpha + C_{m\delta_e}\delta_e + C_{m\dot{\alpha}}\dot{\alpha} + \frac{C_{mq}qc}{\left\| {}^B\vec{v}_{UAV} \right\|} \\
C_n &= C_{n\beta}\beta + C_{n\delta_a}\delta_a + C_{n\delta_r}\delta_r + \frac{b}{2\left\| {}^B\vec{v}_{UAV} \right\|}\left(C_{np}p + C_{nr}r\right),
\end{aligned}
\tag{B36}
$$

where $\delta_e, \delta_a, \delta_r$ represent the deflection of the elevator, aileron, and rudder respectively: p, q, r are the angular rates, b is the wingspan, c is the wing's mean aerodynamic chord, and α and β represent the angle of attack and the sideslip angle, respectively.

4. Engine Model

The mathematical model of the UAV's engine required for the 6-DOF model needed to reflect the relationship between the throttle commanded by the pilot manual control or the ground station, and the thrust exerted to the aircraft's body expressed in Newtons.

This relationship was found in two steps. The first step consisted of an experiment in which the UAV, sitting on the ground and linked to a spring scale, was forced to accelerate its engine from 4,000 to 9,000 Revolutions Per Minute (RPM) as measured by a digital tachometer. The readings were plotted in a chart to compare thrust vs. RPM and a linear fit was obtained, as shown in Figure B 3. This fit is known as the *static thrust* of the aircraft. In this experiment it was determined that the commanded thrust was directly proportional to the RPM scaled by a factor of 8,700 lbs/RPM.

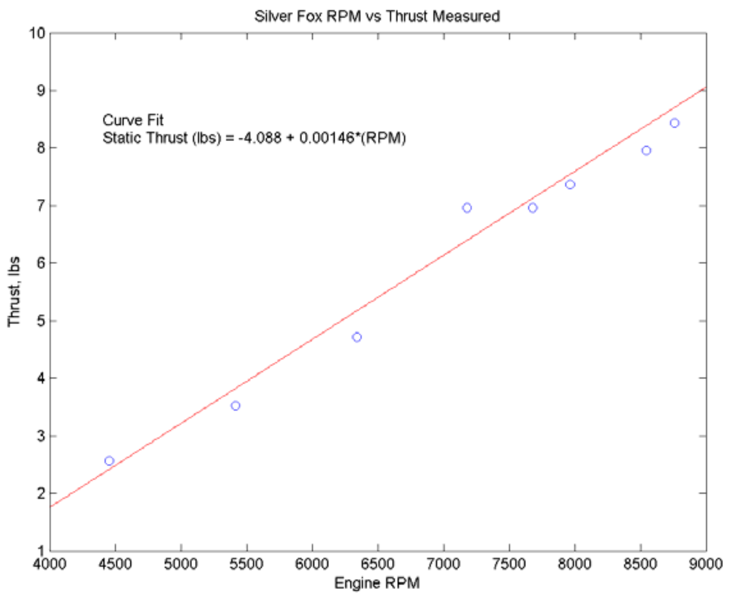


Figure B 3. Silver Fox's Static Thrust Diagram.

Having the static thrust, the second step was to obtain the thrust available, which can be predicted based on momentum theory. In [10] it is shown that the relationship between static thrust and thrust available is given by:

$$\frac{T}{T_o}\left\{\frac{\left|\vec{V}_{UAV}\right|}{w_o}+\sqrt{\left(\frac{\left|\vec{V}_{UAV}\right|}{w_o}\right)^2+4\frac{T}{T_o}}\right\}=1, \tag{B37}$$

where T_o is the static thrust, T is the thrust available,

$$w_o=\sqrt{\frac{T_o}{2\rho A}}, \tag{B38}$$

and A is the propeller disk area. The propeller used in this particular test was an APC 14x8 propeller, where 14 stands for the length in inches, and 8 stands for the propeller pitch in inches.

A plot of the relationship shown in Equation (B37) is shown in Figure B 4.

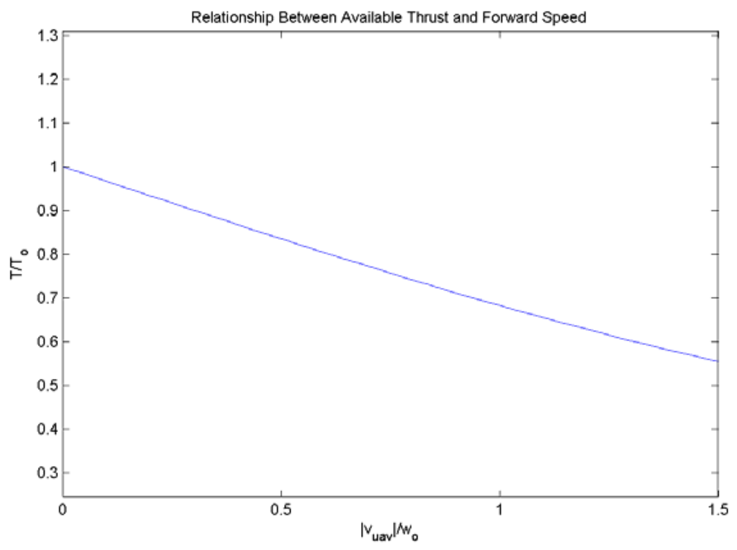


Figure B 4. Variation of Thrust Available with Forward Speed. (From Ref. [10].)

A straight-line approximation, using a standard sea level air density ρ, with the propeller described, results in the thrust available in pounds:

$$T = T_o - 0.0238\left|\vec{v}_{UAV}\right|\sqrt{T_o}. \tag{B39}$$

Figure B 5 shows the Simulink implementation of the engine model and introduces several conversions to match the appropriate units.

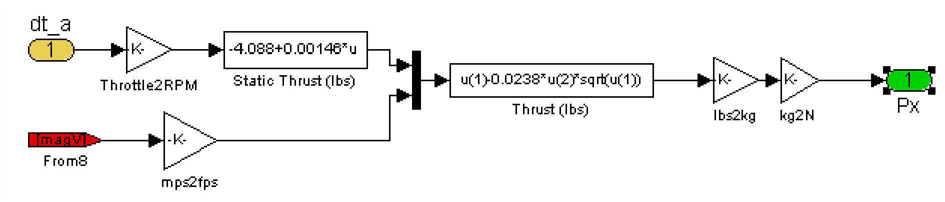


Figure B 5. Simulink Implementation of the Engine Model.

5. Atmosphere and Wind Model

As seen in Equation (B27), the dynamic pressure depends on the air density. To model the variation of the air density correctly with respect to the altitude of the aircraft, the equations adopted by the United States Committee on Extension to the Standard Atmosphere (COESA) were used. The equations and parameters used are documented in [11] and Simulink provides a readily available block, which has as input the altitude in meters and provides, as one of the outputs, the air density in Kg/m^3.

For the wind, real data was used which was previously collected at the U.S. Army's Yuma Proving Grounds (YPG), Yuma, AZ, for the work presented in [12]. The data is indexed by height in meters and is accessed in Simulink using lookup tables. Figure B 6 shows the complete Simulink diagram of the atmosphere and wind model.

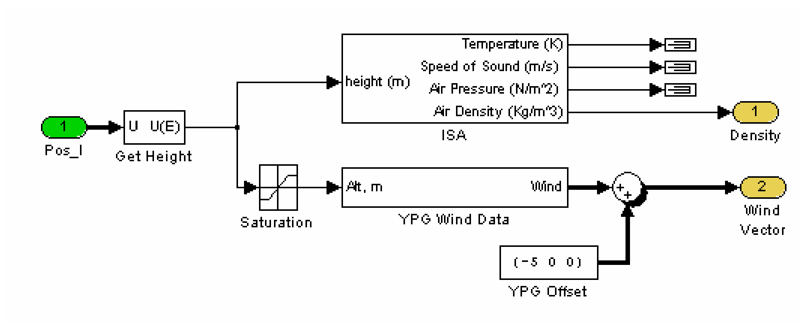


Figure B 6. Atmosphere and Wind Model in Simulink.

A complete Simulink model of the Silver Fox, including the equations of motion, the forces and moments, the engine model and the atmosphere and wind model is shown in Figure B 7.

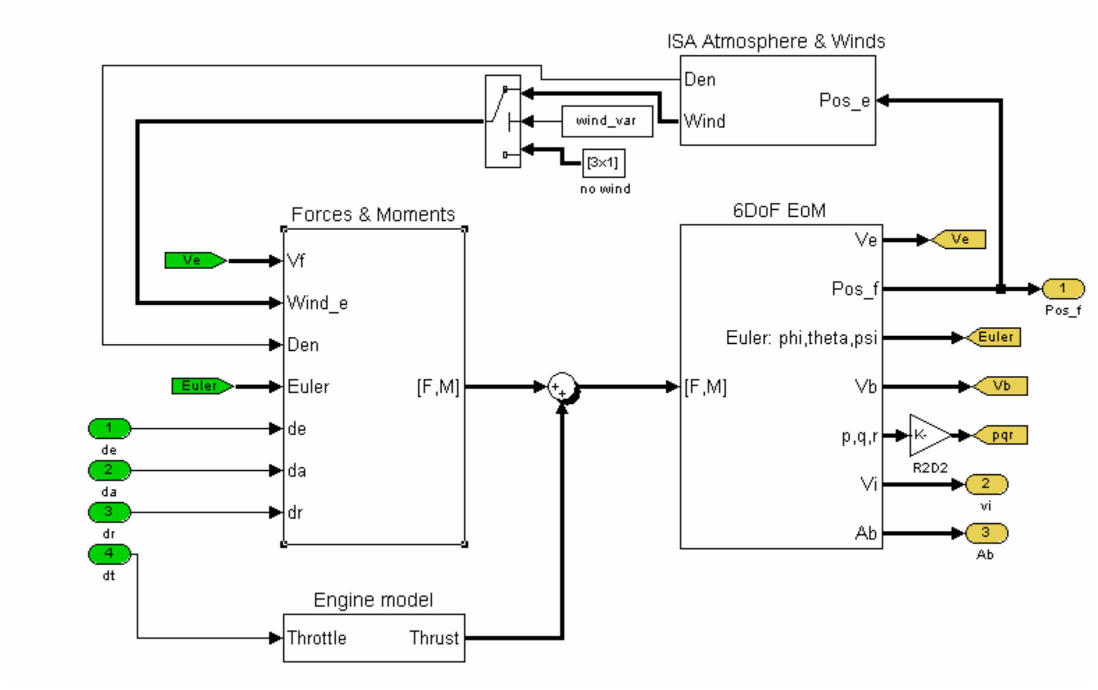


Figure B 7. Complete Simulink Model of the Silver Fox.

6. Actuators Models

The control surfaces onboard the UAV are moved by small servomechanisms, similar to the ones shown in Figure B 8.

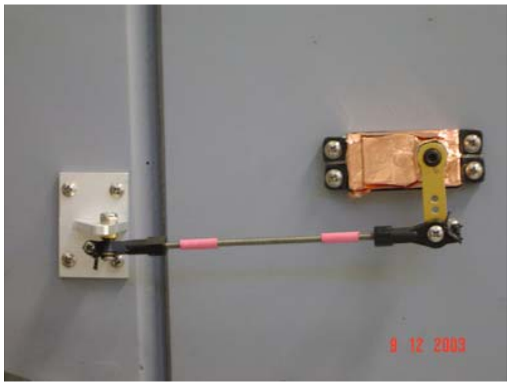


Figure B 8. Servomechanism (Actuator) on a UAV.

These mechanisms have two main limitations, the rate at which they can change the position of the surface, and the maximum deflection they can impose on the surface. Because of this, the actuators were modeled as second order systems with rate limiters and position limiters. The parameters chosen to model the actuators were a natural frequency of 50 rad/sec and a damping ratio of 1. Thus, the actuators were modeled as low pass filters with a cutoff frequency of 5 Hz. The Simulink diagram that models the UAV’s actuators is shown in the following figure.

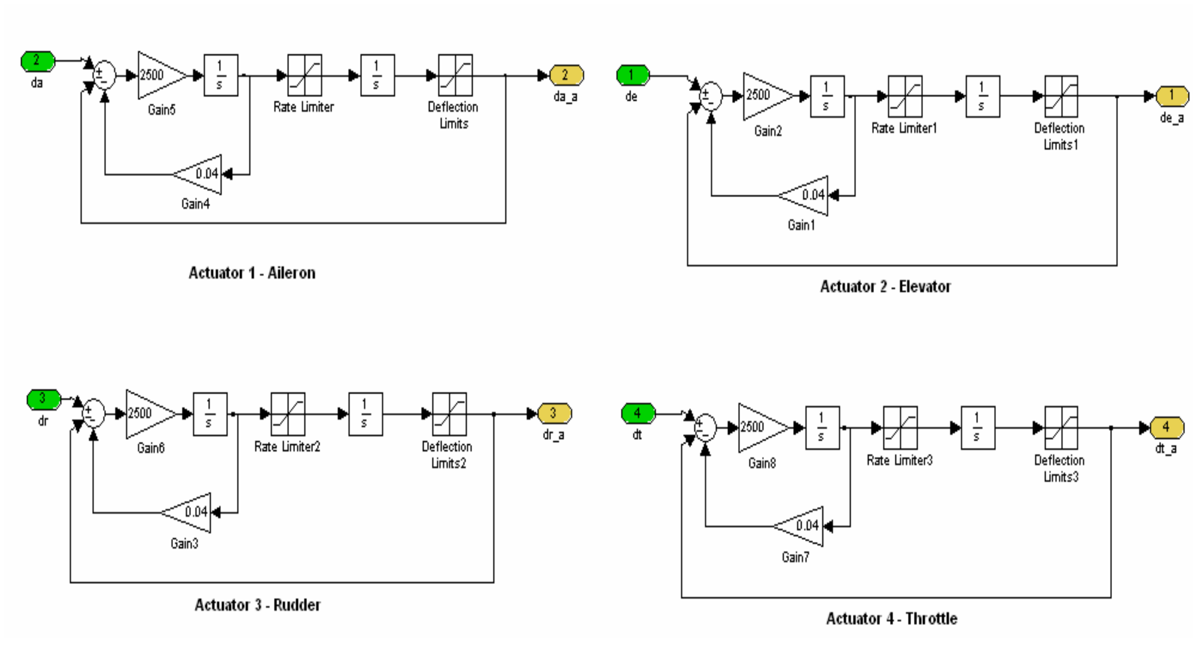


Figure B 9.   Aileron, Elevator, Rudder and Throttle Actuators. Simulink Model.

APPENDIX C: NUMERICAL VALUES OF THE AERODYNAMIC COEFFICIENTS OF THE SILVER FOX 6DOF MODEL

The following MATLAB Script file sets all the initial conditions for the computer simulation and includes all the values for the aerodynamic coefficients of the Silver Fox 6-DOF model.

```
% ======================================================
% File:        fox_load_variables_new.m
% Author(s):   Oleg Yakimenko
% Date:        April-10-2003
% Description: This script file intializes all the variables
%              required for the Silver Fox 6DOF model
% ======================================================

clear all;
clc;

%  Initial Conditions in ENU (all vector data is represented as a column vectors)
Pos_0   = [0; 0; 300.]'*0.3048;       % Initial position vector (m)
Vb_0    = [85.; 0; 0]'*0.3048;        % Initial velocity vector (m/s)
Euler_0 = [0; 0; 0]'*pi/180;          % Initial Euler angles    (rad)
PQR_0   = [0; 0; 0]';                 % Initial Omega           (rad/s)

% Mass and Geometric Parameters
S     = 8.;                 % surface area of wing  (ft^2)
span  = 8.;                 % wingspan              (ft)
chord = 1.;                 % chord                 (ft)
mass  = 20;                 % gross weight          (lbs)
Ixx   = 0.268;              % main moment of inertia around axis Ox (slug*sq.ft)
Iyy   = 0.640;              % main moment of inertia around axis Oy (slug*sq.ft)
Izz   = 0.884;              % main moment of inertia around axis Oz (slug*sq.ft)

% Mass and Geometric Parameters recomputation
S     = S*0.3048^2;         % surface area of wing  (m2)
span  = span*0.3048;        % wingspan              (m)
chord = chord*0.3048;       % chord                 (m)
mass = mass*0.45359237;     % gross weight          (kg)
Ixx   = Ixx*1.355817949;    % main moment of inertia around axis Ox (kg*sq.m)
Iyy   = Iyy*1.355817949;    % main moment of inertia around axis Oy (kg*sq.m)
Izz   = Izz*1.355817949;    % main moment of inertia around axis Oz (kg*sq.m)

% Aerodynamic Derivatives (all per radian)
CL0     = 3*0.076;        % lift coefficient at a = 0
CLa     = 5.097;        % lift curve slope
CLa_dot = 1.93;         % lift due to angle of attack rate
CLq     = 6.03;         % lift due to pitch rate
CLDe    = 0.738;        % lift due to elevator
CD0     = 0.0191;       % drag coefficient at a = 0
Apolar  = 0.038;        % drag curve slope
CYb     = -0.204;       % side force due to sideslip
CYDr    = 0.112;        % sideforce due to rudder
Clb     = -0.0598;      % dihedral effect
Clp     = -0.363;       % roll damping
Clr     = 0.0886;       % roll due to yaw rate
ClDa    = 0.265;        % roll control power
ClDr    = 0.0064;       % roll due to rudder
Cm0     = 0.107;        % pitch moment at a = 0
Cma     = -2.051;       % pitch moment due to angle of attack
Cma_dot = -5.286;       % pitch moment due to angle of attack rate
Cmq     = -16.52;       % pitch moment due to pitch rate
CmDe    = -2.021;       % pitch control power
Cnb     = 0.0562;       % weathercock stability
Cnp     = -0.0407;      % adverse yaw
Cnr     = -0.0439;      % yaw damping
CnDa    = -0.0296;      % aileron adverse yaw
CnDr    = -0.0377;      % yaw control power

% Standartd Atmosphere
```

```
ISA_lapse = .0065;          % Lapse rate            (degC/m)
ISA_hmax  = 2000;           % Altitude limit        (m)
ISA_R     = 287;            % Gas Constant          (degK*m*m/s/s)
ISA_g     = 9.815;          % Gravity               (m/s/s)
ISA_rho0  = 1.225;          % Density at sea level  (kg/m/m/m)
ISA_P0    = 101325;         % Sea-level Pressure    (N/m/m)
ISA_T0    = 289;            % Sea-level Temperature (degK)

%  Load Wind Profile
load YPGwind.mat;
```

LIST OF REFERENCES

[1] James Stewart, *Calculus Early Transcedentals, Fourth Edition,* Brooks/Cole, Pacific Grove, CA, 1999.

[2] Bernard Etkin, and Lloyd Duff Reid, *Dynamics of Flight, Third Edition*, John Wiley & Sons, New York, NY, 1995.

[3] Isaac Kaminer, Antonio Pascoal, Eric Hallberg, and Carlos Silvestre, "Trajectory Tracking for Autonomous Vehicles: An Integrated Approach to Guidance and Control," *Journal of Guidance, Control and Dynamics*, Vol. 21, No. 1, pp. 29-38, January – February 1998.

[4] William Vaglienti, and Ross Hoag, *Communications for the Piccolo Avionics*, *version 1.1.6*, Hood River, OR, 2003.

[5] Mathworks, The, *XPC Target User's Guide*, Nantick, MA, 2003.

[6] Trimble, *agGPS114 Receiver User's Guide*, Overland Park, KA, 2003.

[7] Yaakov Bar-Shalom, Xiao Rong-Li, and Thiagalingam Kirubarajan, *Estimation with Applications to Tracking and Navigation*, John Wiley & Sons, New York, NY, 2001.

[8] Brian L. Stevens, and Frank L. Lewis, *Aircraft Control and Simulation*, Wiley & Sons, New York, NY, 1992.

[9] Ilan Kroo, *A Nonplanar*, *Multiple Lifting Surface Aerodynamics Program. LinAir Pro Version 3.4*, Desktop Aeronautics, Stanford, CA, 1997.

[10] B. W. McCormick, *Aerodynamics of V/STOL Flight*, Academic Press, 1967.

[11] Government Printing Office, *U.S. Standard Atmosphere*, Washington, D.C., 1976.

[12] Vladimir N. Dobrokhodov, Oleg A. Yakimenko, and Christopher J. Junge, "Six-Degree-of-Freedom Model of Controlled Circular Parachute," *Journal of Aircraft*, Vol. 40, No. 3, pp. 482-493, May-June 2003.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Chairman, Code EC
   Dept. of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

4. Dr. Isaac Kaminer, Dept. of Mechanical and Astronautical Engineering
   Naval Postgraduate School
   Monterey, California

5. Dr. Roberto Cristi, Dept. of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

6. Dr. Robert G. Hutchins, Dept. of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

7. Dr. Vladimir Dobrokhodov, Dept. of Mechanical and Astronautical
   Engineering
   Naval Postgraduate School
   Monterey, California

8. Dr. Oleg Yakimenko, Dept. of Mechanical and Astronautical Engineering
   Naval Postgraduate School
   Monterey, California