NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

DESIGN AND EVALUATION OF A DIGITAL FLIGHT CONTROL SYSTEM FOR THE FROG UNMANNED AERIAL VEHICLE

by

Christopher H. Flood

September 2001

Thesis Advisor:	Isaac I. Kaminer
Second Reader:	Oleg Yakimenko

Approved for public release; distribution is unlimited

Report Documentation Page

Report Date	Report Type	Dates Covered (from... to)
30 Sep 2001	N/A	-

Title and Subtitle Design and Evaluation of a Digital Flight Control System for the FROG Unmanned Aerial Vehicle	**Contract Number**
	Grant Number
	Program Element Number
Author(s) Flood, Christopher H.	**Project Number**
	Task Number
	Work Unit Number
Performing Organization Name(s) and Address(es) Research Office Naval Postgraduate School Monterey, Ca 93943-5138	**Performing Organization Report Number**
Sponsoring/Monitoring Agency Name(s) and Address(es)	**Sponsor/Monitor's Acronym(s)**
	Sponsor/Monitor's Report Number(s)
Distribution/Availability Statement Approved for public release, distribution unlimited	
Supplementary Notes	
Abstract	
Subject Terms	
Report Classification unclassified	**Classification of this page** unclassified
Classification of Abstract unclassified	**Limitation of Abstract** UU
Number of Pages 130	

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2001	**3. REPORT TYPE AND DATES COVERED** Aeronautical Engineers Thesis

4. TITLE AND SUBTITLE: Design and Evaluation of a Digital Flight Control System for the FROG Unmanned Aerial Vehicle

5. FUNDING NUMBERS

6. AUTHOR(S) Flood, Christopher H.

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)***

The importance of unmanned aerial vehicles (UAVs) to current and future military operations cannot be understated. This rapidly developing field requires the ability to quickly develop and evaluate advanced control concepts. The FROG UAV serves as a test bed for advanced control and sensor projects at the Naval Postgraduate School. Previous control system projects have made use of a low performance electromechanical autopilot onboard the UAV. This autopilot imposed significant limitations on the responsiveness of the FROG. This project developed and tested an off board digital flight control system for use in lieu of the previous electromechanical device.

The digital flight controller was developed using the $Matrix_X$ rapid prototyping system and a previously validated dynamic model of the FROG. Surrogate flight control servo actuators were characterized in the laboratory and added to the plant model. Classic inner / outer loop controllers were developed for yaw damping and speed, altitude and heading control. The system was then successfully demonstrated with hardware in the loop in the lab. The FROG was then instrumented and a command uplink latency of 170 ms was discovered. This introduced excessive phase lag into the system, which drove the flight controllers unstable. An alternate serial uplink method was developed and tested which reduced the command latency to 76 ms however the remaining phase lag resulted in limit cycle oscillation. Laboratory tests indicated that the current flight controller could withstand a maximum of 50 ms command path delay; without modification.

14. SUBJECT TERMS Manned Aerial Vehicles, UAV, Frog UAV, Digital Flight Controller	**15. NUMBER OF PAGES** 130
	16. PRICE CODE

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

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

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

DESIGN AND EVALUATION OF A DIGITAL FLIGHT CONTROL SYSTEM FOR THE FROG UNMANNED AERIAL VEHICLE

Christopher H. Flood
Commander, United States Navy
B.S., United States Naval Academy, 1984

Submitted in partial fulfillment of the
requirements for the degree of

AERONAUTICAL AND ASTRONAUTICAL ENGINEER

from the

NAVAL POSTGRADUATE SCHOOL
September 2001

Author: _____
Christopher H. Flood

Approved by: _____
Isaac I. Kaminer, **Thesis Advisor**

_____
Oleg Yakimenko, **Second Reader**

_____
Max F. Platzer, Chairman
Department of Aeronautics and Astronautics

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The importance of unmanned aerial vehicles (UAVs) to current and future military operations cannot be understated. This rapidly developing field requires the ability to quickly develop and evaluate advanced control concepts. The FROG UAV serves as a test bed for advanced control and sensor projects at the Naval Postgraduate School. Previous control system projects have made use of a low performance electromechanical autopilot onboard the UAV. This autopilot imposed significant limitations on the responsiveness of the FROG. This project developed and tested an off board digital flight control system for use in lieu of the previous electromechanical device.

The digital flight controller was developed using the $Matrix_X$ rapid prototyping system and a previously validated dynamic model of the FROG. Surrogate flight control servo actuators were characterized in the laboratory and added to the plant model. Classic inner/outer loop controllers were developed for yaw damping and speed, altitude and heading control. The system was then successfully demonstrated with hardware in the loop in the lab. The FROG was then instrumented and a command uplink latency of 170 ms was discovered. This introduced excessive phase lag into the system, which drove the flight controllers unstable. An alternate serial uplink method was developed and tested which reduced the command latency to 76 ms however the remaining phase lag resulted in limit cycle oscillation. Laboratory tests indicated that the current flight controller could withstand a maximum of 50 ms command path delay; without modification.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION 1

II. FROG UAV SYSTEM 3
- **A. DESCRIPTION OF THE AIRPLANE 3**
- **B. FROG RAPID FLIGHT TEST PROTOTYPING SYSTEM 5**
  - **1. $MATRIX_X$ Rapid Prototyping System 6**
  - **2. AC-104 Controller Configuration 8**
  - **3. Command Uplink 9**
  - **4. Sensor Downlink 12**
  - **5. Electromagnetic Interference 14**
- **C. NPS INERTIAL MEASUREMENT UNIT 15**
  - **1. Sensor Description 15**
  - **2. Embedded Micro Controller Development 19**
    - ***a. 3DM Tattletale Program 22***
    - ***b. GPS TattleTale Program 24***
- **D. CROSSBOW ATTITUDE HEADING REFERENCE SET 27**
  - **1. Hardware Description 27**
  - **2. Timing Performance 30**
  - **3. AHRS Noise Output 41**

III. DIGITAL FLIGHT CONTROLLER 43
- **A. SERVO TEST SET CHARACTERIZATION 43**
- **B. DIGITAL FLIGHT CONTROLLER DESIGN 50**
  - **1. Design Methodology and Performance Criteria 50**
  - **2. Yaw Damper 52**
  - **3. Altitude Controller 54**
  - **4. Heading Controller 56**
  - **5. Airspeed Controller 58**
  - **6. Control Mode Coupling 60**
- **C. FROG CONTROL SERVOS 61**
  - **1. FROG Servo Configuration and Instrumentation 61**
  - **2. Control Surface Position Calibration 64**
  - **3. FROG Servo Dynamics 70**
  - **4. FROG Digital Flight Controller Performance 73**
- **D. ALTERNATE COMMAND UPLINK 74**

IV. CONCLUSIONS AND RECOMMENDATIONS 79
- **A. CONCLUSIONS 79**
- **B. RECOMMENDATIONS 79**

APPENDIX A. NPS IMU OPERATING SOFTWARE 81
- **A. 3DM TATTLETALE OPERATING CODE 81**
- **B. 3DMIO.H 90**

C. DATA3DM.H.....93
D. PWM.H 94
E. GPS TATTLETALE OPERATING CODE.....95
F. GPSPARSE.H.....99
G. GPSDATA.H 108
H. ATODDDATA.H.....109
APPENDIX B. CROSSBOW AHRS NOISE ANALYSIS 111
LIST OF REFERENCES 113
INITIAL DISTRIBUTION LIST.....115

LIST OF FIGURES

Figure 2.1. FROG UAV. 3
Figure 2.2. FROG UAV 3 View Drawing. 4
Figure 2.3. FROG Engine Configuration. 4
Figure 2.4. FROG Rapid Flight Test Prototyping System. 6
Figure 2.5. AC-104 Real-Time Controller. 8
Figure 2.6. AC-104 Interface Layout. 9
Figure 2.7. Futaba FP-9ZAP Digital Proportional Radio Control. 10
Figure 2.8. FROG Master/Slave Transmitter Arrangement. 10
Figure 2.9. Futaba® Receiver to Servo Connections from Ref. [1]. 12
Figure 2.10. NPS IMU Downlink Architecture. 13
Figure 2.11. Crossbow AHRS Downlink Architecture. 14
Figure 2.12. BEI Gyrochip™ Rate Gyro 16
Figure 2.13. 3DM™ 3-Axis Orientation Sensor. 18
Figure 2.14. Humphrey VG-34-0201-2 Vertical Gyroscope. 18
Figure 2.15. TattleTale Model 8 Micro-Controller. 20
Figure 2.16. FROG Onboard Computer Functional Architecture after Ref. [2]. 21
Figure 2.17. Crossbow AHRS400CA-100 Attitude Heading Reference System. 27
Figure 2.18. Crossbow AHRS System Architecture. 28
Figure 2.19. Gyroview Software User Interface. 31
Figure 2.20. Gyroview Real-Time Data Display Capability. 32
Figure 2.21. Gyroview Flight Test Data (Level Figure-8 Maneuver). 33
Figure 2.22. Gyroview Flight Test Data (Level Figure-8 Maneuver). 34
Figure 2.23. AHRS Angle Mode Serial Output Signal. 35
Figure 2.24. AHRS Continuous/Angle Mode Serial Output. 35
Figure 2.25. AHRS Polled / Angle Mode Serial Output Response Variation. 36
Figure 2.26. AHRS Polled/Angle Mode Response - 20 Hz Polling Rate. 37
Figure 2.27. AHRS Polled/Angle Mode Response - 30 Hz Polling Rate. 38
Figure 2.28. AHRS Polled/Angle Mode Response - 60 Hz Polling Rate. 39
Figure 2.29. AHRS Continuous/Scaled Sensor Mode Serial Output. 40
Figure 2.30. AHRS Polled / Scaled Sensor Mode Serial Output Response Variation. 41
Figure 3.1 Servo Test Set. 44
Figure 3.2. Control Systems Laboratory Hardware in the Loop System. 45
Figure 3.3. SystemBuild Servo Dynamics Test System. 46
Figure 3.4. Servo Test System Calibration GUI. 47
Figure 3.5. TS-75 Servo Response to 10° STEP Input. 48
Figure 3.6. Second Order Model of TS-75 Servo. 48
Figure 3.7. RealSim Speed Controller GUI. 49
Figure 3.8. TS-75 Response to Increasing Speed Controller Feedback. 50
Figure 3.9. FROG SystemBuild Flight Controller Model. 51
Figure 3.10. RealSim Flight Controller Display. 52
Figure 3.11. Yaw Damper SystemBuild Block Diagram. 53

Figure 3.12. Yaw Damper Response (Simulated Actuators). 53
Figure 3.13. FROG Yaw Damper Performance with TS-75 Actuators. 54
Figure 3.14. Altitude Controller SystemBuild Block Diagram. 55
Figure 3.15. Altitude Controller Response with TS-75 Servos. 56
Figure 3.16. Heading Controller System Build Block Diagram. 57
Figure 3.17. Heading Controller Response With Simulated Actuators. 57
Figure 3.18. Heading Controller Response with TS-75 Servos. 58
Figure 3.19. Speed Controller SystemBuild Block Diagram. 59
Figure 3.20. Speed Controller Response with TS-75 Servos. 59
Figure 3.21. Longitudinal Control Mode Coupling. 60
Figure 3.22. Lateral-Directional Control Mode Coupling. 61
Figure 3.23. FROG Elevator Servo Configuration. 63
Figure 3.24. Series 150 Subminiature Position Transducer. 64
Figure 3.25. The FROG Command Uplink Signal Path. 65
Figure 3.26. Laser Installation for Control Surface Deflection Measurement. 67
Figure 3.27. Cylindrical Laser Target. 67
Figure 3.28 Laser Position Indicator Spot on Angle Scale. 68
Figure 3.29. FROG Aileron Position Command versus D/A Output Value. 69
Figure 3.30. FROG Aileron String Pot Output versus Position. 69
Figure 3.31. FROG Servo Response Delay. 71
Figure 3.32. TS-75 Servo Test Set Response Delay. 72
Figure 3.33. FROG Servo-based Speed Controller Performance. 73
Figure 3.34. Flight Controller with 170 ms Delay. 74
Figure 3.35. FP-R309DPS Receiver Output Signal. 75
Figure 3.36. Flight Controller Performance with 75 ms Delay. 77
Figure 3.37. Servo Phase Lag with 75 ms Command Delay. 78

LIST OF TABLES

Table 2.1. FROG UAV Physical Characteristics. 5
Table 2.2. NPS IMU Sensor Performance Limits. 19
Table 2.3. NPS IMU Downlink Data List. 22
Table 2.4. 3DM Magnetometer & Accelerometer Data Format. 23
Table 2.5. TattleTale Downlink Message Characteristics. 27
Table 2.6. Sensor Mode Serial Data Parameters. 29
Table 3.1. FROG Control Servo Configuration. 62
Table 3.2. FROG Servo Gains. 72
Table 3.3. FP-309DPS Pulse Width Response to FP-9ZAP Controller Commands. 76

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank my wife, Tina. Without your steadfast support, limitless patience and countless scarifies I would never have completed this work. This thesis is as much yours as it is mine. To my son, Kevin, I give my thanks. You have been a great help to your mother and me and have shown maturity far beyond your years. I hope you remember this time in Monterey fondly. To Scott and Brian, who are too young to understand why Daddy has to go back to work after diner, I can play now.

I would also like to thank my advisor, Dr. Isaac Kaminer, for the guidance, and freedom to learn. You have shown me that the mark of an educated man is more than the mere facts he knows but instead is the manner in which he reasons.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

The importance of unmanned aerial vehicles (UAVs) to current and future military operations cannot be understated. This rapidly developing field requires the ability to quickly develop and evaluate advanced control concepts. The FROG UAV serves as a test bed for advanced control and sensor projects at the Naval Postgraduate School. Previous control system projects have made use of a low performance electromechanical autopilot onboard the UAV. This autopilot imposed significant limitations on the responsiveness of the FROG. This thesis developed and tested an off board digital flight control system for use in lieu of the previous electromechanical device.

The digital flight controller was developed using the $Matrix_X$ rapid prototyping system and a previously validated dynamic model of the FROG. Surrogate flight control servo actuators were characterized in the laboratory and added to the plant model. Classic inner/outer loop controllers were developed for yaw damping and speed, altitude and heading control. The system was then successfully demonstrated with hardware in the loop in the lab. The FROG was then instrumented and a command uplink latency of 170 ms was discovered. This introduced excessive phase lag into the system, which drove the flight controllers unstable. An alternate serial uplink method was developed and tested which reduced the command latency to 76 ms however the remaining phase lag resulted in limit cycle oscillation. Laboratory tests indicated that the current flight controller could withstand a maximum of 50 ms command path delay, without modification.

THIS PAGE INTENTIONALLY LEFT BLANK

II. FROG UAV SYSTEM

A. DESCRIPTION OF THE AIRPLANE

The FROG unmanned aerial vehicle (UAV) is a small high wing monoplane used for digital control system research by the Naval Postgraduate School Aeronautics Department. The airplane was manufactured by BAI Aerosystems, as the BAI TERN (Tactically Expendable Remote Navigator), and was formerly designated the FOG-R by the U.S. Army. In the FOG-R configuration the airplane was equipped with a fiber optic data link for command uplink and video downlink. The TERN was designed to carry up to twenty-two pounds of payload for periods of up to four hours. The TERN UAV is currently in use as a test bed for sensor systems by both the US Navy's Strike UAV Program and NASA. In the past, the NPS FROG had been configured with a variety of sensors including an onboard autopilot, various inertial measurement units, GPS receivers, an instrumented nose boom and a digital camera. The FROG is depicted in Figures 2.1 and 2.2.

Figure 2.1. FROG UAV.

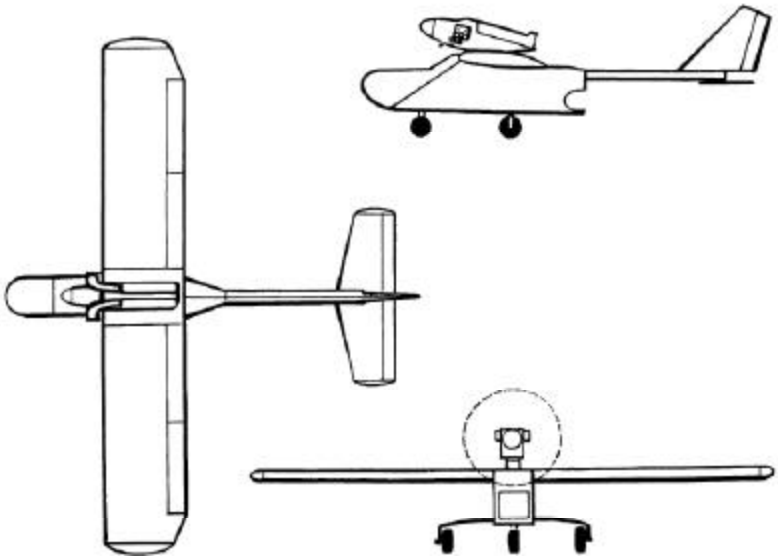

Figure 2.2. FROG UAV 3 View Drawing.

The FROG is configured with a Model BA64 6.4 cubic inch, horizontally opposed, piston engine, manufactured by Brinson Aircraft Company. The 2-cylinder engine developed 9.3 Hp and is equipped with a two bladed propeller mounted in a tractor orientation in a nacelle atop the wing, as depicted in Figure 2.3. The FROG has fixed tricycle landing gear with a steer-able nose wheel. The empennage is connected to the body of the airplane by a 1.75-inch diameter aluminum tube. The FROG is equipped with conventional elevator, rudder, ailerons and flaps. Small servomotors, designed for use in radio-controlled airplanes, actuate the control surfaces. The FROG's significant physical characteristics are presented in Table 1.1.

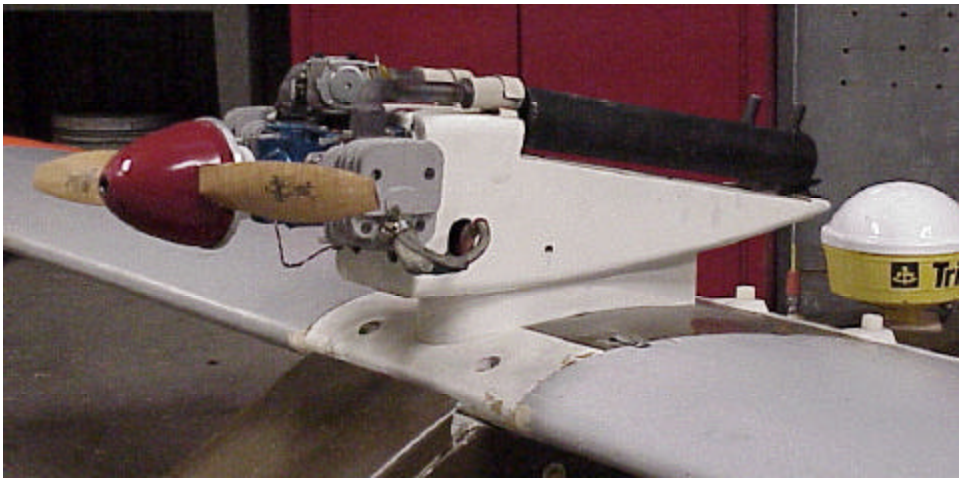

Figure 2.3. FROG Engine Configuration.

PARAMETER	MEASUREMENT
Length	8.125 ft
Height	1.75 ft
Weight	67.7 lbs
Power Plant	9.3 Hp / 2 Cycle
Wing Airfoil	NACA 2415
Horizontal Stabilizer Airfoil	NACA 0006 (Approx.)
Wing Span (b)	126.5 in
Tail Span (b_w)	39.75 in
Vertical Tail Span (b_v)	15.0 in
AR_w	6.32

Table 2.1. FROG UAV Physical Characteristics.

B. FROG RAPID FLIGHT TEST PROTOTYPING SYSTEM

The FROG Rapid Flight Test Prototyping System (RFTPS) is a broad description of the hardware and software architecture used control the FROG in flight. The RFTPS may be conveniently divided into a command channel (uplink) and feedback channel (downlink). The ground segment of the command channel includes a safety pilot with manual radio controller, an AC-104 computer running the flight control software and a pulse code modulation (PCM) transmitter. The airborne segment includes a PCM receiver and the servo actuators. The feedback channel includes the differential global positioning system (DGPS) receiver, inertial sensors (NPS inertial measurement unit or Crossbow attitude heading reference system), control surface position transducers, and wireless spread spectrum modems. Flight control commands are generated on the ground by either the safety pilot or by the AC-104 computer. Command signals from the AC-104 computer are converted to a PCM signal by a Futaba® radio controlled airplane transmitter, which broadcasts them to the airplane. The airplane's Futaba® receiver then decodes the PCM signal and generates PWM commands for each of the control servo channels. In the feedback channel, sensor outputs are digitized and transmitted via spread spectrum modem to the ground station for processing. Two

different inertial measurement units are currently in use with the FROG. When configured with the NPS IMU, the DGPS and IMU sensor data are merged into a single time phased serial output stream. This is accomplished by an embedded microcomputer within the IMU. The CrossBow® attitude heading reference system's (AHRS) higher bandwidth necessitated addition of a second downlink modem to the airplane. The RFTPS architecture is depicted in Figure 2.4.

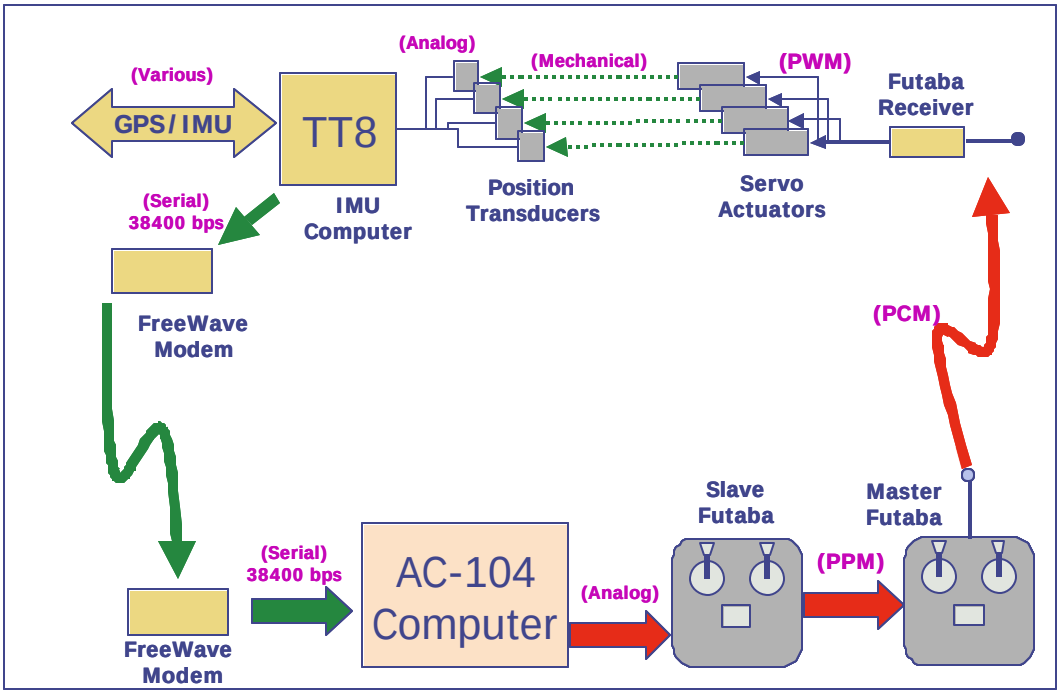

Figure 2.4. FROG Rapid Flight Test Prototyping System.

1. MATRIX$_X$ Rapid Prototyping System

The MATRIX$_X$ software suite provides and integrated environment for control system design, software engineering, data acquisition and testing. The software suite consists of Xmath, SystemBuild, RealSim, AutoCode, and the pSOSystem real time operating system. The Xmath program provides the system analysis and visualization environment. Xmath includes over 700 predefined functions and commands and includes a compact scripting language for simplified command and function programming. Basic Xmath commands support operations such as creating, plotting, saving and loading data. Additional add-on modules provide sophisticated control system design and analysis functions. SystemBuild visual modeling and simulation software provides a graphical control system design environment. Continuous time,

discrete time and hybrid systems are easily constructed by selecting elements from predefined palettes. SystemBuild includes built in simulation tools that allow the user to interactively verify, test and modify system models. Data generated during SystemBuild simulations may be captured and further analyzed using the powerful built-in (or user defined) functions in Xmath. The functions and implementation of Xmath/SystemBuild is analogous to MATLAB/SimuLink. The AutoCode module is an automatic code generator for SystemBuild models. The AutoCode software processes the SystemBuild model files and creates either ANSI C or Ada code, which can then be compiled to create a stand-alone real-time executable program. AutoCode builds the program scheduler by means of a template file for the target real time operating system (RTOS). The scheduler performs overall direction and control of inserting inputs, scheduling tasks, posting outputs and dispatching the tasks that perform the work of the real time system. The application scheduler operates on the principle of rate-monotonic scheduling, deriving priorities for the tasks from the repetition rate for periodic subsystems. Higher priorities are assigned to faster subsystems while slower ones are assigned lower priority. Understanding of the scheduler mechanization is critical to achieving predictable real-time performance. Once compiled, the stand-alone executable code is suitable for the test-bed environment or for use in an embedded real-time system. The RealSim controller allows the user to perform real-time simulations of feedback control systems designed in SystemBuild. RealSim compiles and links the application code and includes provisions for connecting real hardware to the controller for rapid prototyping and hardware-in-the-loop (HITL) testing. A data acquisition module permits the user to record any selected input/output parameters for further analysis in Xmath. Run-time graphical user interfaces can be built that allow the user to observe and interact with the simulation while it is running on a real-time computer. $MATRIX_X$ also includes custom drivers for the PC-104 based AC-104B computer to permit easy download and execution of the real-time system. The pSOSystem RTOS provides an execution environment on the computer that runs the real-time code. AutoCode includes a template that optimizes the C code for the pSOSystem RTOS.

2. AC-104 Controller Configuration

The complete RealSim system includes a host PC (Windows 95/98/NT/2000 and the $MATRIX_X$ software suite) and a second PC with an RTOS, to act as the controller. The AC-104 is a compact microcomputer manufactured by Integrated Systems Incorporated (now WindRiver Systems, Inc.), Figure 2.5. The AC-104 facilitates real-time control and prototyping of hardware systems by integrating a complete microcomputer with PC-104 based hardware I/O modules.

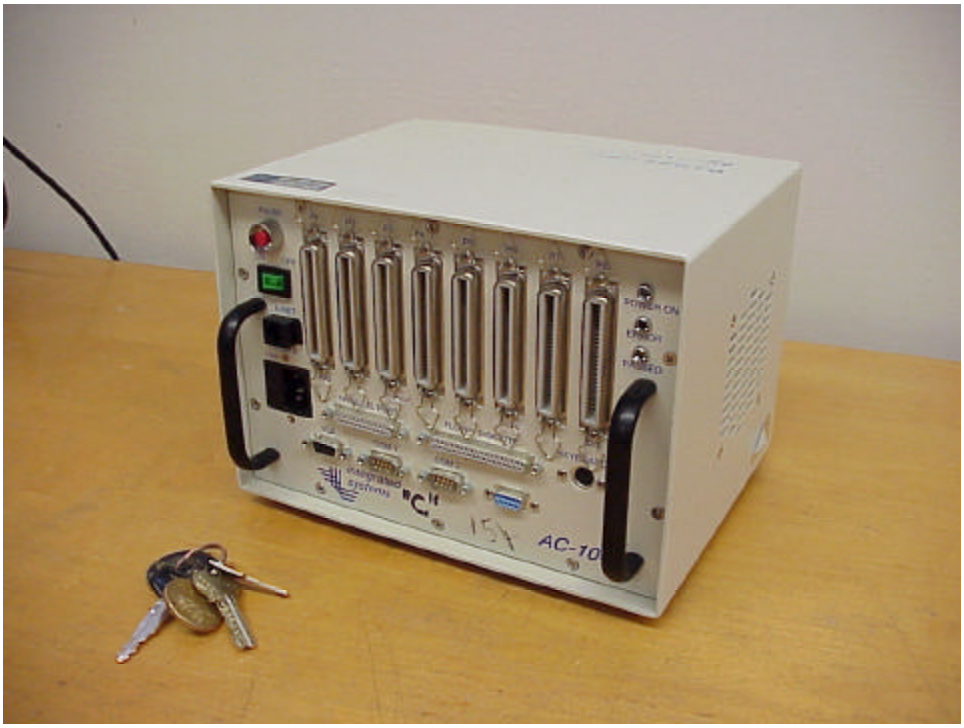

Figure 2.5. AC-104 Real-Time Controller.

The AC-104 is based on an Advantech PCM-5862 motherboard configured with a Pentium MMX processor operating at 233 MHz. The NPS AC-104's are configured with 16 MB of EDO RAM, and a 4 MB flash disk for non-volatile storage. Basic I/O is provided by a PCI–SVGA display controller, two serial ports (RS-232/422/485), an enhanced parallel port, keyboard controller and a PCI based 10Base-T Ethernet connection. The Host computer communicates with the AC-104 via the Ethernet connection and may do so by direct cable connection or across a distributed network. Enhanced I/O functions for hardware control and data acquisition are provided by add-on cards, which interface with a PC-104 ISA expansion bus. The AC-104 comes

configured with an Analogic AIM16 PC-104 16-bit analog-to-digital (A/D) converter, a Diamond Systems Ruby-MM 12 bit digital-to-analog (D/A) converter and an SBS GreenSprings Modular I/O Industry Pack IP-68322 data acquisition and control module. The IP-68322 module integrates a Motorola MC68322FC micro-controller with a Xilinx 3030 field programmable gate array (FPGA) onto a standardized form factor to provide advanced data acquisition and hardware control capabilities. The IP-68322 is a daughter board on a two slot Flex/104A carrier board. NPS has added an SBS GreenSpring IP-Serial board to the second slot on the Flex/104A board. The IP-serial board provides two RS-232-C/422 serial channels and provides programmable baud rates up to 2 Mbit/sec. The AC-104 has eight 50-pin Centronics connectors for PC-104 based I/O. All connections are located on the front face of the AC-104, Figure 2.6. Standard PC connectors are also provided for all non-PC-104 I/O.

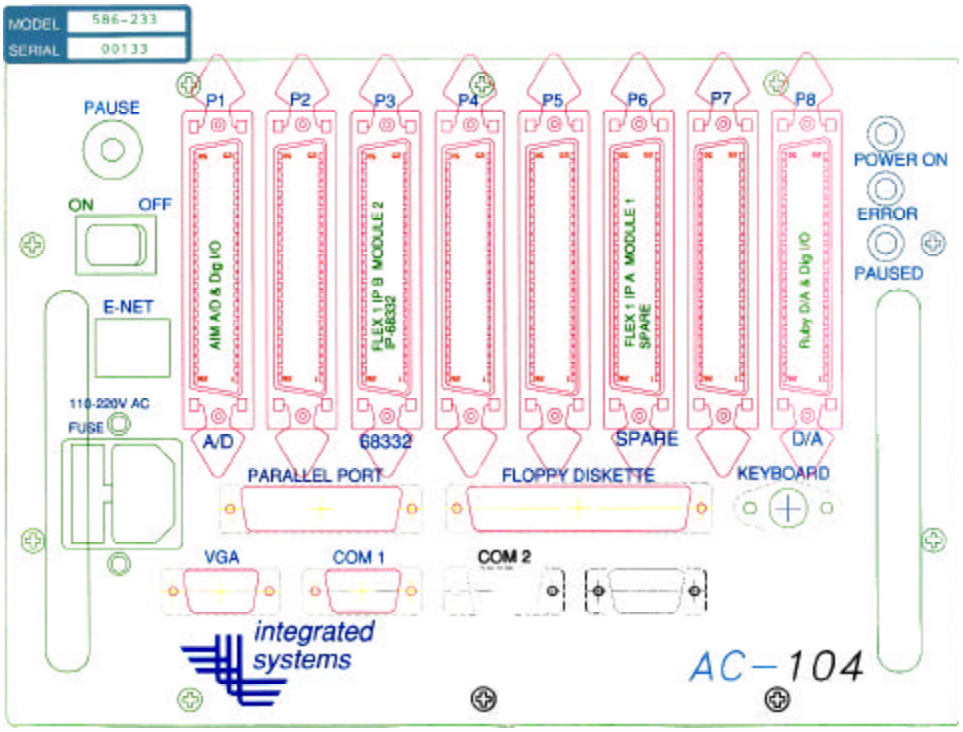

Figure 2.6. AC-104 Interface Layout.

3. Command Uplink

The FROG is controlled by command signals transmitted from a ground based Futaba FP-9ZAP digital proportional radio control set, Figure 2.7. The NPS FROG command channel takes advantage of a unique feature designed into the FUTABA

transmitters. Two Futaba transmitters may be linked together, by means of a cable, to form a MASTER / SLAVE system, as depicted in Figure 2.8.

Figure 2.7. Futaba FP-9ZAP Digital Proportional Radio Control.

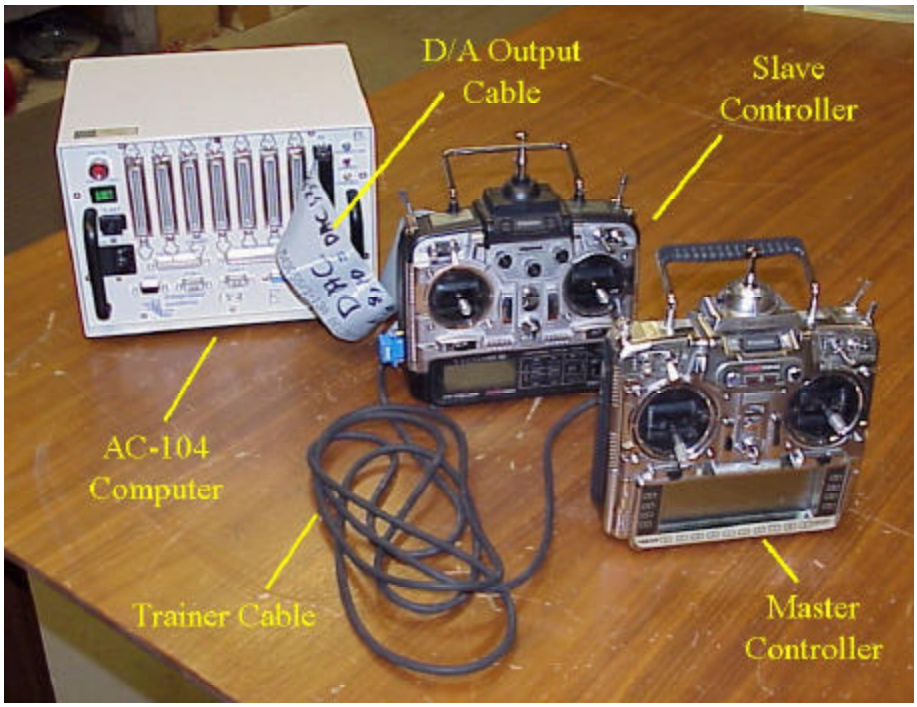

Figure 2.8. FROG Master/Slave Transmitter Arrangement.

This configuration was originally designed to facilitate the training of novice radio controlled airplane pilots. The instructor pilot can operate the Master controller while the student operates the Slave. When the Master controller is active the instructor pilot's command inputs are converted to pulse code modulated (PCM) signals that are transmitted to the airplanes receiver. The pilot's control joysticks are connected to potentiometers, which provide a voltage that is proportional to the joystick position. This voltage value is then used to encode the PCM stream. The Master controller transfers control to the Slave by actuation of a single electric switch. When the Slave controller is active the Slave's commands are converted to pulse period modulation (PPM) signals, which are then relayed to the Master via the link cable. The Master controller decodes the PPM signals from the Slave and converts them to PCM for output on the Master's transmitter. Significant latency in the AC-104's commands, due to this mechanization, was discovered during this research. This time delay poses significant a challenge for control system design. The Slave controller is a highly modified 8-channel Futaba FP-8UAP digital proportional radio. In the Slave, the joystick potentiometers have been disconnected and externally generated voltages are passed directly to the controller's A/D converter via a DB-9 connector. The AC-104 based controller converts airplane control commands into scaled voltages via an integral D/A converter. These signals are then passed to the Slave controller by a locally manufactured data cable connected to the DB-9. The Futaba transmitters have a maximum range of approximately 1.5 miles. This short range severely limits the volume in which this airplane may operate. Figure 2.4 depicts the RFTPS command channel architecture.

Onboard the FROG, a Futaba FP-R309DPS receiver decodes the PCM command signal and generates pulse width modulated (PWM) commands on each of eight separate channels. Each channel may be connected to one of the control servos, as depicted in Figure 2.9. The FROG is configured with a variety of servos; each with different performance specifications. All servos share a common specified neutral pulse width (positions the servo at the mid-range location) of 1.52 ms. Due to installation requirements, the neutral control surface position does not correspond to the neutral servo position.

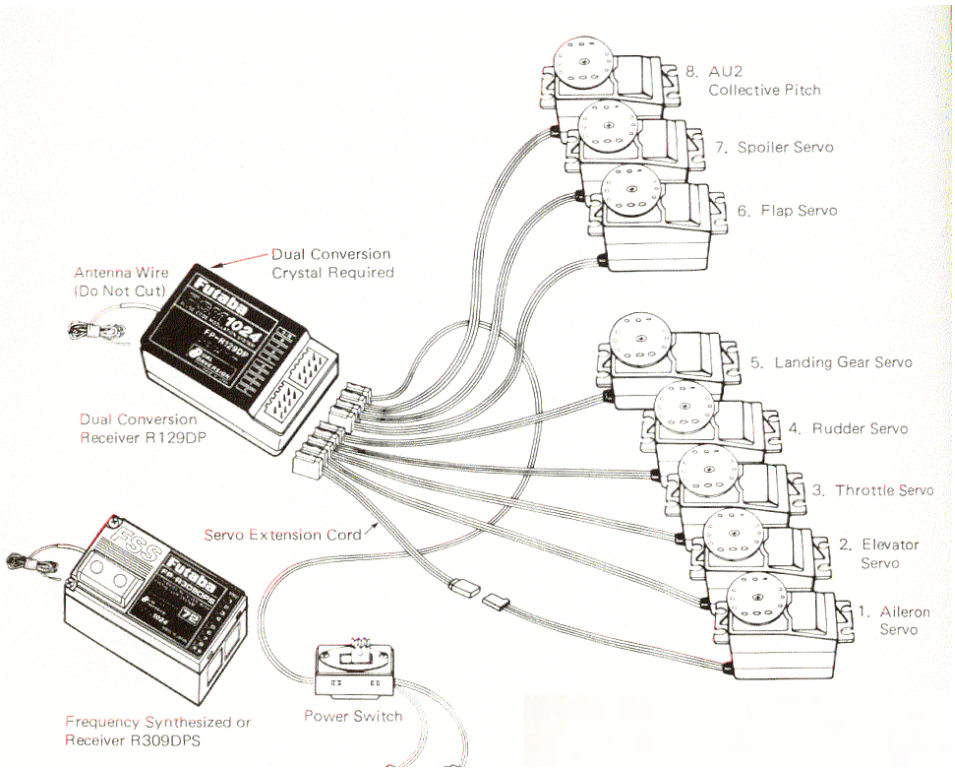

Figure 2.9. Futaba® Receiver to Servo Connections from Ref. [1].

4. Sensor Downlink

Sensors onboard the FROG provided feedback to the digital flight control system. At the present time there are two different GPS/INS sensor suites under development for use with the FROG. The first is based on an indigenously designed and manufactured inertial measurement unit (the NPS IMU) and a DGPS receiver manufactured by Trimble, Inc. The second is comprised of the same DGPS receiver and a commercial attitude heading reference system (AHRS) manufactured by Crossbow Technology, Inc. The data interface differs significantly for these two configurations, which necessitates unique avionics architectures for each. Both sensor suites include a Trimble *Ag*GPS 132 DGPS receiver. The *Ag*GPS is a 12 channel L-band differential correction receiver that provides sub-meter accuracy. The GPS receiver is configured with two programmable RS-232 serial ports and provides position updates at a maximum frequency of 10 Hz.

The NPS IMU configuration consists of the A*g*GPS 132 receiver and an indigenously designed and manufactured IMU. The IMU contains both analog and

digital sensors. In order to convert the analog signals to digital data and format a combined digital output stream a microcomputer was developed. The computer is based on two TattleTale 8 data loggers, manufactured by Onset Computer Corporation. The Tattletales are configured in a Master/Slave arrangement. Each TattleTale includes a Motorola 68322 micro-controller, an eight channel 12-bit A/D converter and two RS-232 serial ports. The digital and analog sensor data and GPS serial output are processed in the TattleTale computers and formatted into custom serial messages for down link to the ground station. The Master Tattletale outputs IMU sensor data at 40 Hz and interleaves GPS messages at 10 Hz. The serial transmission rate is 38,600 bps. A more detailed discussion of the IMU computer's design and operation may be found in paragraph II.C.2. The Master Tattletale computer is connected to a FreeWave wireless spread spectrum data transceiver (modem) manufactured by FreeWave Technologies, Inc. The FreeWave modem has a power output of 1/3 Watt and operates at in a frequency range of 902 – 928 MHz. It is capable of communicating at a line of sight range of up to 20 miles and supports data transmission at baud rates from 1200 bps – 115.2 Kbps. A matching FreeWave modem is connected to the PC-104 based IP-Serial card on the AC-104 ground station and provides a continuous data stream to the FROG controller. The NPS IMU downlink architecture is depicted in Figure 2.10.

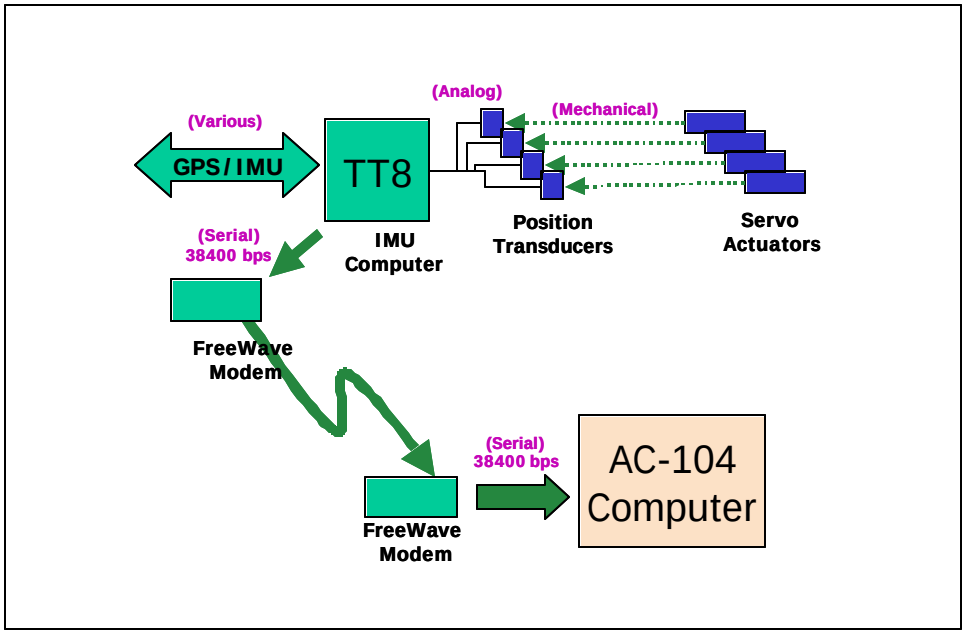

Figure 2.10. NPS IMU Downlink Architecture.

The Crossbow AHRS is currently configured for continuous digital output of all sensor data at an average rate of 65 Hz. The TattleTale downlink bandwidth limits updates to approximately 50 Hz so an alternate downlink configuration was developed in order to take full advantage of the AHRS faster update rate. The A*g*GPS receiver (10 Hz) and the Crossbow AHRS (60 – 70 Hz) are each connected to their own dedicated FreeWave modem. These two modems communicate independently with two identical modems which are connected to Channels A and B on the AC-104 IP-Serial card. The digital flight controller reads each serial stream and updates the control system accordingly. The Crossbow AHRS downlink configuration is depicted in Figure 2.11.

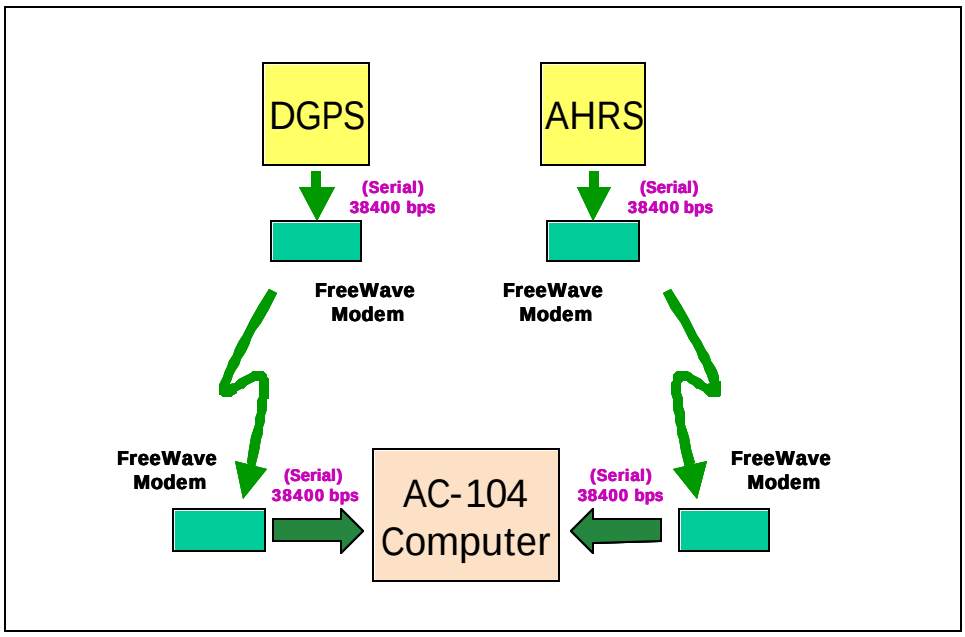

Figure 2.11. Crossbow AHRS Downlink Architecture.

5. Electromagnetic Interference

Prior to one flight test, spurious signals were found to be interfering with the Futaba command signals. The FROG had been configured with both the NPS IMU and Crossbow AHRS. During the ground check the safety pilot found that the control servos responded erratically to each input. The electromagnetic interference (EMI) was initially attributed to the presence of two FreeWave modems, in close proximity to the airplane's Futaba receiver, but extensive trouble shooting at the UAV lab was unable to duplicate the effect. While the FreeWave modems may contribute to an adverse

electromagnetic environment the NPS IMU itself may also be a significant contributor. The IMU has a DC-to-DC switching converter mounted externally to provide regulated power to the various avionics sub-systems. The EMI effects of this converter should be investigated. Additionally, the frequency spectrum that the Futaba controllers use is publicly available. As such, accidental (or intentional) signal jamming may be encountered. Transition to serial uplink transmission would significantly improve the signal to noise tolerance of the system and reduce the likelihood of accidental interference.

C. NPS INERTIAL MEASUREMENT UNIT

The design and manufacture of an integrated inertial measurement unit (IMU) is a significant undertaking. The NPS IMU was developed to provide a low cost, high quality inertial measurement unit for use in digital control systems for the FROG UAV. The decision to locally design and manufacture an IMU was necessitated by the inadequate performance of an IMU-600D IMU manufactured by Watson Industries. The NPS IMU was designed not only as a sensor for FROG flight control projects but also as a teaching tool that would allow investigation of such subjects a sensor modeling, time-correlated random constant errors (such as bias and misalignment), complementary and Kalman filtering, digital avionics and real-time operating systems for embedded control. The adequacy of the NPS IMU for in flight inertial measurement has not been evaluated yet. The hardware and embedded controller's software have just recently reached the state where they can support sensor calibration. IMU calibration, to determine the time-correlated errors, can be accomplished at NPS using a 2-axis turning table and is expected to proceed in the near future. While much of the NPS IMU's potential has yet to be realized, it has provided valuable experience in the software design, data bandwidth management, micro-controller design and the practical challenges involved in any system development effort where specific design criteria are developed in parallel to the hardware.

1. Sensor Description

The development of the NPS IMU must be characterized as evolutionary. As such the sensor suite and associated electronics have undergone several significant revisions. In it's current form the NPS IMU is configured with solid-state rate

gyroscopes, accelerometers, magneto-resistive flux sensors and a spinning mass vertical gyroscope. Three Gyrochip™ AQRS-00064-104 angular rate sensors, manufactured by the Systron Donner Inertial Division of BEI Technologies, Inc, are used to measure angular rates. The Gyrochip™ rate sensors utilize a micromachined double-ended quartz tuning fork fabricated from mono-crystalline piezoelectric quartz. Specifically designed for "demanding automotive and commercial" applications, the gyros have a specified range of ± 64°/sec. Applying the Coriolis effect, a rotational motion about the sensors input axis produces a DC voltage (analog output) proportional to the rate of rotation. As each gyro is housed in it's own individual package it is expected that misalignment and relative orthogonality will be significant contributors to the time-correlated random error. The AQRS Gyrochip™ is depicted in Figure 2.12.

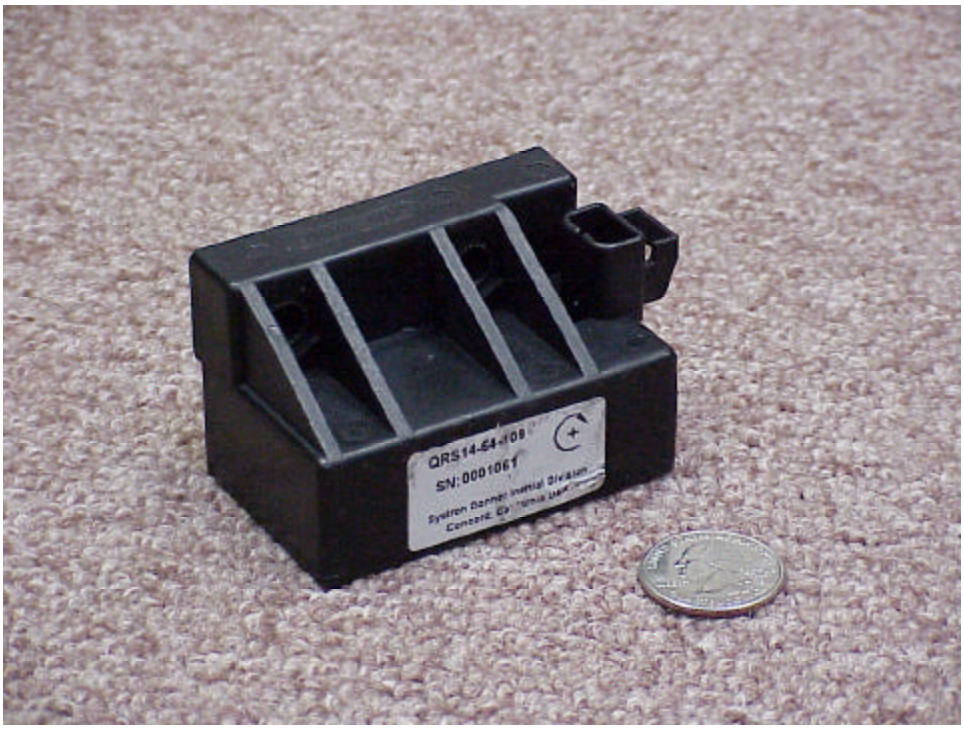

Figure 2.12. BEI Gyrochip™ Rate Gyro

The 3DM™ 3-axis orientation, manufactured by MicroStrain, Inc., provides linear acceleration and magnetic flux measurements. The 3DM™ is designed to provide tilt sensing via orthogonal accelerometers and magnetic compass functions via orthogonal flux gate magnetometers. The 3DM™ is configured with two ± 2 g

ADXL202E dual axis IC based accelerometers, manufactured by Analog Devices, Inc. The accelerometers contain polysilicon surface micromachined sensors and signal conditioning circuitry to implement an open loop acceleration measurement architecture. The output of each sensor element is lowpass filtered and then converted to a duty cycle (pulse width) modulated signal for output to the 3DM's own microprocessor. Magnetic field sensing is provided by orthogonally mounted IC based linear magnetic field sensors, manufactured by Honeywell. The HMC1021/1022 magneto-resistive sensors are made of a nickel-iron (Permalloy) thin film deposited on a silicon substrate. The Honeywell magneto-resistive sensors are simple resistive bridge devices that provide an analog voltage output corresponding to any ambient or applied magnetic field. A 12-bit A/D converter within the 3DM™ digitizes the analog output of the magnetic sensor. The 3DM™ uses the accelerometers and magnetometers to calculate pitch, roll and yaw angles relative to the earths magnetic and gravitational fields. The 3DM™ can also be programmed to output raw accelerometer and normalized magnetic field strength values converted to engineering units. All data is output at 9600 baud via an RS-232 serial port. The 3DM supports a maximum output rate of approximately 30 Hz. During early development of the NPS IMU the 3DM™ suffered spurious mode switching between continuous and polled modes of operation. At that time the erratic behavior was attributed to EMI within the IMU case. To alleviate this problem MicroStrain provided a special version of the 3DM™ embedded operating software on EEPROM. This software supports only the polled/raw sensor output mode of operation. The original 3DM™ EEPROM has been retained and the unit could be converted back to the angle measuring configuration once EMI concerns have been addressed. The 3DM™ orientation sensor is depicted in Figure 2.13.

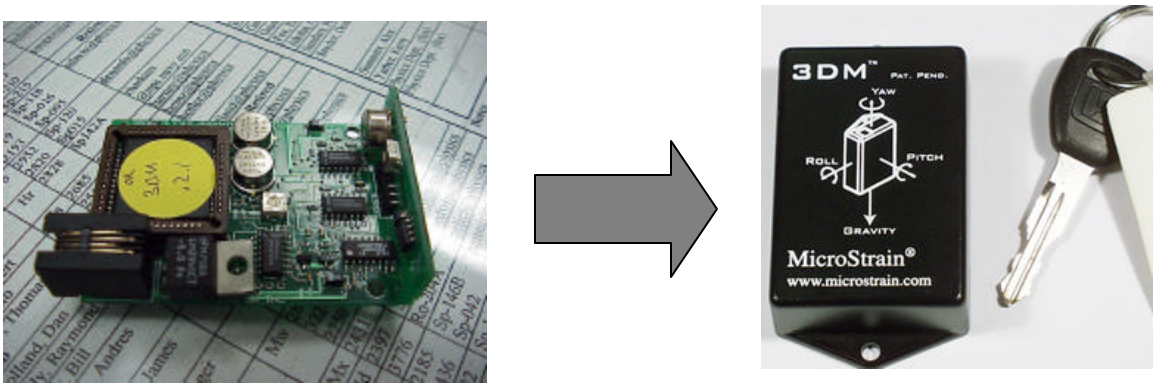

Figure 2.13. 3DM™ 3-Axis Orientation Sensor.

The NPS IMU contains a second attitude source in the form a VG-34-0201-2 vertical gyroscope manufactured by Humphrey, Inc. The VG-34 vertical gyro is a classic gimbaled spinning mass gyroscope. The vertical gyro contains two electrolytic level sensors, which provide a "gravity" reference for pitch and roll position. The VG-34 has a specified accuracy of ±0.1° for both pitch and roll. Conductive plastic potentiometers are used to provide analog pitch and roll output signals. The Humphrey vertical gyro's requirement for 28 Vdc and relatively high power consumption was the significant driver in the design of the IMU's power supply. The VG-34 vertical gyroscope is depicted in Figure 2.14.

Figure 2.14. Humphrey VG-34-0201-2 Vertical Gyroscope.

Table 2.2 lists the operating limits of the NPS IMU's sensors and the output data formats.

Sensor	Parameter	Limits	Output
Gyrochip™ Rate Gyros	Angular rate	± 64 ° / sec	Analog – voltage
3DM™ Orientation Sensor	Linear Acceleration	± 2 g (6 KHz bandwidth)	Serial – RS-232 at 9600 buad
	Magnetic Field	± 2 guass (Normalized)	Serial: RS-232 at 9600 baud
Humphrey VG-34 Vertical Gyro	Pitch / Roll	± 60° / ± 90°	Analog - voltage

Table 2.2. NPS IMU Sensor Performance Limits.

2. Embedded Micro Controller Development

The FROG's airborne sensor suite includes the NPS IMU (or Crossbow AHRS), the Trimble *Ag*GPS 132 differential GPS receiver and a variety of analog transducers (position, pressure and temperature). The serial downlink architecture mandated that all sensor data be digitized prior to transmission to the ground station for processing. It was also desired to minimize the potential for EMI by employing a single wireless modem to transmit the onboard data. Within the NPS IMU both digital and analog sensor outputs are present. Additionally, the required GPS data are embedded in two separate serial messages. In order to provide a single formatted IMU output message that could meet high update rates, with minimal latency, it was necessary to develop an embedded microcomputer. At a minimum the microcomputer needed the ability to convert analog signals to digital (A/D), store and process the sampled data, and send and receive serial data. The microcomputer was required to interface with as many as sixteen analog sensors, receive serial data from two separate sources (operating at different baud rates and update frequencies) and transmit serial data on a third channel. The TattleTale Model 8 data logger, manufactured by ONSET Computer Corporation, was selected for this purpose. The TattleTale 8 is a low cost miniature micro-controller that provides an 8-channel A/D converter, fourteen digital I/O lines, two RS-232 ports and is powered by a Motorola 68322 32-bit microprocessor and a PIC 16C64 coprocessor, Figure 2.15.

The Tattletale 8 is also configured with 256Kb of Flash EEPROM for storing the controller program and 1 MB of RAM.

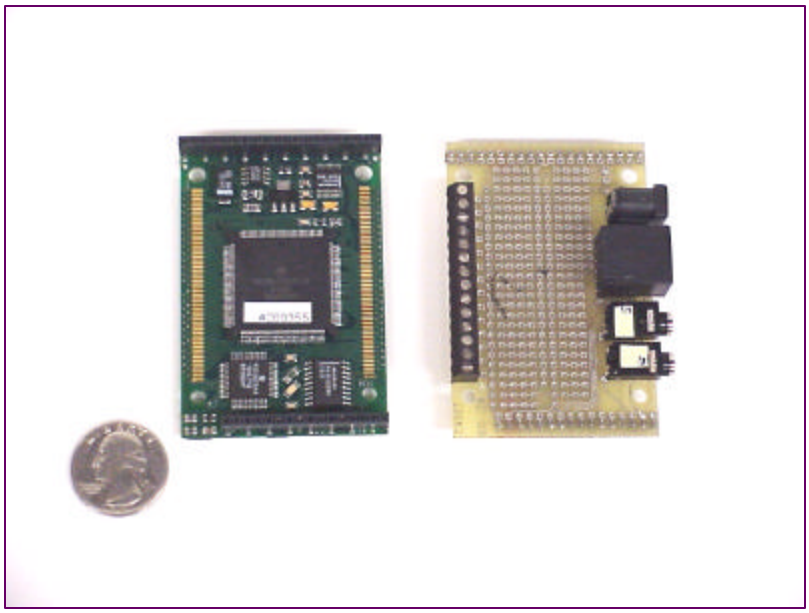

Figure 2.15. TattleTale Model 8 Micro-Controller.

As a single TattleTale could not satisfy the demands for analog to digital conversion and serial communication, two TattleTales are networked to form the embedded IMU computer. Initially, one processor was programmed to read and process the IMU sensor data (hereafter referred to as the "3DM TattleTale") and the other was programmed to read and process the GPS serial output messages (hereafter referred to as the GPS TattleTale). The GPS and 3DM TattleTales time-shared a single wireless modem by means of an external CMOS multiplex switch that allowed the 3DM TattleTale to transmit at a different rate than the GPS TattleTale. In this scheme the output signals were triggered to the GPS output message, which was believed to be a reliable 10 Hz. Both TattleTale's were programmed in BASIC and the resulting code output GPS data at 10 Hz and IMU data at approximately 20 Hz. As the system requirements were better understood the required IMU sensor update rate was increased to a minimum of 40 Hz. In order to meet the new timing requirements LT Matt Commerford crafted an entirely new real-time operating system for each TattleTale using the Aztec C language (a variant of ANSI C). Additionally, LT Commerford developed a Master/Slave architecture which greatly increased the output rate while also

significantly improving the systems overall determinism. Though the previous nomenclature has been retained the current Master/Slave micro-controller configuration is more appropriately called the "FROG onboard computer". The FROG computer functional architecture is depicted in Figure 2.16. The following discussion draws heavily from the unpublished notes of LT Commerford, reference 2.

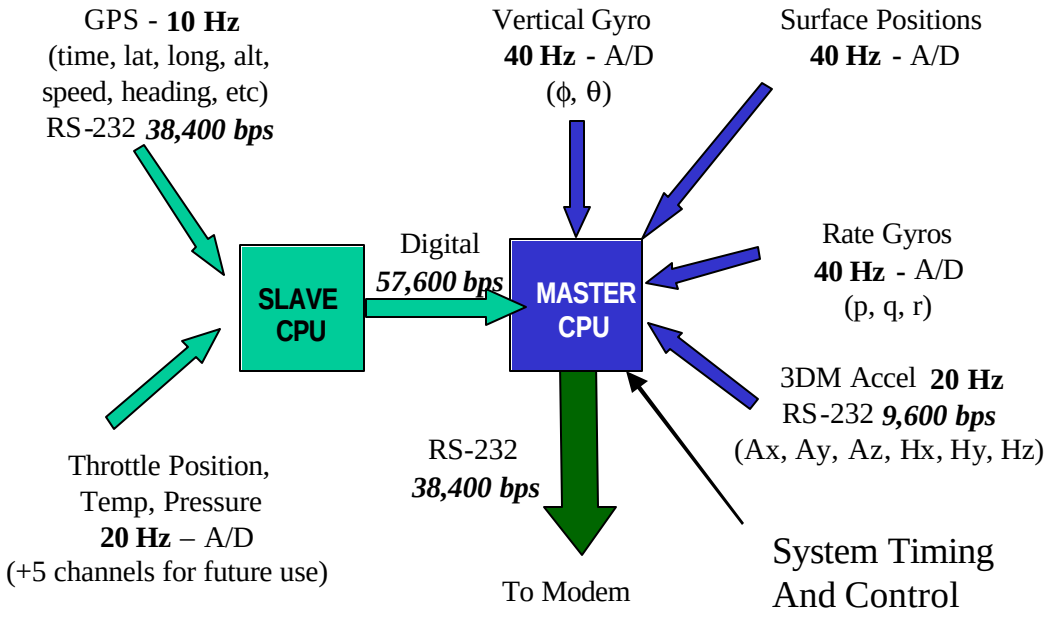

Figure 2.16. FROG Onboard Computer Functional Architecture after Ref. [2].

The 3DM TattleTale is the Master processor in the Master/Slave configuration. It controls the overall system timing and acts as the scheduler for all serial output to the ground station. The Master processor samples the IMU's analog sensors, polls the 3DM sensor and receives digital data from the GPS TattleTale via a digital I/O line configured for serial communication. The GPS TattleTale receives the DGPS receiver's serial output and parses the data for the minimum required data set. The customized GPS message is then transferred to the Master processor where it is queued for transmission. The Slave processor's A/D channels are also sampled and passed to the Master for output with the IMU data. Each sensor has its own inherent limitations that restrict the maximum allowable data rate. Samples from each sensor are taken at various data rates and merged together in order to achieve a highest possible downlink data rate while maintaining an easily decoded and error free data stream. In order to maximize the

downlink bandwidth all sensor data is converted to binary prior to transmission (vice ASCII). The use of binary coded data significantly reduces the number of bytes transmitted. The combined sensor output provides measurements of the complete state vector, with the exception of heading (ψ). The measured parameters, associated sensors and computer update rates are presented in Table 2.3.

Parameter	Data Rate	Source	Notes
Angle rates (p,q,r)	40 Hz	AQRS 104	4.096 Volt A to D sample
Pitch / Roll Angle (ϕ,θ)	40 Hz	VG-34	4.096 Volt A to D sample
Control Surface Position (elevator, aileron, rudder)	40 Hz	Series 150 String Pots	4.096 Volt A to D sample
Throttle Position	10 Hz	Series 150 String Pot	4.096 Volt A to D sample
Acceleration (a_x, a_y, a_z)	20 Hz	3DM	RS-232 9,600 baud
Magnetic Field (H_x, H_y, H_z)	20 Hz	3DM	RS-232 9,600 baud
GPS Time (t)	10 Hz	Ag132 GPS	RS-232 38,400 baud
GPS Latitude (Lat)	10 Hz	Ag132 GPS	RS-232 38,400 baud
GPS Longitude (Long)	10 Hz	Ag132 GPS	RS-232 38,400 baud
GPS Altitude (Alt)	10 Hz	Ag132 GPS	RS-232 38,400 baud
GPS Ground Speed (Knots)	10 Hz	Ag132 GPS	RS-232 38,400 baud
GPS Ground Track (Degrees True)	10 Hz	Ag132 GPS	RS-232 38,400 baud
Magnetic Variation (Deg)	10 Hz	Ag132 GPS	RS-232 38,400 baud
Outside Air Temperature	20 Hz	TBD	4.096 Volt A to D sample
Differential Pressure (for airspeed)	20 Hz	TBD	4.096 Volt A to D sample
Future Use	20 Hz	TBD	5 Spare A/D channels (α, β, V_{bat}, etc.)

Table 2.3. NPS IMU Downlink Data List.

a. 3DM Tattletale Program

The 3DM TattleTale (Master) was required to sample and process all inertial sensor and control surface position data and schedule and perform the transmission of all airborne data. This task was complicated by the differing update rates between the 3DM sensor and the GPS receiver. It was desired that all sensor data

be updated and transmitted at the fastest reliable rates. The sensors supported three different update rates: 10 Hz, 30 Hz and continuous. An analysis of the data bandwidth requirements indicated that a maximum minor frame size of 20 ms (50 Hz) was possible. In order to ensure successful operation a minor frame length of 25 ms was chosen (40 Hz) for the scheduler. The 3DM's special EEPROM only supports polled mode operation. This requires that the Tattletale send a "request for data" command (90h or 144 decimal) to which the 3DM will respond with the 13-byte message presented in Table 2.4.

Data	Description
Diagnostic Byte	0x41h if Valid; 0x6Xh if error ('X' is an error code)
H_{X-m}	X Axis Magnetometer Data MSB
H_{X-l}	X Axis Magnetometer Data LSB
H_{Y-m}	Y Axis Magnetometer Data MSB
H_{Y-l}	Y Axis Magnetometer Data LSB
H_{Z-m}	Z Axis Magnetometer Data MSB
H_{Z-l}	Z Axis Magnetometer Data LSB
A_{X-m}	X Axis Accelerometer Data MSB
A_{X-l}	X Axis Accelerometer Data LSB
A_{Y-m}	Y Axis Accelerometer Data MSB
A_{Y-l}	Y Axis Accelerometer Data LSB
A_{Z-m}	Z Axis Accelerometer Data MSB
A_{Z-l}	Z Axis Accelerometer Data LSB

Table 2.4. 3DM Magnetometer & Accelerometer Data Format.

The 3DM data are transmitted MSB first and LSB second for each measured value. The bytes are then decoded as Value = 256 x MSB + LSB.

The 3DM provides a single output rate of 9600 baud that should support update rates in excess of 70Hz however timing tests revealed that a maximum reliable update rate was only about 30 Hz. In order to provide consistent and predictable timing performance it was decided to poll the 3DM at 20Hz (once every other minor frame). Once received, the 3DM data is buffered in the Tattletale's RAM and is transmitted with

each periodic IMU message. The remaining analog sensors in the IMU data package are sampled at 40 Hz. To simplify the decoding of the IMU data, a single standard IMU data message was developed. This message is transmitted at 40 Hz; with the 3DM data updated only every other frame. The "userser.c" serial driver written for the RFTPS decodes this downlink data message. As a result of this implementation, the 3DM output occurs in repeated pairs. In order to perform frequency domain analysis it is first necessary to strip off every other data value and use the sampling rate of 20 Hz for the analysis. The 3DM™ TattleTale program is included in Appendix A.

b. GPS TattleTale Program

The Trimble A*g*132 GPS receiver computes position at a maximum frequency of 10 Hz. The GPS position data are then transmitted at 38400 baud to the GPS TattleTale processor. All output conforms to the National Marine Electronics Association (NMEA) GPS data protocol. Unfortunately, the parameters needed for navigation and control are not contained in a single standard NEMA message. The minimal GPS data set can be generated from two separate NEMA messages: $GPGGA and $GPRMC. The combined $GPGGA and $GPRMC sentences consist of approximately 161 ASCII text characters (varies slightly depending upon data content). At an update rate of 10 Hz and baud rate of 38400 bps these two complete messages would consume approximately thirty percent of the available downlink bandwidth. Fortunately, it is possible to glean a minimal data set from these messages and generate a single binary coded GPS data message that only requires 31 bytes. The GPS TattleTale receives the GPS receiver's output messages via one of its RS-232 serial ports. The messages are then parsed to form a compact GPS message. The compact GPS data message is then transmitted at 57600 baud to the Master processor, on the 3DM Tattletale, via one of the 68322's digital I/O lines. Table 2.5 presents the data contained in the standard NEMA $GPGGA and $GPRMC output messages and the NPS IMU GPS downlink message.

Parameter	$GPGGA	$GPRMC	NPS IMU
UTC of Position Fix	X	X	X
Latitude	X	X	X
Latitude Direction (N or S)	X	X	
Longitude	X	X	X
Longitude Direction (E or W)	X	X	
GPS Quality Indicator	X		X
Number of Satellites in use	X		
Horizontal Dilution of Precision (HDOP)	X		
Antenna Altitude (MSL)	X		X
Geoidal Separation	X		
Time since last Differential mode update	X		
Differential Mode reference Station ID#	X		
Data Status		X	
Speed over ground (kts)		X	X
Track made good in degrees True		X	X
UTC Date		X	
Magnetic Variation		X	X
Magnetic Variation Direction		X	X
Checksum	X	X	

Table 2.5. Comparison of GPS Message Data Content.

The specified GPS update rate of 10 Hz corresponds to a period of 100 ms for each GPS message. Lt Commerford's extensive analysis of the GPS output signal confirmed that the ASCII text data is reliably received from the GPS at an average rate of 10 times per second, without dropouts our lost samples. The exact time of the arrival of the GPS data however exhibited significant variability. While the average update rate was 10 Hz each individual update arrived early or late; with some observed as much as 100 ms time late (i.e. 200 ms between successive updates). This uncertainty in sample time presents problems when trying to model the GPS for control system design and analysis and may require greater gain and phase margins to account

for unmodeled (or inaccurately modeled) sensor behavior. The GPS parsing routine is designed to wait until a full message has been received. Therefore, the timing of the computers major and minor output frames was coordinated by the Master processor's real-time clock signal (vice the GPS message signal used in the previous implementation). The Slave processor's operating code is included in Appendix A.

Downlink data is transmitted in a binary format as a series of 8-bit bytes. This coding scheme reduces the throughput requirement on the downlink channel but requires the ground station's serial driver to decode it before it can be input into the flight controller. Values that range between 0 and 255 can be represented by a single byte, with no conversion necessary. Values that range between 0 and 65535 must be represented by a two 8-bit bytes. Our convention is to send the most significant byte (MSB) first, followed by the least significant byte (LSB). To decode a two-byte unsigned integer the following formula is applied:

$$Value = 256 \bullet MSB + LSB \tag{2.1}$$

For signed integers a slightly different conversion must be used; since the most significant bit of the MSB is usually a sign bit. The procedure used to decode 16-bit 2's complement data is

$$\begin{aligned} &Value = 256 \bullet MSB + LSB \\ &if \ \text{Value} > 2^{15} \\ &Value = 2^{16} - Value \\ &endif \end{aligned} \tag{2.2}$$

Since the 3DM TattleTale downlinks three different data messages a unique two-byte header was incorporated to distinguish each one. Table 2.5 presents a summary of the TattleTale downlink messages characteristics.

Message Content	Header (Hex / Decimal)	Number of Data Bytes (less Header)	Transmit Rate
IMU Sensor Data	FF FF / 255 255	28	40 Hz
GPS Data	EE EE / 238 238	29	10 Hz
GPS TattleTale A/D Data	DD DD / 221 221	16	20 Hz

Table 2.5. TattleTale Downlink Message Characteristics.

D. CROSSBOW ATTITUDE HEADING REFERENCE SET

The AHRS400CA-100 is a low cost, compact solid-state AHRS manufactured by Crossbow Technology, Inc. The AHRS measures linear acceleration, angular velocity, and magnetic flux for three orthogonal axes and computes stabilized values of pitch, roll and heading by using proprietary Kalman filter algorithms. Output data is provided in both digital and analog formats via a standard female DB-15 connector. The AHRS is depicted in Figure 2.17.

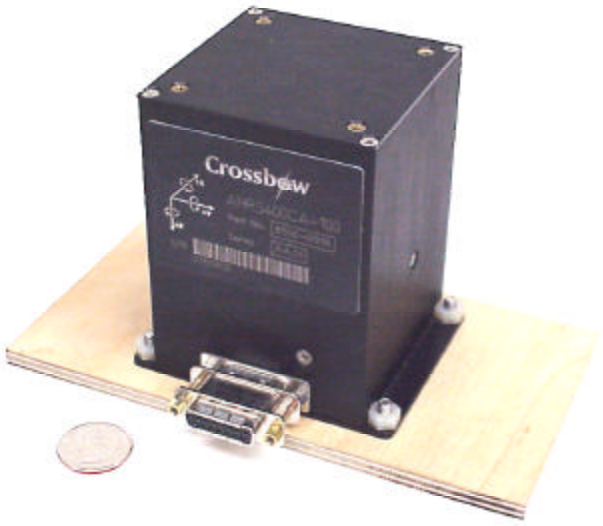

Figure 2.17. Crossbow AHRS400CA-100 Attitude Heading Reference System.

1. Hardware Description

The AHRS features silicon micro-machined accelerometers and gyroscopes and flux gate magnetometer. The AHRS400CA-100 is configured with a ± 2 g tri-axial accelerometer, a ± 100°/sec rate tri-axial gyroscope, a tri-axial fluxgate magnetometer and a temperature sensor. The angular rate sensors consist of vibrating ceramics plates that utilize the Coriolis forces of output angular rate independently of acceleration. The

tree MEMS accelerometers are surface micro-machined silicon devices that use differential capacitance to sense acceleration. The sensor outputs are converted to digital signals in a 14-bit A/D converter and then processed by the embedded microprocessor. The microprocessor provides serial output at 38400 baud via the RS-232 compliant interface. Analog output signals are provided by a 12-bit D/A converter. The AHRS system architecture is depicted in Figure 2.18.

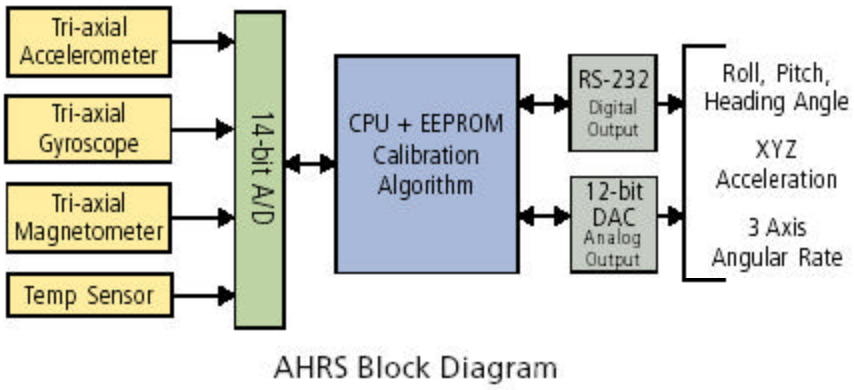

Figure 2.18. Crossbow AHRS System Architecture.

The AHRS has three sensor modes of operation, voltage, scaled sensor, and angle mode. In voltage mode, the analog sensors are sampled and converted to digital data with 1 mV resolution. The rate sensor, magnetometer and angle analog outputs are disabled in this mode though this data is still provided in the serial data stream. In voltage mode the Crossbow provides only un-scaled raw sensor output without any calibrations or corrections. When operating in the scaled mode the Crossbow AHRS400 is a classic inertial measurement unit (IMU). In the scaled sensor mode, the analog sensors are sampled and converted to digital data as before but then are temperature compensated, corrected for misalignment and scaled to engineering units. A factory calibration table for each sensor is stored in the AHRS non-volatile memory. The AHRS Kalman filter is not enabled in this mode so rate sensor bias values can be expected to change over time. Also stabilized pitch, roll and yaw angles are not available. The analog output signals are enabled in this mode. In the angle mode, the AHRS acts as a complete attitude and heading reference and outputs stabilized roll, pitch

and yaw angles along with the angular rate, acceleration and magnetic field information available in the scaled mode. The AHRS' Kalman filter operates in the angle mode to track the rate sensor bias and calculate the attitude angles. The attitude angles are computed by integrating the rate sensor outputs. The accelerometers are used to correct for gyro drift in pitch and roll and the magnetometers are used to compensate for drift in yaw. The serial data parameters for each of the sensor modes are presented in Table 2.6. The header and Checksum are passed as single bytes. In angle and scaled sensor modes all data (except temperature and time) are sent as 16-bit signed integers in twos complement format, most significant byte first. The temperature and timer data are sent as 16-bit unsigned integers.

Angle Mode	**Scaled Sensor Mode**	**Voltage Mode**
Header (255)	Header (255)	Header (255)
Roll Angle	Roll Angular Rate	Roll Gyro Voltage
Pitch Angle	Pitch Angular Rate	Pitch Gyro Voltage
Heading Angle	Yaw Angular Rate	Yaw Gyro Voltage
Roll Angular Rate	X-Axis Acceleration	X-Axis Acceleration Voltage
Pitch Angular Rate	Y-Axis Acceleration	Y-Axis Acceleration Voltage
Yaw Angular Rate	Z-Axis Acceleration	Z-Axis Acceleration Voltage
X-Axis Acceleration	X-Axis Magnetic Field	X-Axis Mag Sensor Voltage
Y-Axis Acceleration	Y-Axis Magnetic Field	Y-Axis Mag Sensor Voltage
Z-Axis Acceleration	Z-Axis Magnetic Field	Z-Axis Mag Sensor Voltage
X-Axis Magnetic Field	Temp Sensor Voltage	Temp Sensor Voltage
Y-Axis Magnetic Field	Time	Time
Z-Axis Magnetic Field	Checksum	Checksum
Temp Sensor Voltage		
Time		
Checksum		

Table 2.6. Sensor Mode Serial Data Parameters.

The data message uses a single header byte with a value of 0FFh. During the development of the serial decoder for the FROG controller it was discovered that this value appears numerous times within a single data packet. In order to decode the serial data it is necessary to check each byte to determine if it is the header byte. When a byte value equals 0FFh a checksum must be computed for the following 28 bytes (22 in scaled or voltage mode) and compared to the next byte to see if it contains a valid checksum. If the checksum matches then a single data packet has been found. If not then the process continues in a serial fashion (byte-by-byte). The use of a single header byte (who's value commonly appears within the data message) creates a computationally intensive and inefficient decoding job. This situation could be greatly improved by adding a second header byte to each AHRS output message. The probability of having two consecutive 0FFh's is extremely low and would greatly reduce number of computations required in order to decode the message. With a two-byte header each byte is compared to the byte that preceded it. When both bytes have a value of 0FFh a valid header has been found and the following bytes are almost certain to hold the data string. Crossbow Technology's cost estimate to change to a two-byte header exceeded the purchase price of the AHRS itself.

2. Timing Performance

The Crossbow AHRS400 was purchased principally as an AHRS for the FROG. As such, the performance in the angle sensor mode is of primary interest. The AHRS may be operated in either continuous update or polled mode. In the continuous mode data is streamed at the AHRS maximum rate. In the polled mode uses a challenge and response format in which the AHRS responds with a single update message for each "request for data" command received. During operation the ARHS processor runs in a loop – collecting data from the A/D converter and processing/formatting the data for output. The data is output to the user in a parallel process. Each data cycle consists of three tasks. First, the sensors are sampled. Second, the microprocessor processes the data and stores it in a set of registers for transfer to the serial output buffer. While the AHRS is waiting for the serial buffer to clear the processor will simultaneously sample the sensors again. Third, the unit actually transfers the data out to the RS-232 serial buffer. In the case of analog output the data is placed on the output pins immediately

after the data processing step (no serial buffer delay). The serial data is transferred only when the previous data packet has cleared the serial buffer. The AHRS continues to take data (and over-write the data in the output registers) while waiting for the output buffer to clear. Consequently, only about every third measurement is actually available over the RS-232 interface.

Crossbow Technologies only specifies a maximum serial update (75 Hz) and gives no other specific information on output performance. They do state that in continuous mode the system processor activity is highly deterministic and accurate timing information can be derived from the overall loop rate. Upon delivery of the AHRS a flight test was performed to assess the units suitability for use in the FROG. As AC-104 serial drivers were not yet written, the Gyroview software provided by Crossbow was used to capture and decode the AHRS output. Gyroview version 2.1 includes the ability to control the AHRS operating mode and includes provisions for real-time display and frequency domain (via fast Fourier transform) analysis of the sensor outputs. The Gyroview interface and sample output are depicted in Figures 2.19 and 2.20 respectively.

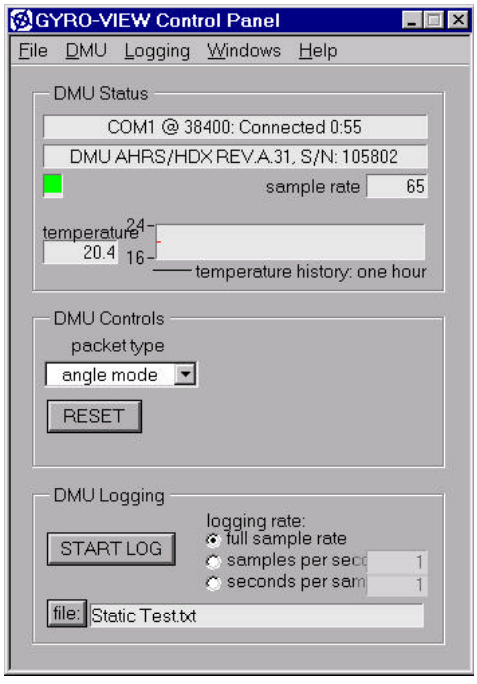

Figure 2.19. Gyroview Software User Interface.

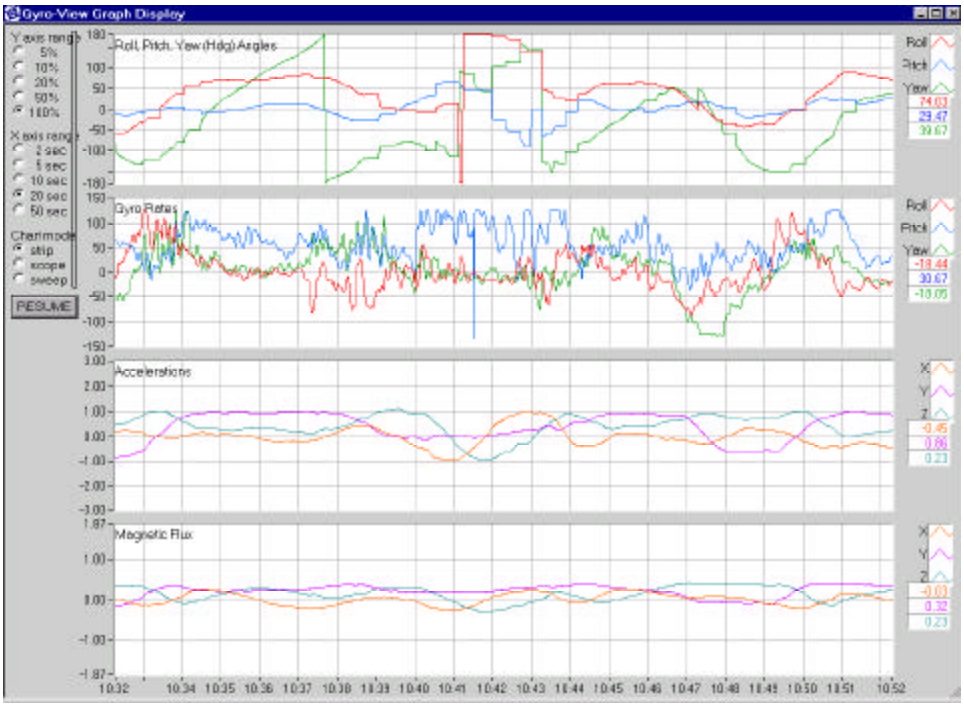

Figure 2.20. Gyroview Real-Time Data Display Capability.

Post flight analysis of the AHRS sensor and computed angle outputs indicated that the AHRS could provide the level of accuracy required for the flight controller and future control projects. Figures 2.21 and 2.22 depict the AHRS data for a double figure eight maneuver. The analysis of the Gyroview data did, however, reveal a highly erratic and inconsistent update rate, as shown in Figure 2.22. This discovery negates the utility of the Gyroview Fourier analysis module, as a consistent sampling interval cannot be maintained. CrossBow's technicians reviewed this finding and indicated that the timing irregularities in the captured data were due to unspecified problems in the Gyroview program itself and that the AHRS output was stable and consistent.

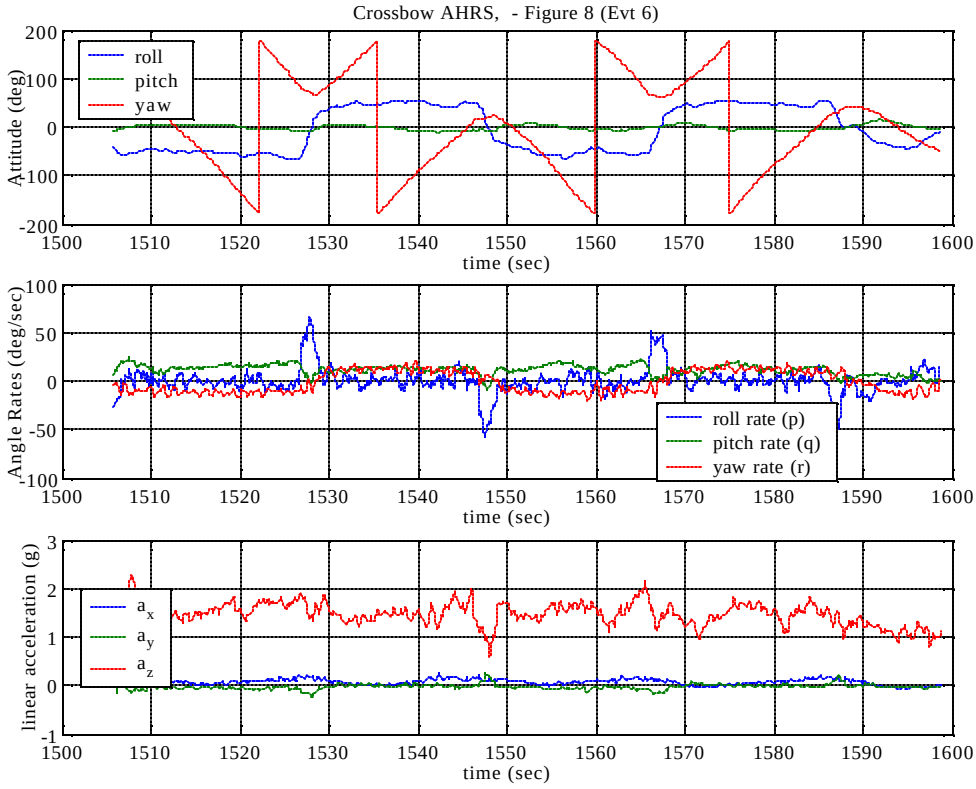

Figure 2.21. Gyroview Flight Test Data (Level Figure-8 Maneuver).

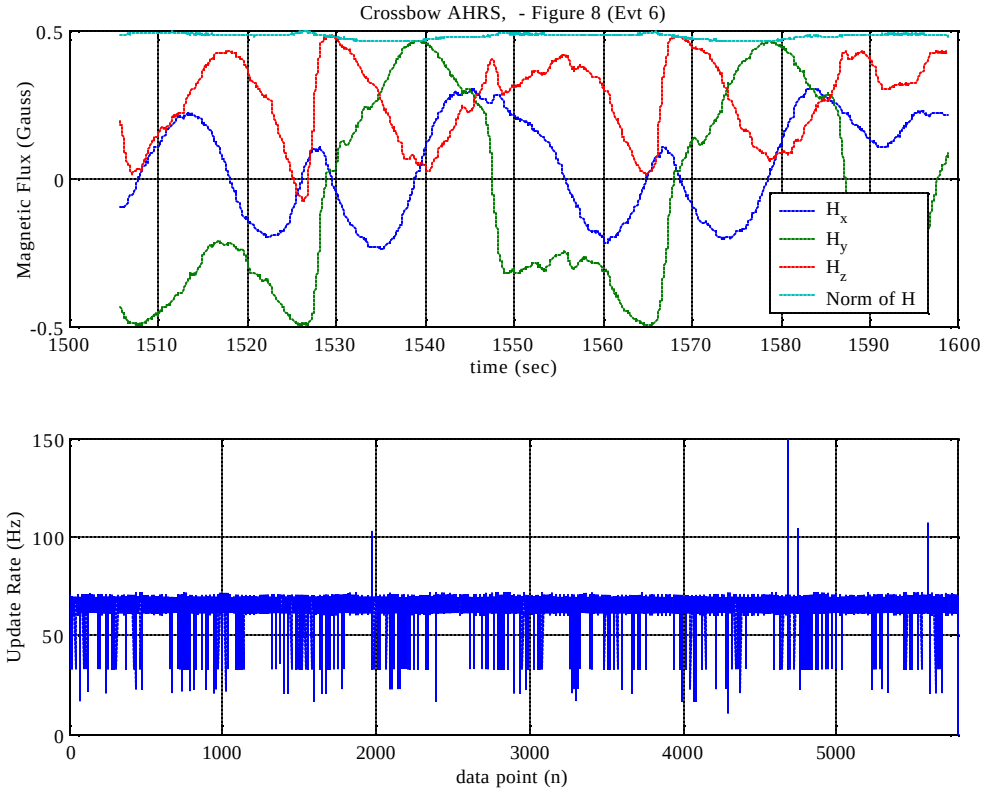

Figure 2.22. Gyroview Flight Test Data (Level Figure-8 Maneuver).

To test this assertion, and establish baseline timing performance specifications for use in our discrete time control system, we performed detailed timing analysis of the Crossbow AHRS400 serial output signals. Timing tests were performed in both the continuous and polled operating modes and in the angle and scaled sensor measurement modes. The AHRS serial output rate was set to the default value of 38400 baud for all measurements. The update rates, message durations, inter-message gaps and polled mode response latency were measured with a DSO-2102 PC—based digital storage oscilloscope, manufactured by Link Instruments, Inc. In the angle mode the 30-byte data message duration was a consistent 7.510 ms, as shown in Figure 2.23.

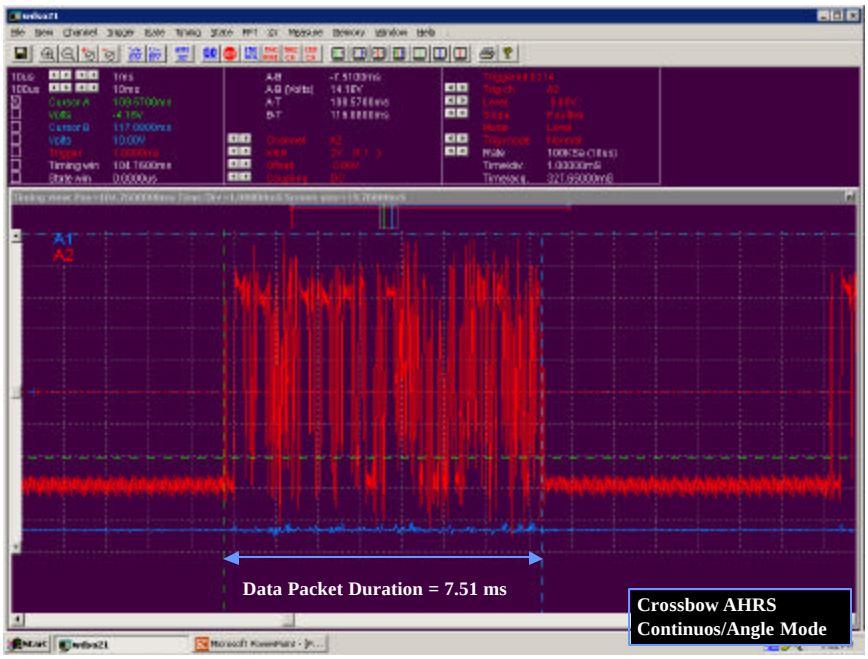

Figure 2.23. AHRS Angle Mode Serial Output Signal.

In continuous/angle mode the output period alternated between 14.350 ms and 16.400 ms resulting in an alternating frequency of 69.7 Hz and 61.3 Hz. The alternating output rate was highly stable and the slow-fast-slow-fast pattern was repeated without interruption, as depicted in Figure 2.24.

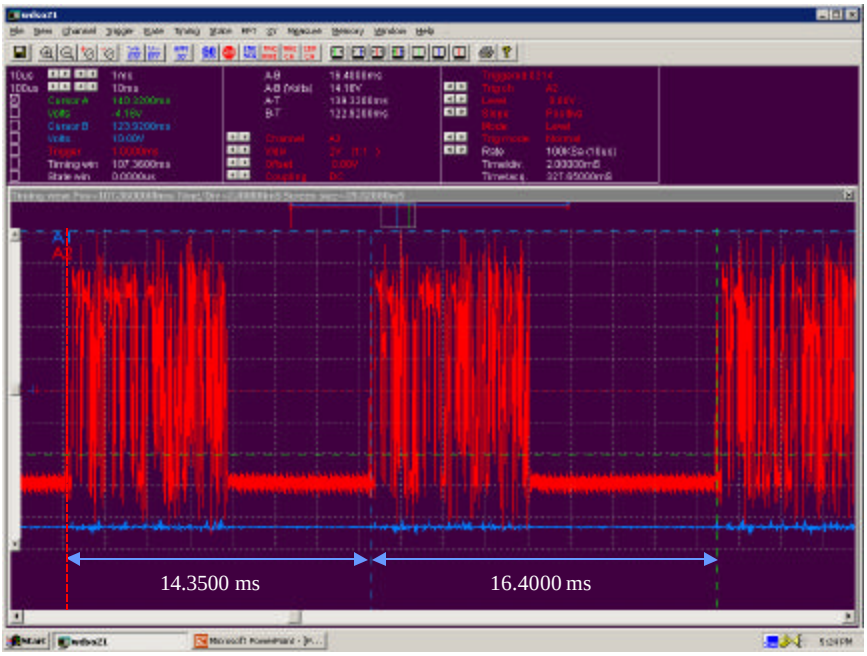

Figure 2.24. AHRS Continuous/Angle Mode Serial Output.

As the continuous/angle mode did not have a constant output frequency, the polled mode was tested. The AHRS can be placed in the polled mode by sending an ASCII "P" command. Once in the polled mode, the AHRS will send one complete data message in response to each request for data command (ASCII "G"). In order to stimulate the AHRS at precise frequencies a Motorola 68322 based TattleTale 8 micro-controller was programmed to send the request for data command (ASCII "G") over the RS-232 line to the Crossbow AHRS. The command signal and the AHRS response were then captured on the digital storage oscilloscope. Prior to each test, the micro-controllers RS-232 command rates were verified using the digital storage oscilloscope.

In order to determine the AHRS range of response latency a sample of 169 polling command/response events was recorded. The oscilloscope was set to trigger on the data request command line and the sampling rate was adjusted so that the scopes data buffer would only capture a single command/response event. The display was then set to accumulate the successive traces without refresh. The Tattletale polling rate was set to 1 Hz to ensure that each command/response event could be considered and independent event. Figure 2.25 shows the delay between data request initiation and AHRS response varies from a minimum of 13.998 ms to a maximum of 29.692 ms.

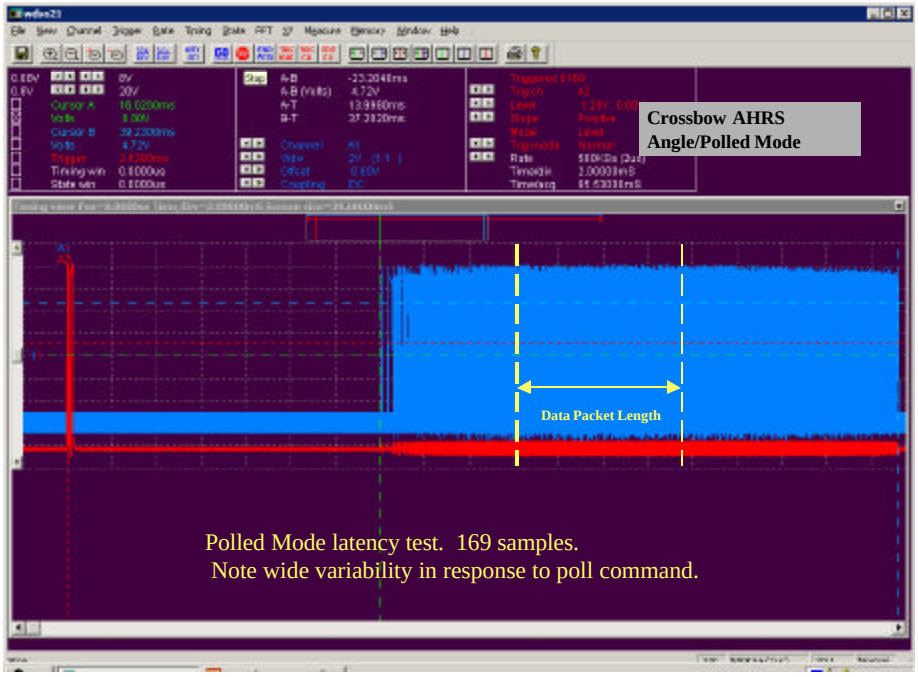

Figure 2.25. AHRS Polled / Angle Mode Serial Output Response Variation.

These results suggest that the AHRS should be capable of responding to polling commands at a maximum polling rate of 33.6 Hz. In order to confirm this result the AHRS was polled at various frequencies and the response behavior was observed. The AHRS was able to meet the response time constraints when polled at 20 Hz, Figure 2.26. Contrary to expectations, the AHRS could not respond fast enough when polled at 30 Hz. Figure 2.27 clearly shows two AHRS responses arriving within the same polling frame.

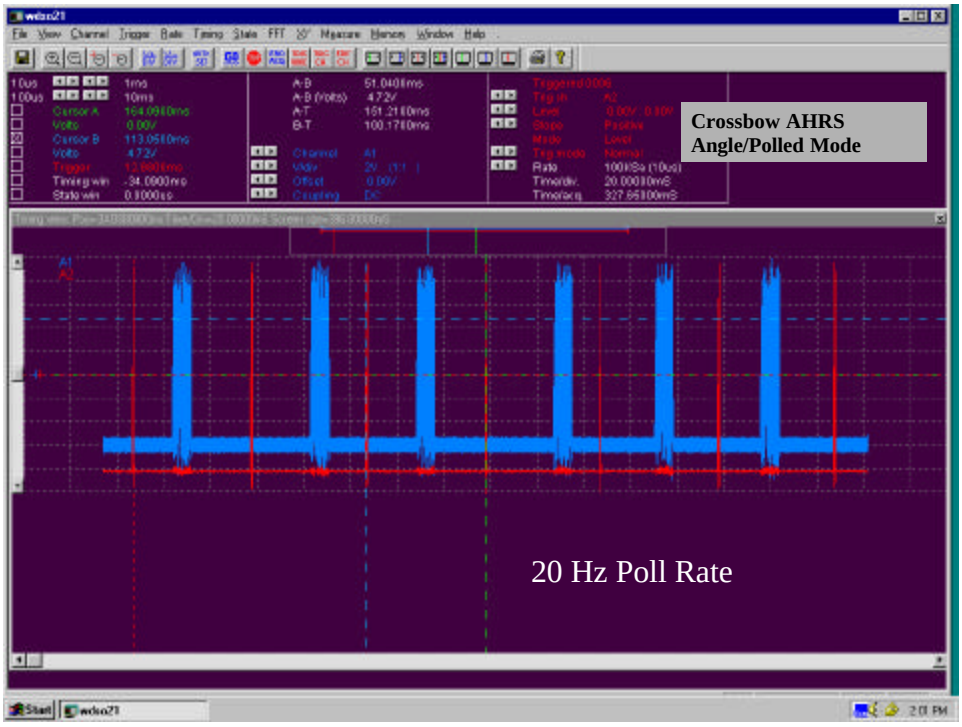

Figure 2.26. AHRS Polled/Angle Mode Response - 20 Hz Polling Rate.

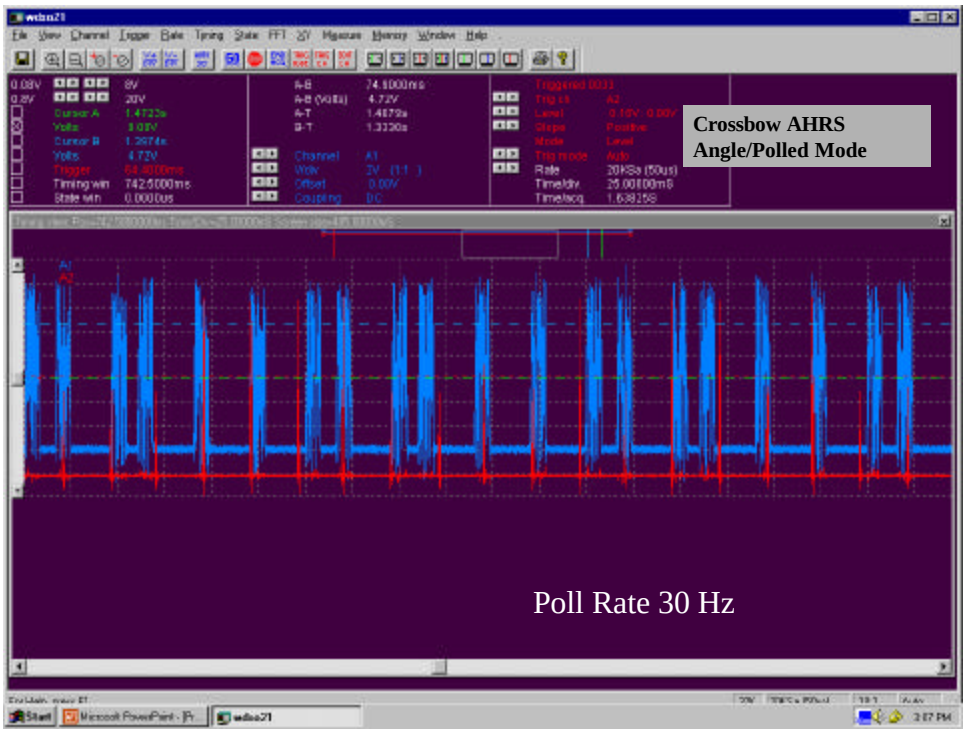

Figure 2.27. AHRS Polled/Angle Mode Response - 30 Hz Polling Rate.

Performance continued to degrade as the polling frequency was increased but then appeared to improve as the polling command rate approached the continuous mode's output rate. At 60 Hz the AHRS could meet the polling rate for ten to eleven cycles and then would drop a whole response, Figure 2.28. From these tests it was deduced that the AHRS internal data cycle is unaffected by the selection of polled or continuous mode. If by chance the polling command arrives just after a data sample has been posted for output the latency can be quite small. However, if the polling command arrives when the AHRS is posting new data to its registers a whole processing cycle may elapse before output is available. If the polling rate could be set to match the natural alternating frequency observed in continuous mode the AHRS would appear to function flawlessly.

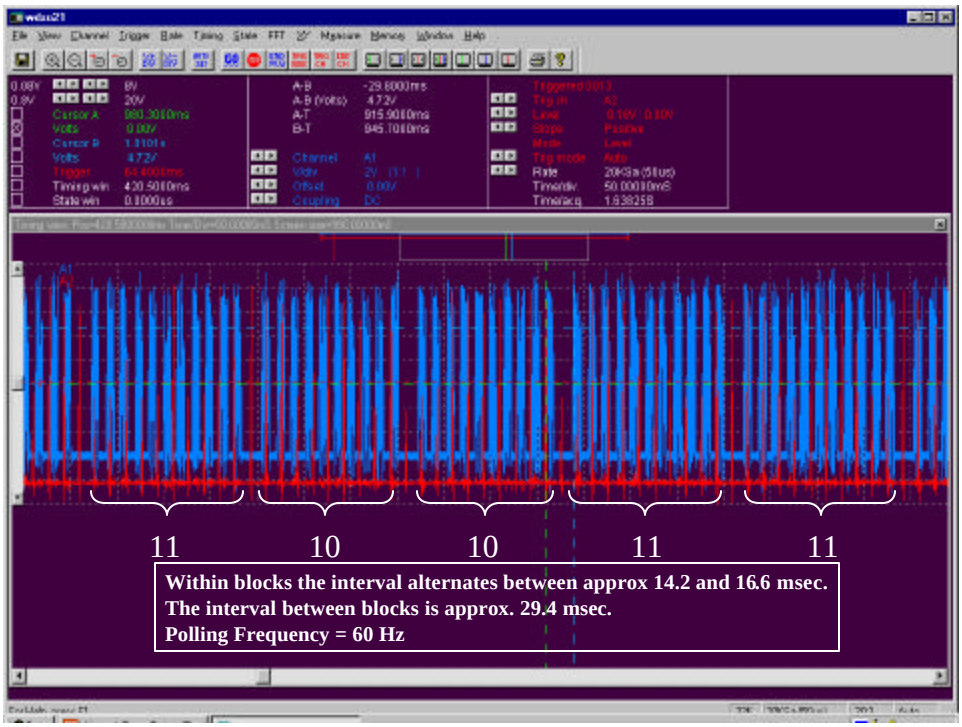

Figure 2.28. AHRS Polled/Angle Mode Response - 60 Hz Polling Rate.

These results were provided to Crossbow Technologies for comment. Crossbow advised that the apparent absence of determinism in the polled/angle mode response was considered a "feature". No explanation was offered for the alternating period observed in continuous/angle mode.

In light of these findings, timing performance was also evaluated in the scaled sensor mode. In scaled sensor mode the data message is 23 bytes long; vice 30 bytes in angle mode. The scaled mode message required 6.16 ms, at 38400 bps. In continuous/scaled sensor mode, the AHRS output period was a nearly constant 8.720 ms, which corresponds to a frequency of 114.6 Hz. There was no evidence of the alternating period observed in continuous/angle mode. The continuous/scaled sensor mode serial output performance is depicted in Figure 2.29.

Figure 2.29. AHRS Continuous/Scaled Sensor Mode Serial Output.

In the polled/scaled sensor mode the AHRS behavior was similar to that observed in polled/angle mode. A sample of 100 command/response events was collected in the same manner described for the polled/angle mode and is presented in Figure 2.30. The AHRS response delay varied from a minimum of 8.638 ms to a maximum of 16.480 ms. A maximum permissible polling rate for polled/scaled sensor mode was not determined.

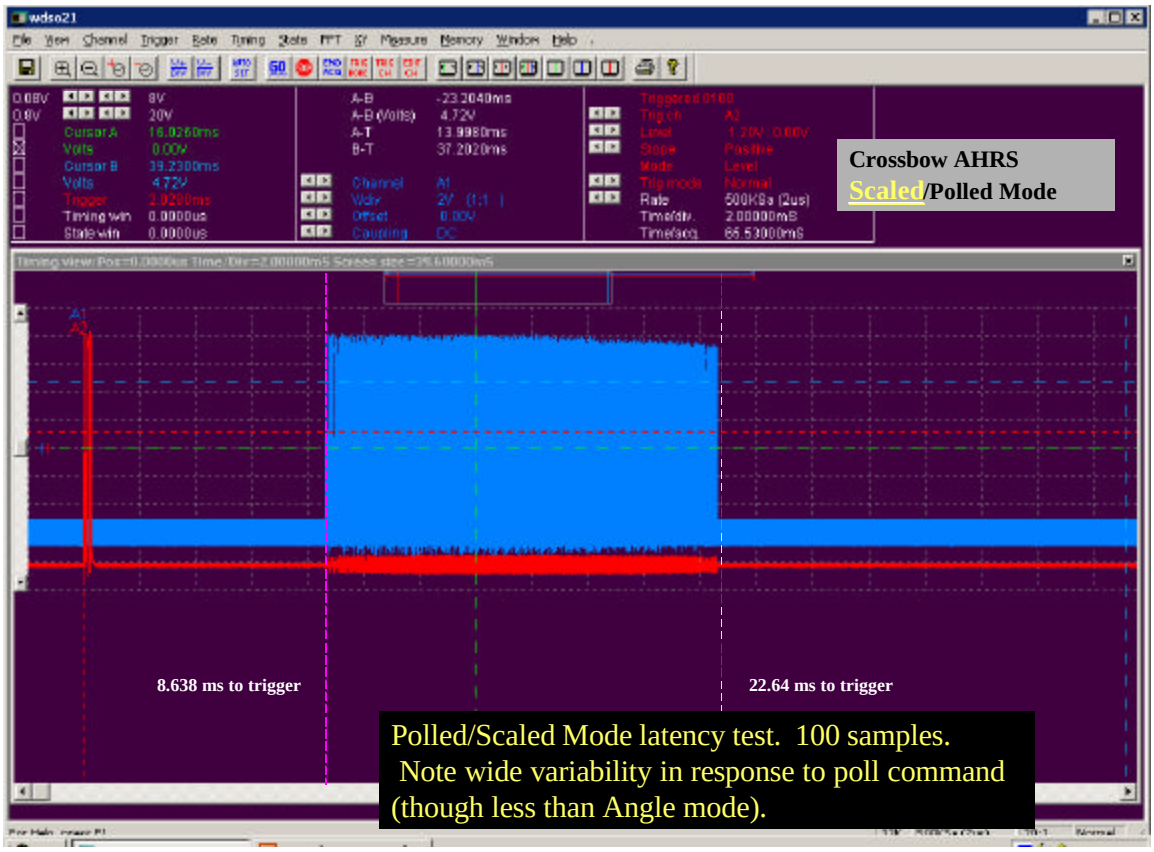

Figure 2.30. AHRS Polled / Scaled Sensor Mode Serial Output Response Variation.

3. AHRS Noise Output

The noise characteristics of the AHRS were evaluated for future incorporation into the FROG plant model. The AHRS was positioned on a stable concrete slab covered with a vibration isolating padding. The AHRS was initialized and allowed to operate for several minutes to allow the Kalman filters to converge on the gyro biases. Data was collected with Gyroview software and post processed in Matlab®. The mean pitch, roll and yaw rates were on the order of 0.005 degrees per second with an average standard deviation range between 0.079 and 0.088 degrees per second. The standard deviation of the orientation angles ranged from a low of 0.019 to a high of 0.025 degrees. The standard deviations for the three accelerometers ranged from a low of 0.00033 to a high of 0.00045 g's. The standard deviation of magnetometer output ranged from 0.00085 to 0.0011 Gauss. The computed mean values, standard deviations and covariance matrices for each set of parameters are presented in Appendix B. The observed ambient noise values are suitable for control system projects on the FROG and

should be incorporated into the FROG plant model prior to further controller development.

III. DIGITAL FLIGHT CONTROLLER

The FROG autopilot was designed using classical feedback methodologies. The autopilot was required to provide the following functions: altitude control via elevator, speed control via throttle and heading control via ailerons. The autopilot was also required to incorporate a yaw damper. The first step in the control system design process was to model the plant (system to be controlled) as accurately as possible. The non-linear FROG aerodynamic flight model used in this effort was developed, evaluated and refined by several previous students, including Hallberg [Ref. 3], Papgeorgiou [Ref. 4] and Pollard [Ref. 5]. The model had been developed for a single trimmed speed of 88 fps so the autopilot was designed around this speed. The FROG aerodynamic model did not incorporate the control servo dynamics so these had to be added before the flight controller design could be started. The autopilot was designed using MatrixX version 6.2.2. Once completed the autopilot was demonstrated via hardware in the loop (HITL) using the TS-75 servo actuators in the Controls Laboratory. The autopilot provided stable and reliable performance with the TS-75 HITL.

A. SERVO TEST SET CHARACTERIZATION

The first step in the flight controller design was to develop a control servo model to incorporate into the FROG plant model. Once the actuator dynamics were known a second order servo model was developed. Initial development work on the FROG autopilot was conducted in the NPS Aeronautical Engineering Department's Controls Systems Laboratory. During initial autopilot development the airplane's servos had not yet been instrumented for position feedback. Servo dynamics and hardware in the loop (HITL) testing was performed using a surrogate servo test set that interfaced with the AC-104 computer. Only after the prototype autopilot had been completed and demonstrated in the Lab was the necessary instrumentation installed on the FROG control surfaces to evaluate the airplane's true servo response. The FROG servos were then tested on the airplane to determine their dynamics and the overall control system temporal response characteristics.

The dynamic response characteristics of the miniature electric servos, installed in the servo test set, were determined using the data acquisition and hardware control

capabilities of the MATRIX$_X$/AC-104 system. Each servo test set was equipped with four Sys3000 TS-75 miniature electric servos, manufactured by Tower Hobbies. The servos were mounted in a metal tabletop cabinet and were equipped with pointers so that their position could be readily observed, Figure 3.1.

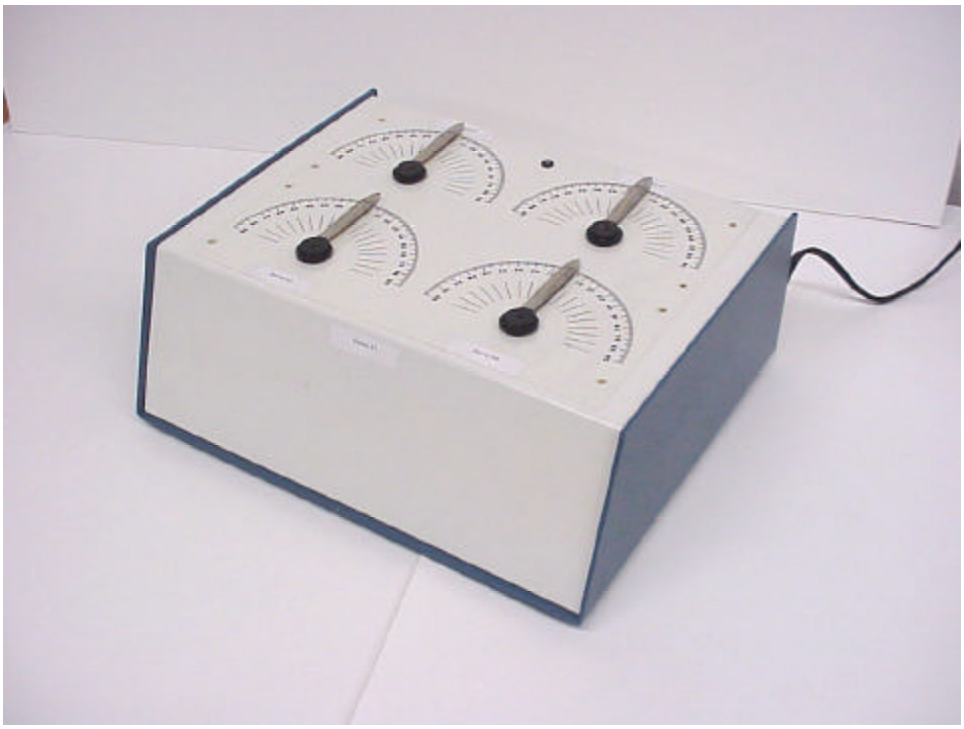

Figure 3.1 Servo Test Set.

Each TS-75 servo had been modified so that its internal feedback voltage could be monitored on an external DIN-50 connector. A second 50-pin DIN connector was provided to permit the input of servo PWM command signals. The laboratory HITL system is depicted in Figure 3.2.

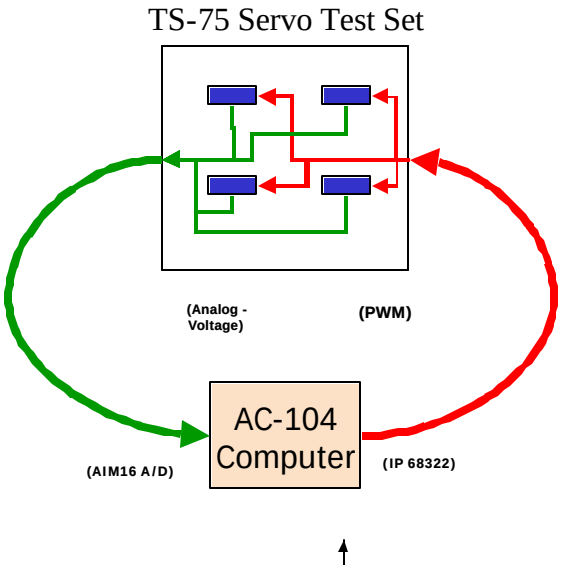

Figure 3.2. Control Systems Laboratory Hardware in the Loop System.

Matrix$_X$ version 6.2.2 was used to create a test system to permit servo dynamics assessment and control of the servos in the test set. The SystemBuild servo test system is depicted in Figure 3.3. The servo commands were input in degrees (± 90°) and then converted to pulse width values by a first order transfer function. In order to synchronize the physical servo pointer position with the output command, the Deg-to-PWM transfer function employed a variable slope and intercept that could be adjusted at run time. The resulting PWM commands were sent to the IP-68332, which generated the PWM signals for the TS-75 servos. The servo feedback voltage was digitized by the AIM16, 16-bit A/D converter and converted to degrees using the following formula:

$$\frac{180^\circ}{V_{+90^\circ} - V_{-90^\circ}}\left(V_{out} - V_{0^\circ}\right) \qquad (3.1)$$

The values of the minimum, maximum and zero degree voltages could be edited during run-time to permit calibration of the V-to-Deg transfer function.

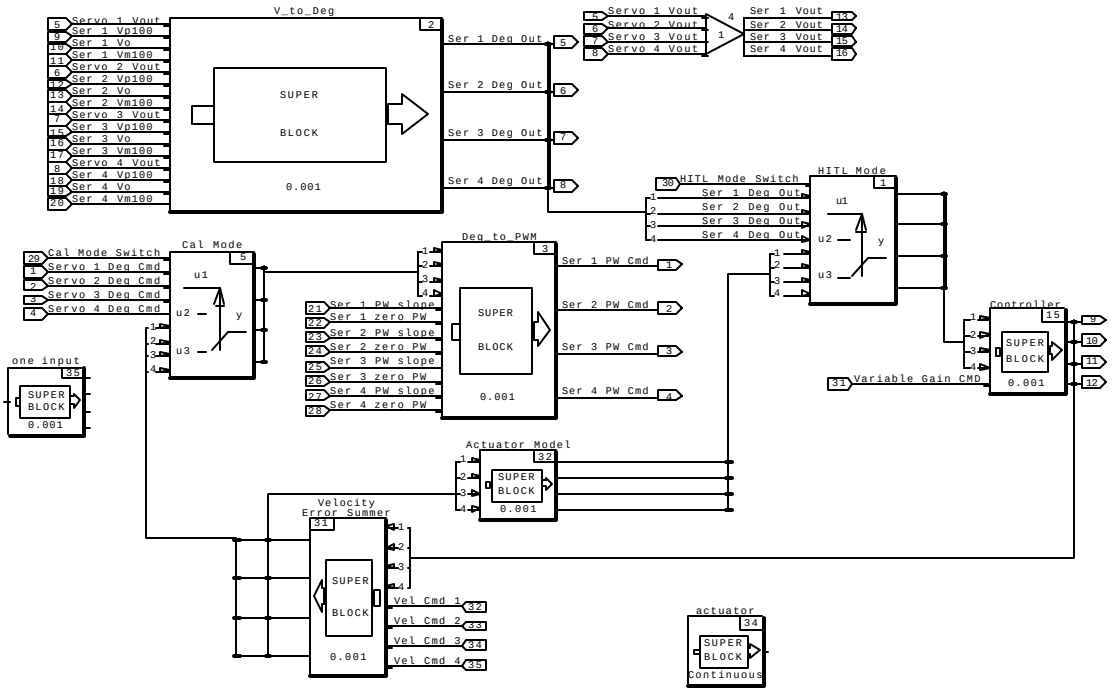

Figure 3.3. SystemBuild Servo Dynamics Test System.

The controller included a run-time graphical user interface, which allowed the user to input servo position commands and monitor servo position feedback. Controls were included to permit real-time adjustment of the command and feedback transfer functions. The servo test system calibration mode GUI depicted in Figure 3.4.

The simulation was Autocoded with a minor frame size of 0.001 seconds (Scheduler frequency = 1KHz). The RealSim Data Acquisition Editor was used to capture the controller command outputs and raw and processed servo responses for posttest analysis.

Once the test system was in place the actuators were stimulated with large and small STEP inputs in order to observe the plant dynamics. Limited data was available for the TS-75 servos. The TS-75 servos have a specified torque of 110.00 oz-in and a response rate of 0.19 seconds per 60°. Though small variations existed between individual actuators it was determined that the natural frequency was 38.73 rad/sec and the damping ratio (V) was 0.426. During testing the servo position feedback data showed a repeatable periodic offset. The offset was visible in each of the STEP

response plots and was observed on more than one servo test set. These position offsets occurred as the actuators slowed and approached the commanded position and could not be characterized as noise. It is unclear if the offsets originated within the internal servo feedback circuit or were introduced due to a binary error within the AIM16 A/D. The servos response to a 10° STEP input is depicted in Figure 3.5.

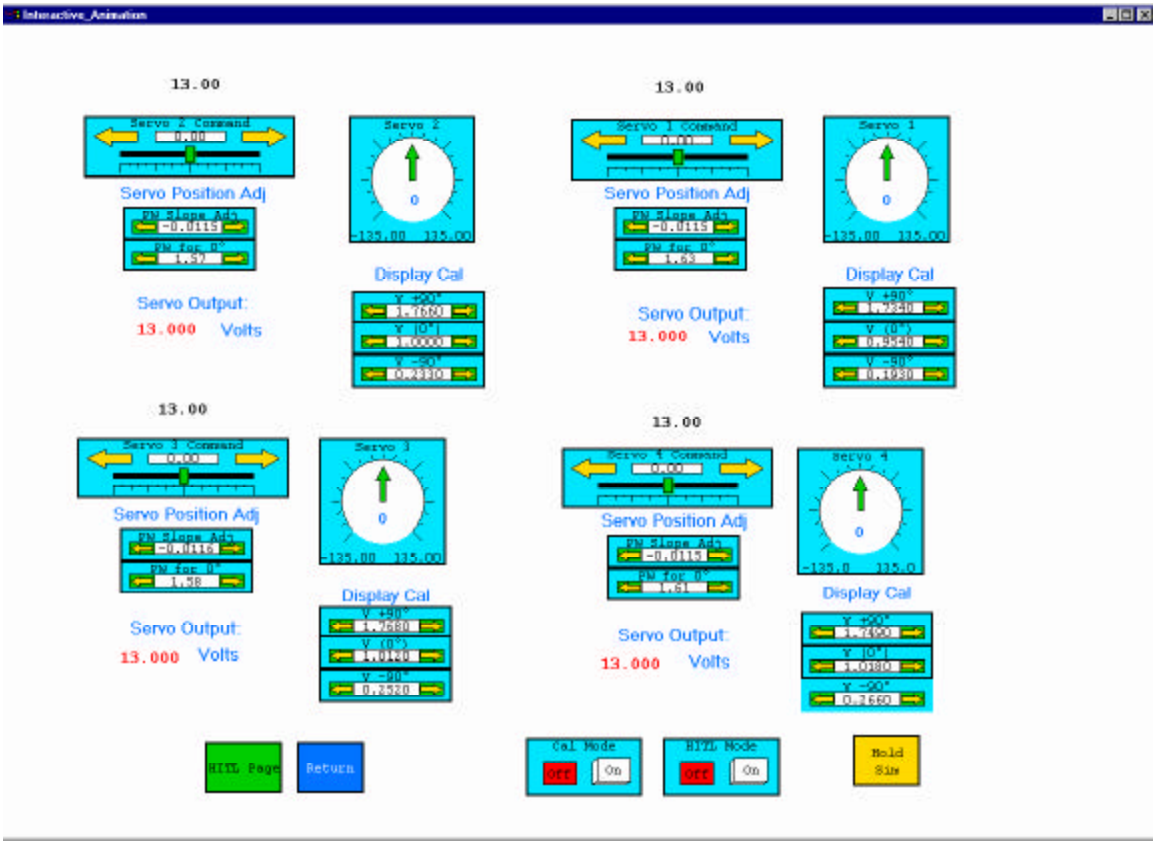

Figure 3.4. Servo Test System Calibration GUI.

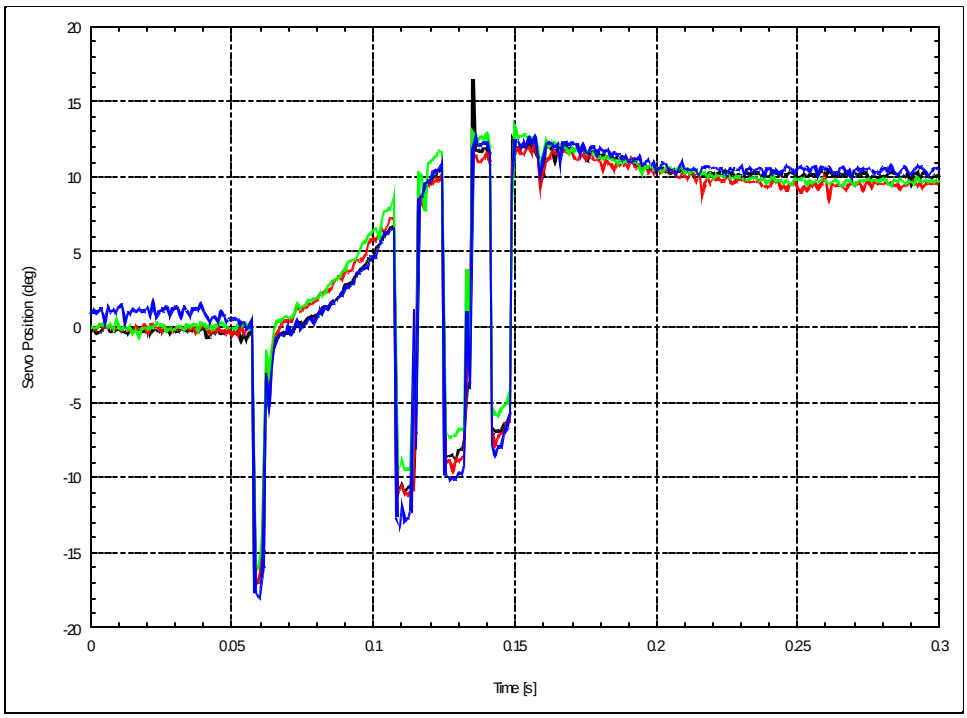

Figure 3.5. TS-75 Servo Response to 10° STEP Input.

The servo response to large STEP commands (greater than approx. 30°) revealed the presence of an internal rate limiter. The servo rotation rate was limited to 346° per second, which was reached after about 20° of rotation. The dynamic response data was used to create a 2nd order servo model, Figure 3.6.

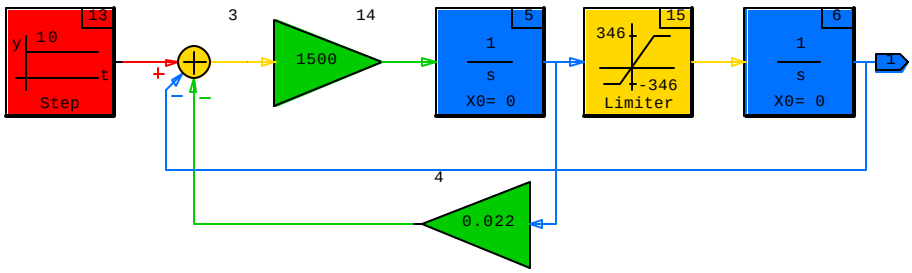

Figure 3.6. Second Order Model of TS-75 Servo.

To evaluate the accuracy of this model it was incorporated into a simple speed control system. The controller was then linearized and a Bode analysis was performed to determine the gain and phase margins. The Bode analysis indicated that the speed controller had a gain margin of 30.37 dB @ 6.6 Hz and the phase margin was 88.6° @

0.179 Hz. The speed controller was then modified to permit the TS-75 servos to be inserted as HITL in place of the 2nd order servo model. The controllers gain margin was experimentally determined by incorporating a variable gain in the feedback loop. A separate Speed Controller GUI was developed to permit monitoring of the servo response and control of the feedback gain, Figure 3.7.

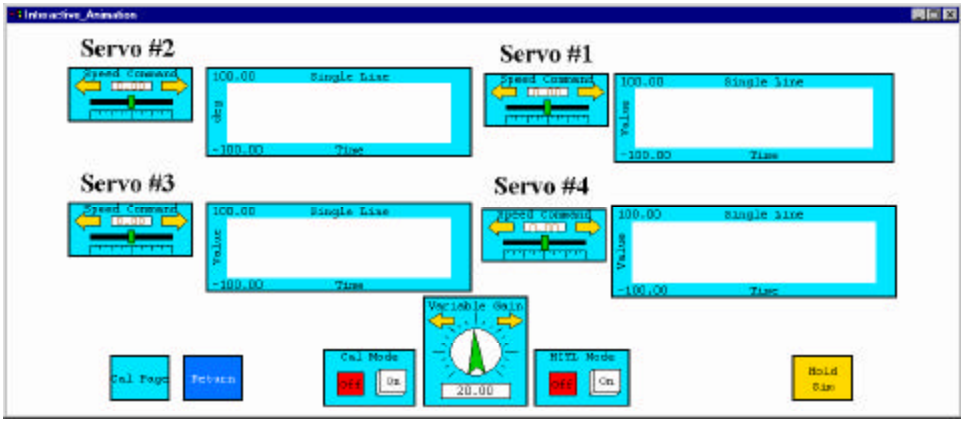

Figure 3.7. RealSim Speed Controller GUI.

The gain margin was determined by increasing the feedback loop gain until the controller became unstable. Three of the test servos exhibited a gain margin of 26.24 dB while the fourth had a gain margin of 27.42 dB. These gain margin values are in good agreement with the predicted value of 30.37 dB (13.6% and 9.8% difference respectively). The servo response to varying feedback gain is depicted in Figure 3.8.

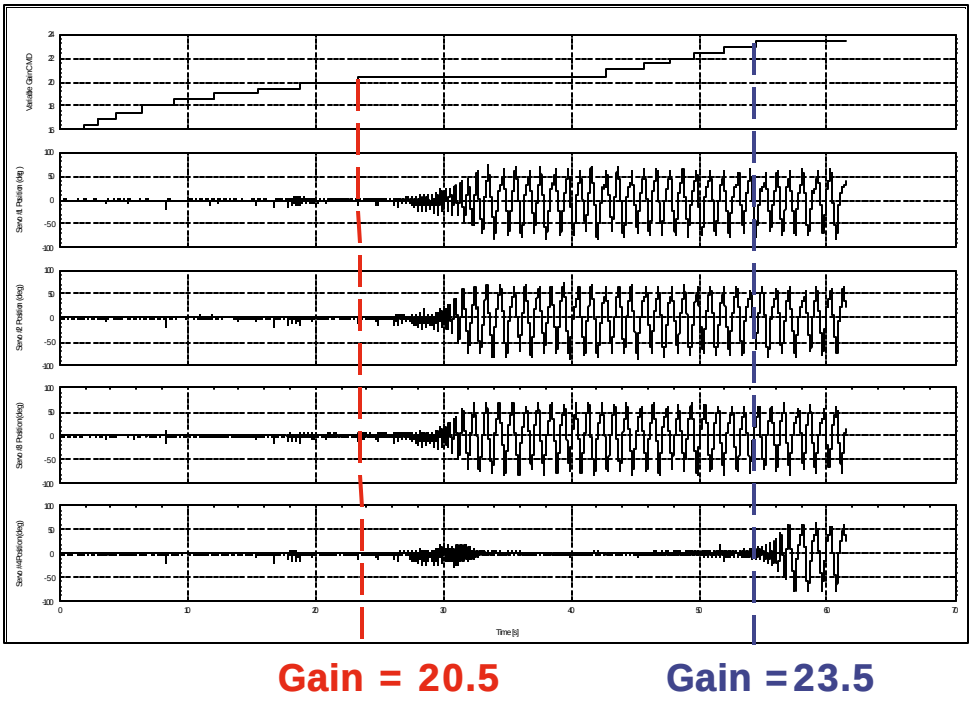

Figure 3.8. TS-75 Response to Increasing Speed Controller Feedback.

B. DIGITAL FLIGHT CONTROLLER DESIGN

The flight controller (autopilot) was designed to provide the following functions: altitude control via elevator, speed control via throttle and heading control via ailerons. The autopilot was also required to incorporate a yaw damper. MatrixX version 6.2.2 and the RFTPS were used to design and test the controller. The completed the autopilot was demonstrated via hardware in the loop (HITL) using the TS-75 servo actuators.

1. Design Methodology and Performance Criteria

The autopilot was designed using classical inner/outer loop feedback methodologies. The control channels were designed in the following order: yaw damper, altitude control on elevator, heading control on aileron and finally speed control on throttle. At each stage the inner and outer loop performance was evaluated using the non-linear flight dynamics model of the FROG in MatrixX. The basic design procedure for each controller was the same.

- Design the feedback (compensation) loop in SystemBuild.
- Linearize the new system (plant plus compensator) in Xmath.
- Evaluate the stability of the new system via root locus.

- Determine the systems phase and gain margin (Bode plot).
- Determine systems response to a Step command.
- Adjust the compensation gains and repeat steps two through five until the specified design criteria were met.
- Once satisfied with the linearized systems performance check the non-linear system by inserting a Step source into the SystemBuild simulation and evaluate the systems performance.

The SystemBuild flight controller is presented in Figure 3.9. The servo calibration GUI was modified and augmented with a separate autopilot control GUI, which is depicted in Figure 3.10

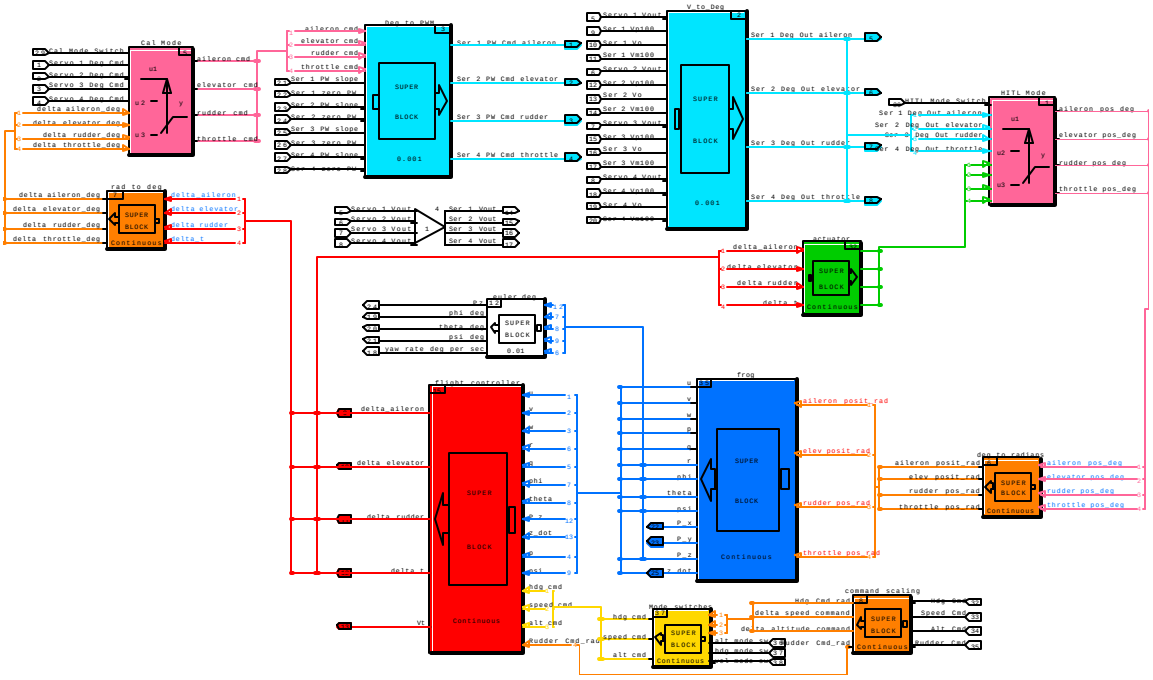

Figure 3.9. FROG SystemBuild Flight Controller Model.

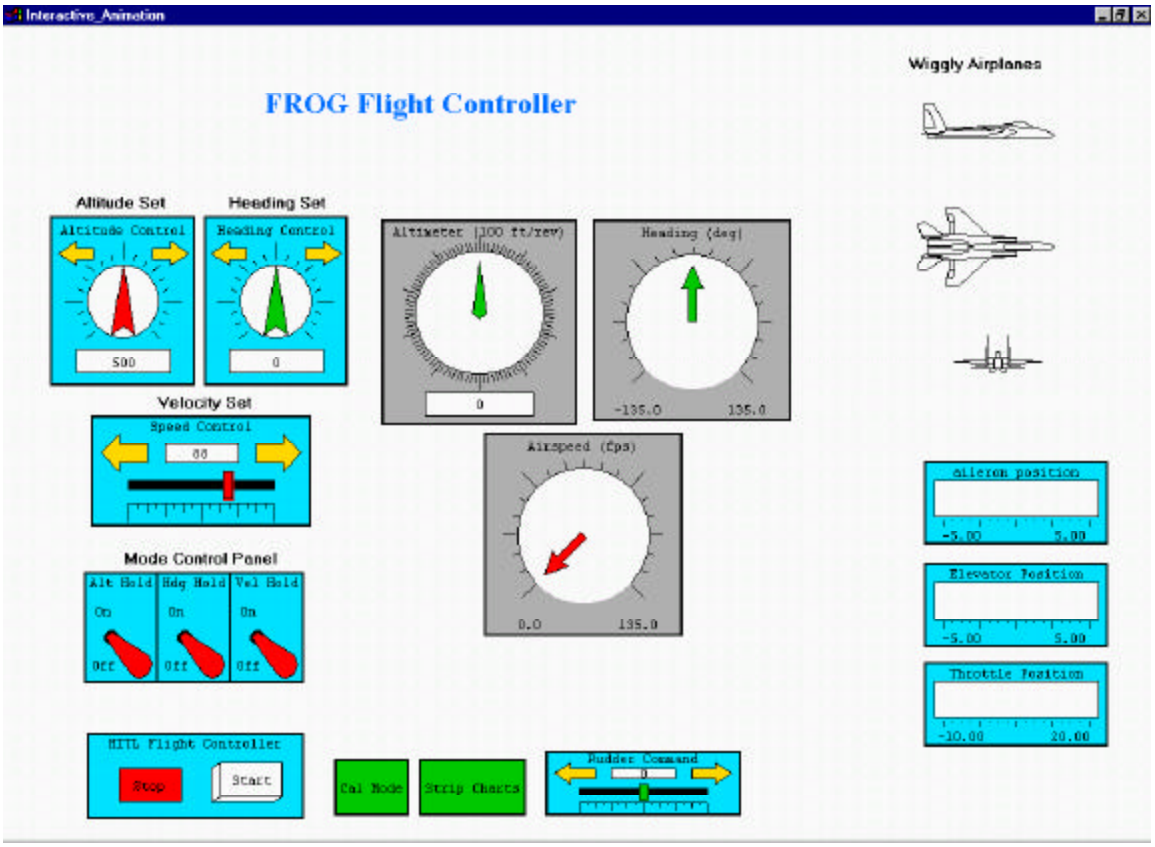

Figure 3.10. RealSim Flight Controller Display.

2. Yaw Damper

The yaw damper was designed to provide stability augmentation by feeding back commanded angle of bank (ϕ) (from the heading controller) divided by true airspeed (Vt). Commanded angle of bank was chosen, in lieu of a washout filter, so that yaw damper would not attempt to counter a commanded turn. The compensator chosen was of the proportional/integral type (PI) with an additional scaling gain. As specific yaw damper performance requirements were not given a target of 6 dB gain margin and 60° phase margin was used. The yaw damper block diagram is presented in Figure 3.11. The linearized yaw damper displayed a gain margin of 13.1 dB @ 38.39 rad/sec and a phase margin of 76.1° @ 7.73 rad/sec.

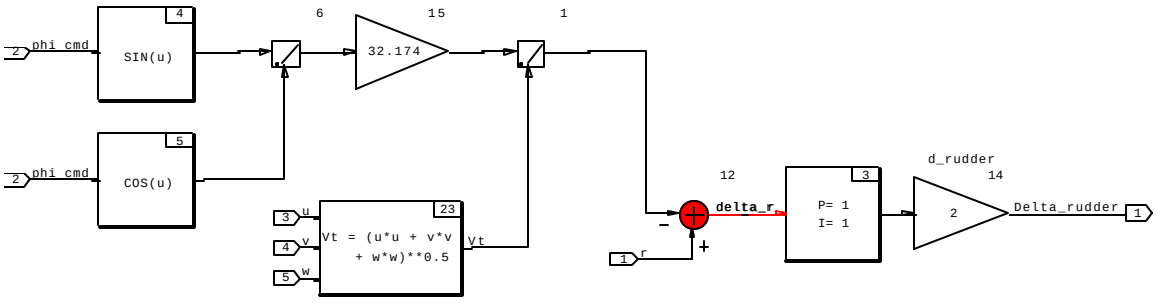

Figure 3.11. Yaw Damper SystemBuild Block Diagram.

Figure 3.12 presents the FROG's response (with simulated actuators) to an 11.4° (0.2 rad) step input to the rudder position. The yaw damper immediately counter acts the rudder disturbance and bank angle is returned to zero within 14 seconds. The airplane's heading exhibits a maximum deviation of 2.3° and is restored within approximately six seconds.

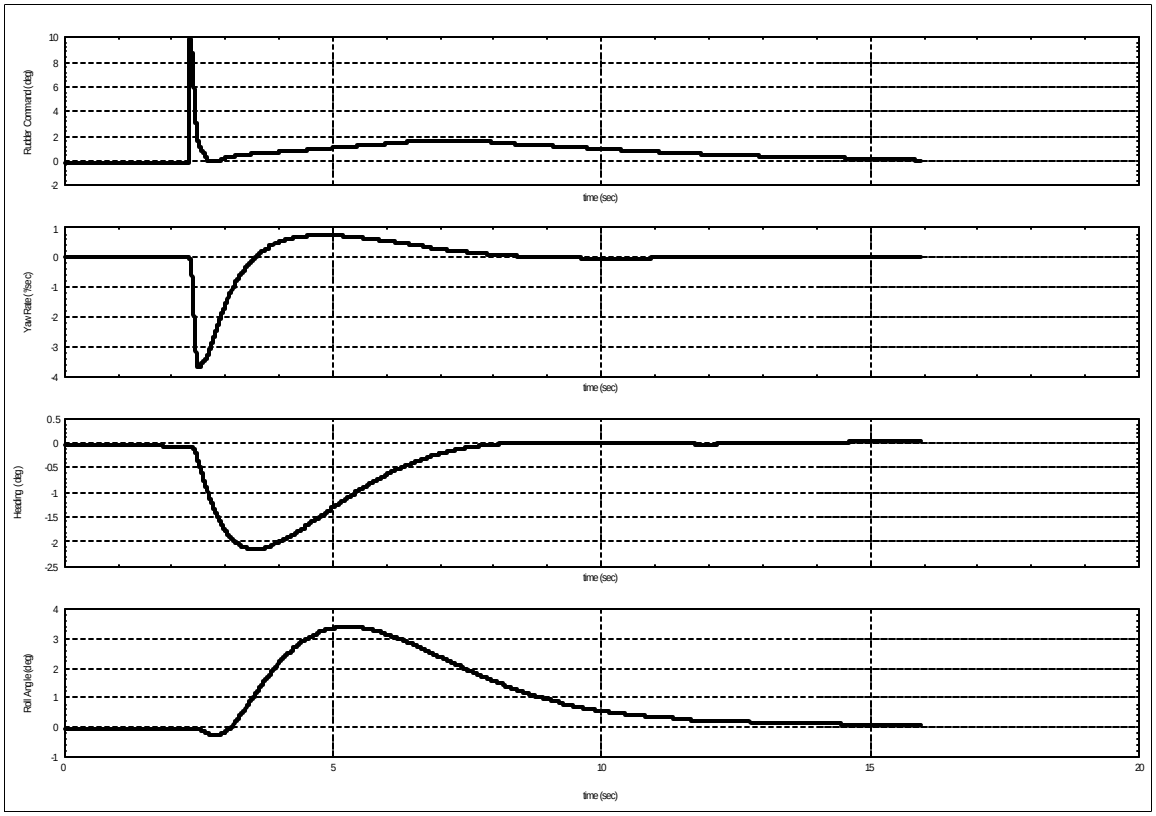

Figure 3.12. Yaw Damper Response (Simulated Actuators).

Figure 3.13 presents the HITL performance of the yaw damper with HITL; when subjected to a 10° rudder change (from –5° to +5°). The heading reaches a maximum deviation of approximately 2.2° after 1.5 seconds and has returned to within 1° of the original value by four seconds. It is noted that the presence of a small periodic signal was observed in the TS-75 servo's output. This small disturbance signal degraded the yaw damper's performance somewhat when compared to the simulation. The disturbance signal was observed also observed when the controller was disconnected and was therefore not an undesirable controller response. Other possible feedback schemes include yaw rate feedback, roll rate feedback and sideslip feedback. Each of these has significant influence on the short period and spiral modes of an airplane and merit further investigation for use on the FROG.

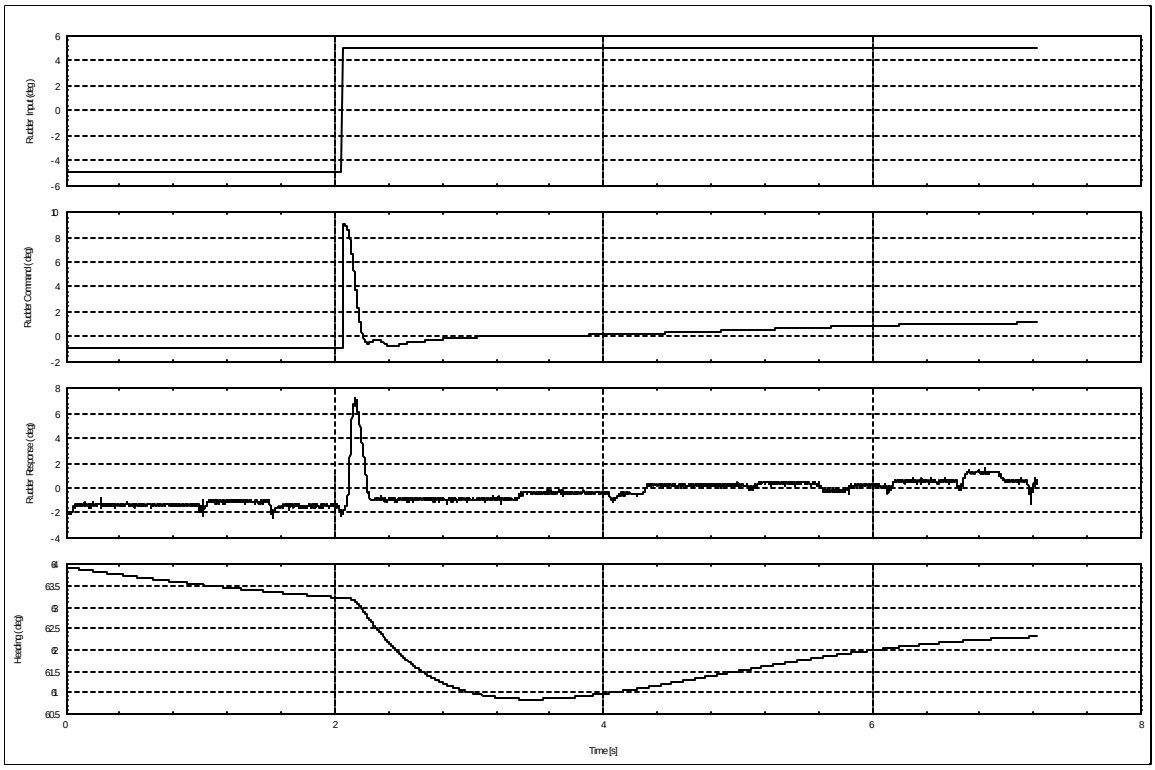

Figure 3.13. FROG Yaw Damper Performance with TS-75 Actuators.

3. Altitude Controller

The altitude controller was designed to satisfy the following criteria:

- The closed-loop system must be stable.

- The elevator command loop bandwidth should not exceed 7 rad/sec.
- Minimum gain margin of 6 dB and phase margin of 30°.

Additionally, the controller was designed to limit Step response overshoot of no more than 10% and a have a rise time of about 10 seconds. The inner loop consisted of a proportional/derivative (PD) compensator using pitch rate (q) and pitch attitude (θ) feedback. The outer loop employed a PID compensator with altitude (P_z) and altitude rate ($\dot{h}$) to produce commanded pitch attitude. The altitude controller block diagrams are presented in Figure 3.14.

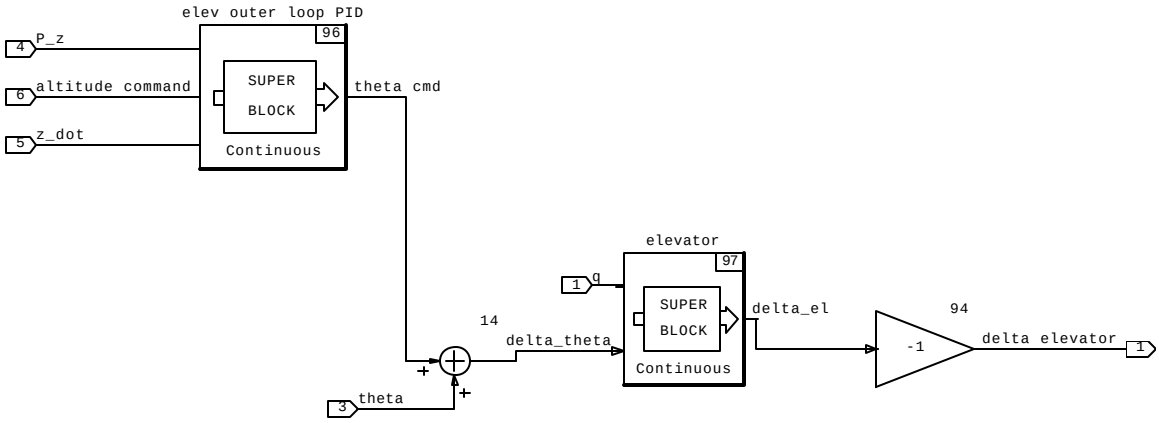

Figure 3.14. Altitude Controller SystemBuild Block Diagram.

The altitude controller exhibited a gain margin of 13.86 dB @ 16.46 rad/sec and a phase margin of 58.6° @ 6.50 rad/sec. The linearized model's performance met the 10% overshoot and 10 second rise-time constraints but the non-linear simulation slightly under performed. The non-linear controller exhibited an over shoot of approximately 15% and a rise time of approximately 11 seconds. The HITL response is presented in Figure 3.15. The HITL overshoot was approximately 14% with a rise time of approximately 9 seconds. A significant periodic oscillation, with a frequency = 0.83 Hz, was observed on the elevator command and elevator servo output channels. Peak-to-Peak variations of approximately 1° were observed but the variation did not appear to affect the controller's altitude performance. While this behavior was acceptable for the altitude controller it would likely produce unacceptable variations in pitch attitude that would adversely affect the ability to stabilize optical sensors. The source of the variation was not determined.

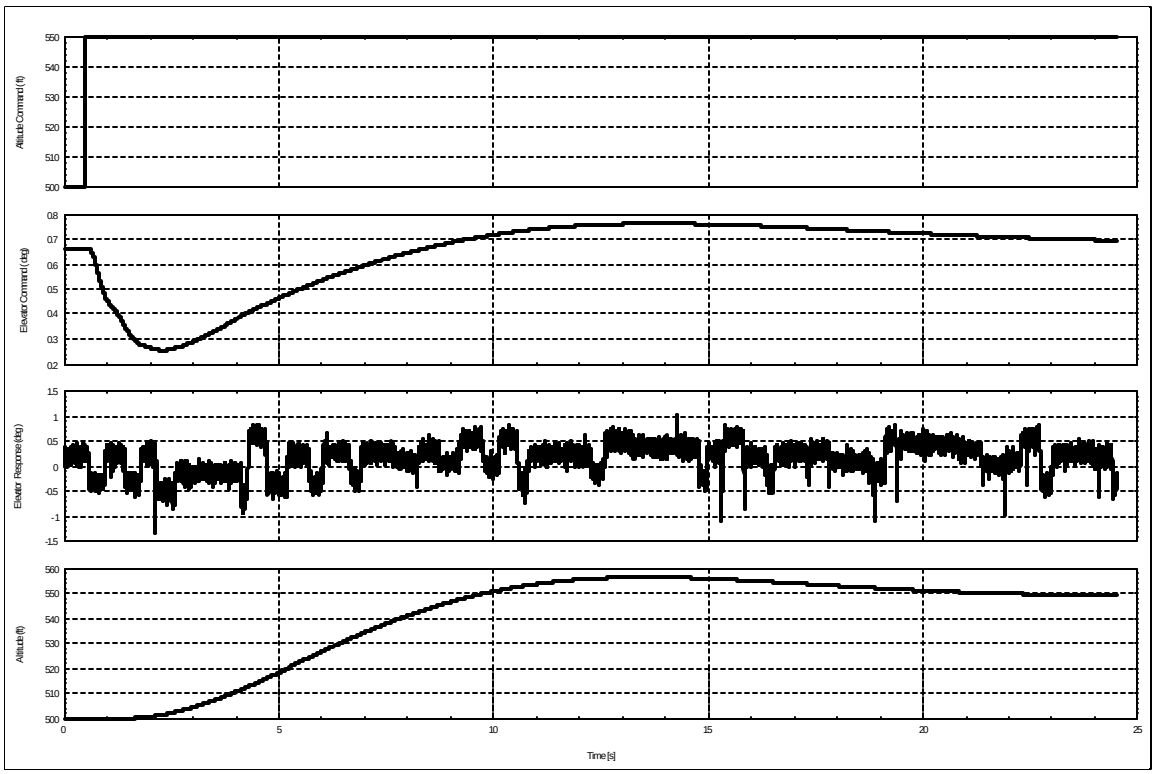

Figure 3.15. Altitude Controller Response with TS-75 Servos.

4. Heading Controller

The heading controller was designed to meet the same performance requirements as the altitude controller. The heading controller design incorporated inner, middle and outer loop feedback loops and is depicted in Figures 3.16 and 3.17. The outermost loop took commanded heading (ψ_c) and produced an angle of bank command (ϕ_c). The middle loop took ϕ_c and converted it into a roll rate command (p_c) while the inner most loop converted p_c to an aileron deflection command (δ_a). The inner loop consisted of a PI controller. The middle loop consisted of a PID controller and the outer loop consisted of a PI controller. The outer loop also included a limiter that restricted ϕ output to less than 0.5 radians (approximately 28.5°). Each loop layer was designed such that the crossover frequencies were separated by at least one decade.

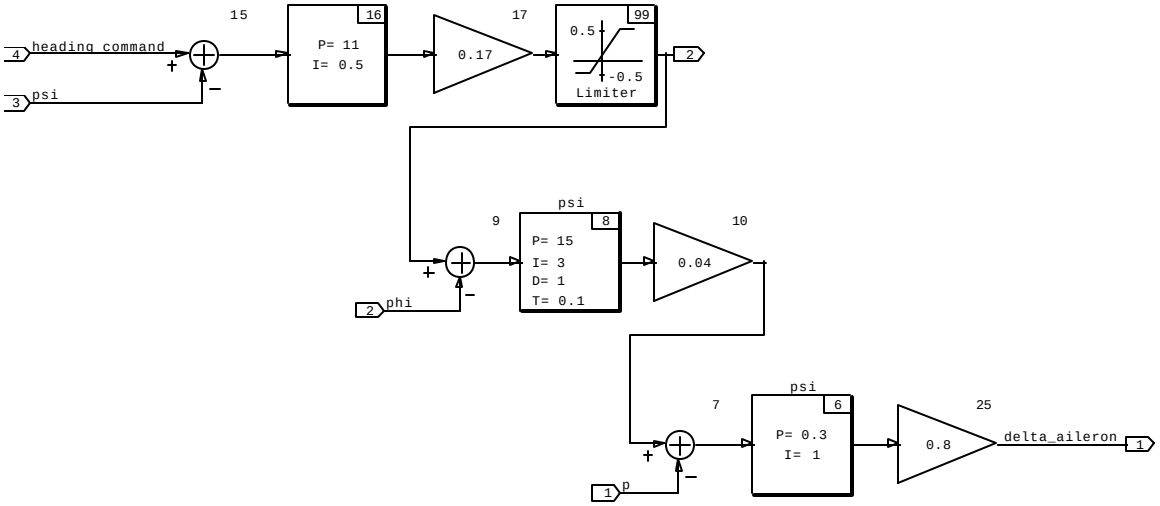

Figure 3.16. Heading Controller System Build Block Diagram.

The heading controller exhibited a gain margin of 14.74 dB @ 38.49 rad/sec and a phase margin of 80° @ 5.97 rad/sec. The heading controller's response, with simulated actuators, to a step heading command of 29° is presented in Figure 3.17. The airplane heading (ψ) exhibits a 10% overshoot and a rise time of approx. 5.5 seconds. It can also be seen that the yaw damper responds as expected to produce a coordinated turn.

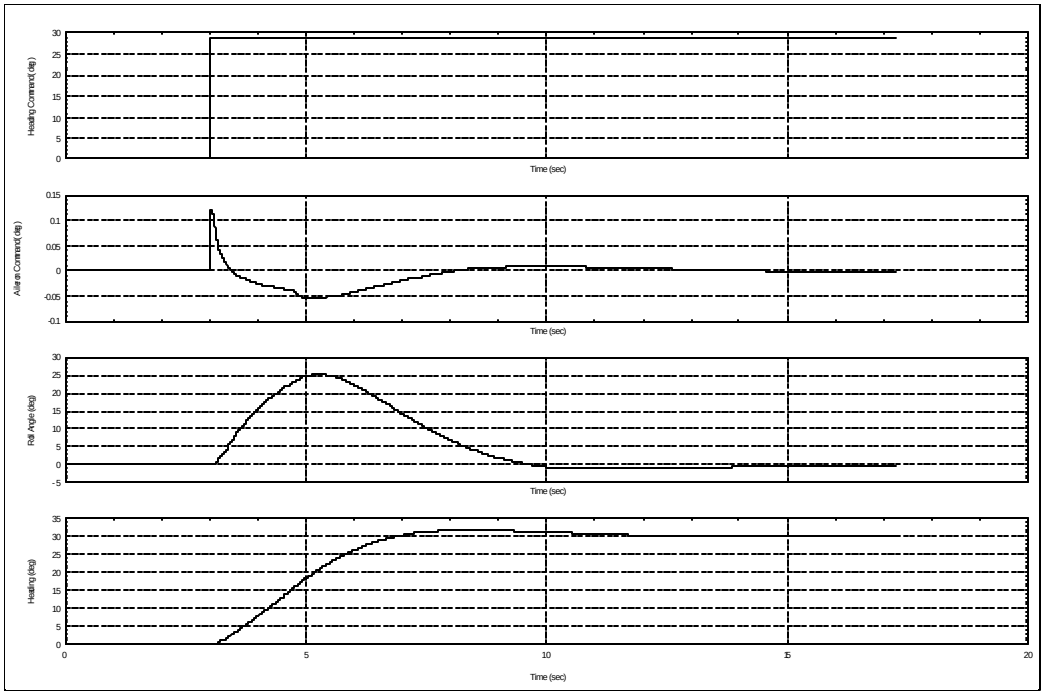

Figure 3.17. Heading Controller Response With Simulated Actuators.

The HTIL response of the heading controller is presented in Figure 3.18. The HITL response exhibits a ψ overshoot of approx. 8% and a rise time of 4.8 seconds, which compares favorably with the simulation results. It was noted that the aileron servo output exhibited nearly the same periodic variation as was observed in the yaw damper. It is possible that the source of this signal would only be found in one of these channels and that it is propagating into the other channel via the airplane's roll/yaw coupling. Further analysis of this phenomenon is warranted.

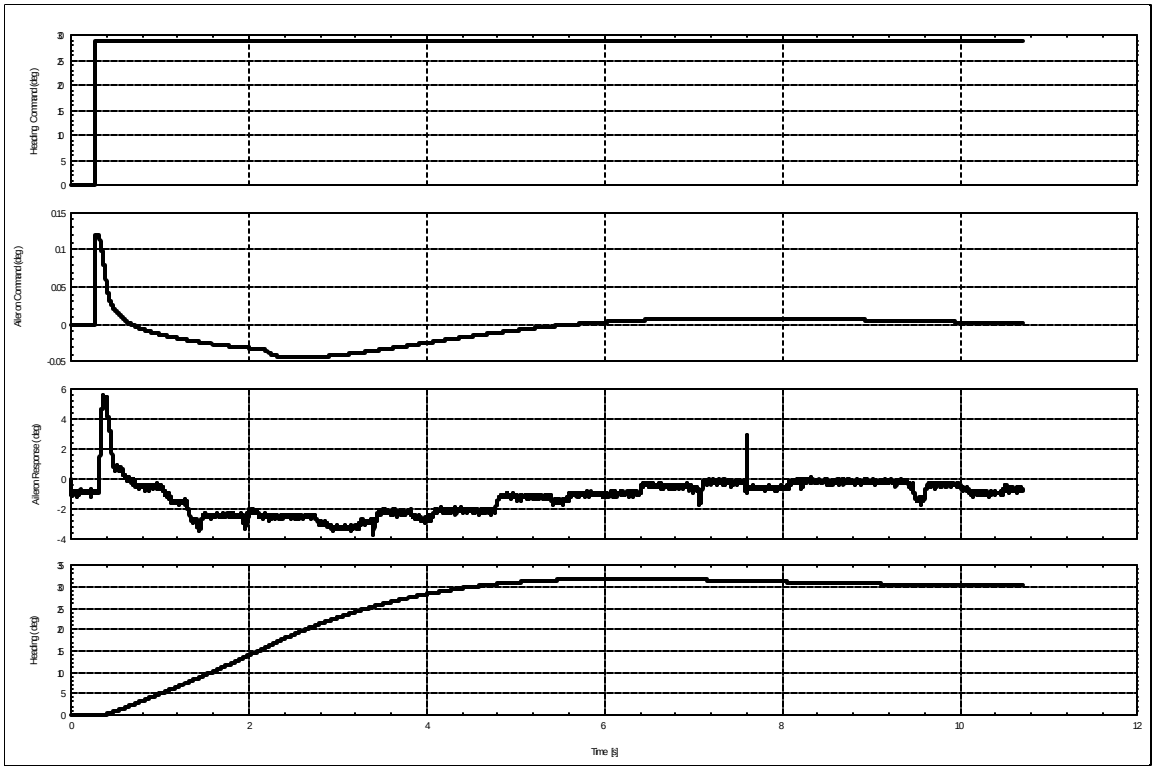

Figure 3.18. Heading Controller Response with TS-75 Servos.

5. Airspeed Controller

A PI speed controller was implemented on the throttle. The speed controller block diagram is depicted in Figure 3.19. The speed controller's gain and phase margin were 36.7 dB @ 38.75 rad/sec and 88.1° @ 0.33 rad/sec respectively.

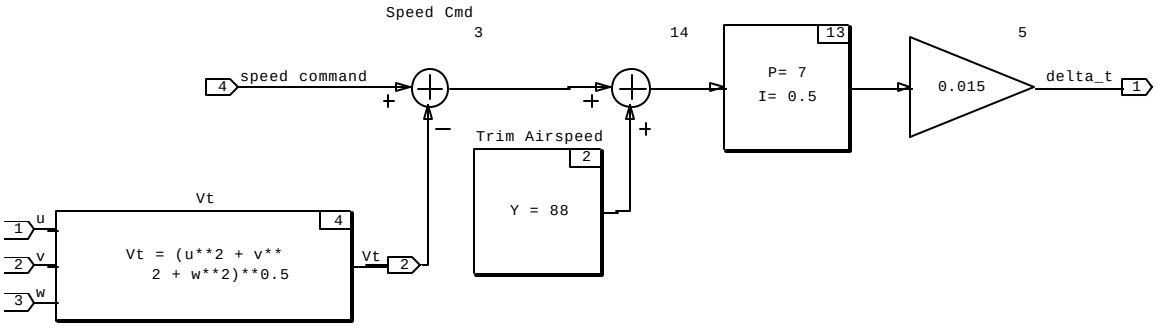

Figure 3.19. Speed Controller SystemBuild Block Diagram.

The response of the speed controller to a 10 fps STEP command was the same for the simulated and HITL actuators, Figure 3.20. The airspeed overshoots by approximately 5% and has a rise time of about 9 seconds. It is noted that a small amplitude periodic signal is imposed upon the throttle servo output signal. This variation has the same period as that observed on the elevator command channel and is likely due to the close coupling between throttle and elevator in the airplanes longitudinal modes.

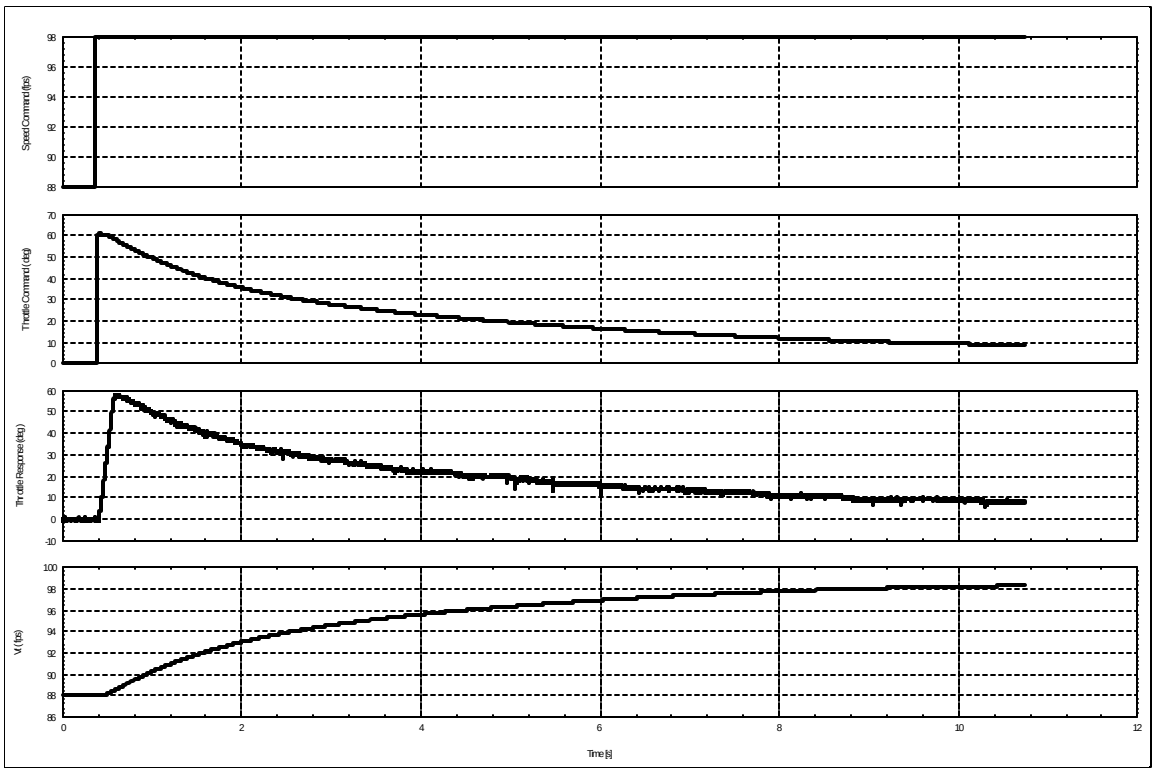

Figure 3.20. Speed Controller Response with TS-75 Servos.

6. Control Mode Coupling

The coupling of the longitudinal and lateral control modes was observed on both the simulated and HITL versions of the system. On the longitudinal side, the relationship between throttle (speed) changes and elevator (pitch attitude) changes is shown in Figure 3.21. In the figure, a +10 fps STEP command is sent to the speed controller at t = 1.0 sec. As the airplane begins to accelerate it generates more lift and also begins to climb. The altitude controller responds to this climb by programming a tail up elevator command to arrest the climb and return the airplane to its original altitude. It can also be seen that the airplane establishes a new trim attitude with $\theta = -0.13°$. The lateral control coupling was observed by commanding a STEP heading change and observing the aileron and rudder response, Figure 3.22. As the heading controller commands the ailerons to deflect the rudder immediately deflects to provide turn coordination and the airplane smoothly turns to the commanded heading.

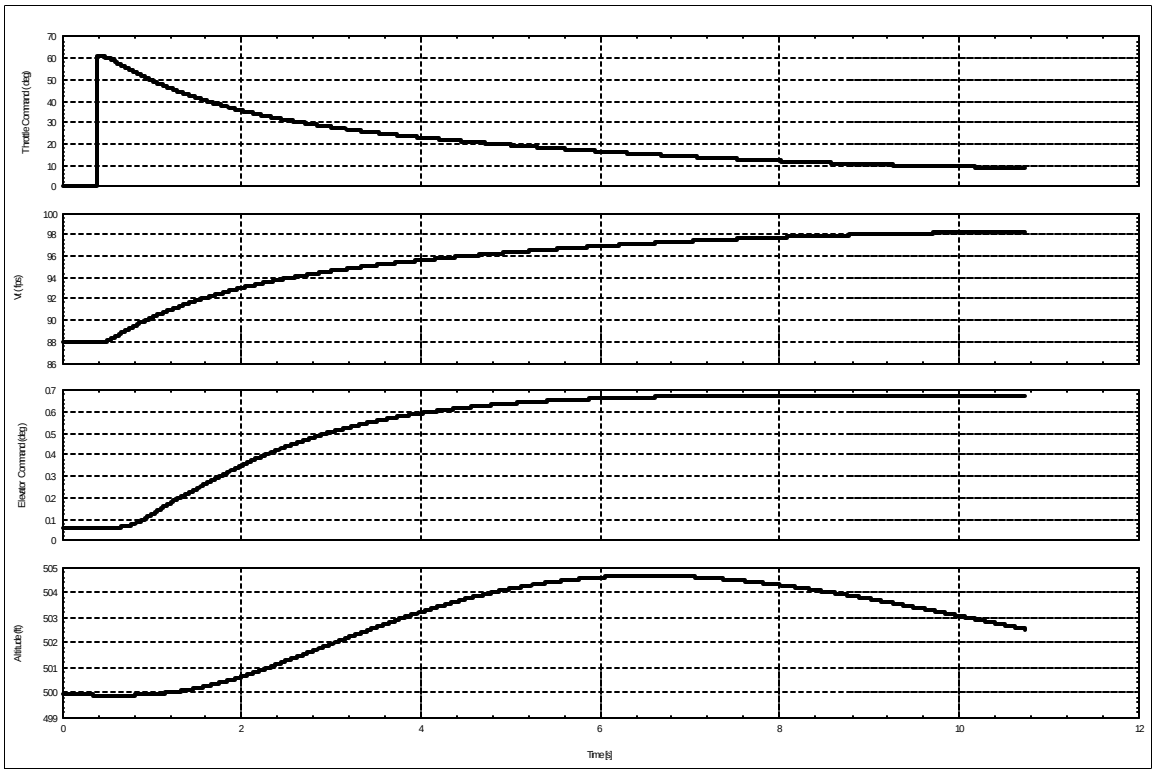

Figure 3.21. Longitudinal Control Mode Coupling.

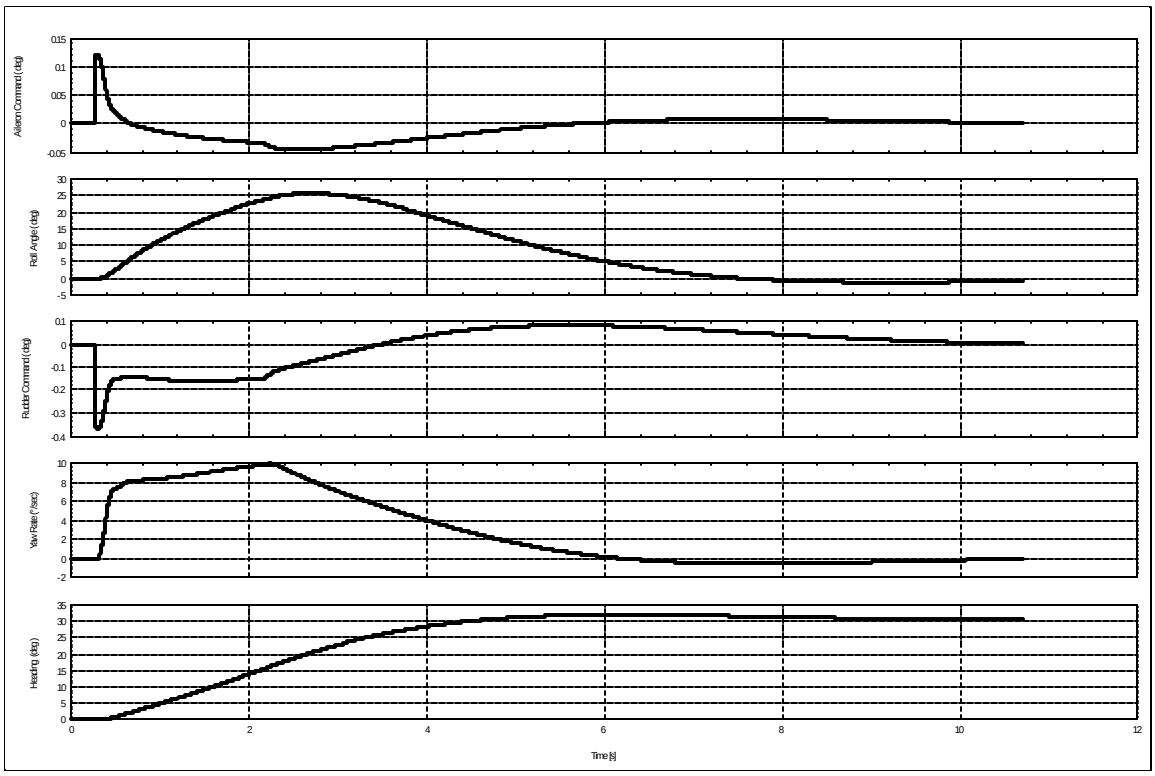

Figure 3.22. Lateral-Directional Control Mode Coupling.

C. FROG CONTROL SERVOS

The FROG control servos were not instrumented in time to support initial autopilot design so the TS-75 servos were used as a surrogate. Once the FROG was instrumented the RealSim servo test system was modified to permit dynamic response testing of the servos installed on the airplane.

1. FROG Servo Configuration and Instrumentation

The FROG was configured with a number of different servo models. The elevator servo specifications matched those for the test set TS-75 servos but the aileron, rudder and throttle servos are specified at approximately half the output torque and give a somewhat slower response. The FROG servo configuration is listed in Table 3.1

Control Channel	Futaba Model	Torque (oz-in)	Speed (sec / 60°)	Dimensions Width x Length x Height (in)
Elevator	S3302	110.0	0.19	1.14 x 2.32 x 1.97
Ailerons	FP-S130	55.60	0.24	0.75 x 1.50 x 1.31
Rudder	FP-S130	55.60	0.24	0.75 x 1.50 x 1.31
Throttle	FP-S130	55.60	0.24	0.75 x 1.50 x 1.31
Flaps[1]	FP-S125	129.30	0.62	0.88 x 1.75 x 1.69
Servo Test Set[2]	TS-75	110.0	0.19	1.14 x 2.32 x 1.97

Note 1: Flaps are not used by the autopilot.
Note 2: Provided for comparison.

Table 3.1. FROG Control Servo Configuration.

It was neither practical nor desirable to obtain servo position feedback in the fashion used for the Lab test set servos. Tapping into the servos internal feedback circuit would have required cutting a hole in the servo's case, which would have permitted moisture, dust and EMI intrusion. Each one of these are to be avoided in flight worthy hardware. It was also impractical to attach external instrumentation directly to the FROG servos so the control surfaces were instrumented instead. The FROG servos are attached to the movable control surface by means of a linkage rod. Rotation of the servo shaft was translated to a linear push/pull on the rod. The linkage rod was attached to a mounting horn on the control surface. The push/pull motion of the linkage rod is translated into rotation at the control surface. The elevator servo installation is depicted in Figure 3.23.

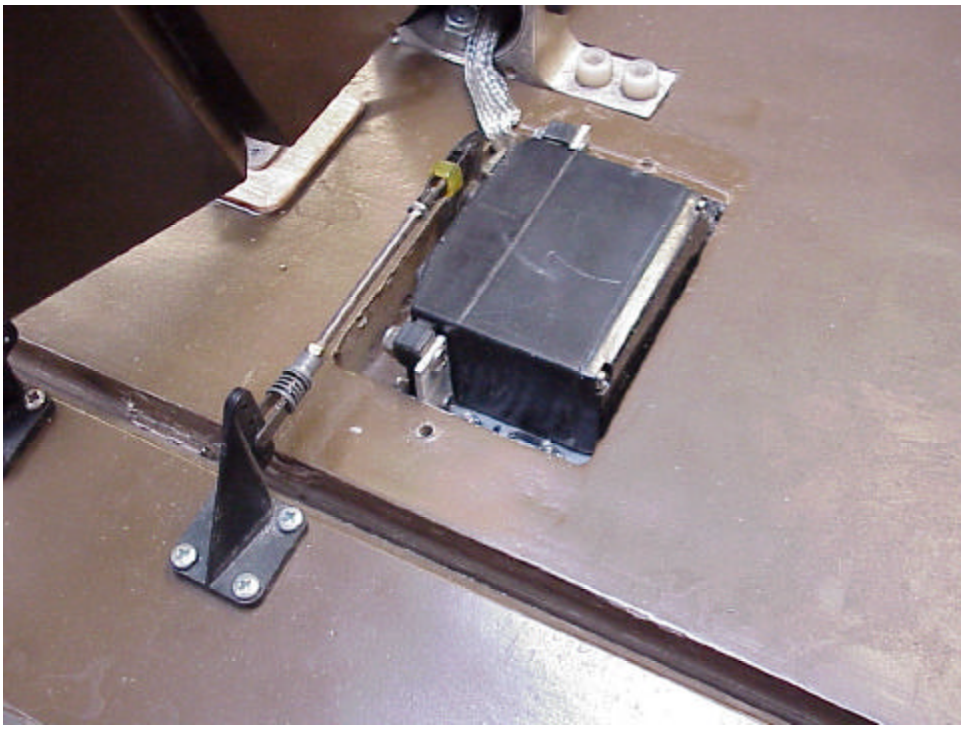

Figure 3.23. FROG Elevator Servo Configuration.

The servo position and control surface hinge configuration result in a nonlinear transfer function from servo rotation angle to control surface deflection angle. The control surfaces were instrumented with Series 150 subminiature linear position transducers, manufactured by SpaceAge Control, Inc. The position transducers, commonly known to as “string pots” consist of a small threaded drum attached to a single turn rotary potentiometer. The potentiometer shaft is connected to a coil spring which provides a positive retract force for all positions. The transducers cable is connected to a horn mount on the control surface so that the control surface neutral position roughly corresponds to mid-point in the transducers range, Figure 3.24. As the control surface moves the transducers cable is either extended or retracted. As the cable moves the string pots resistance increases or decreases.

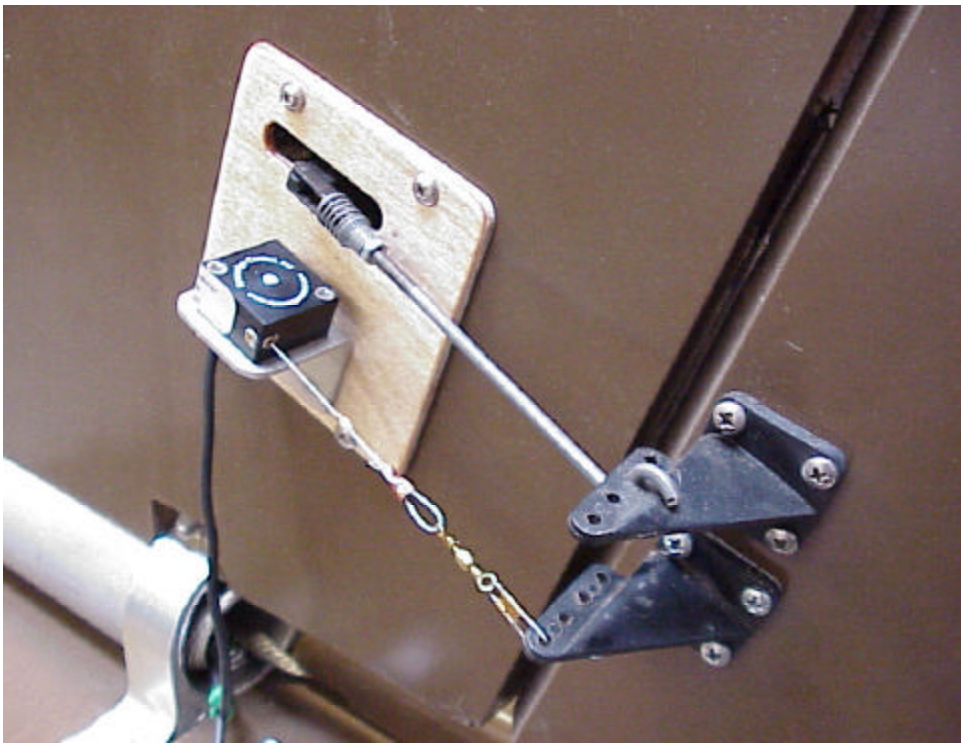

Figure 3.24. Series 150 Subminiature Position Transducer.

The string pots have a mechanical limit of 340° and are rated at 5K ohms, ±10%. Each string pot was provided a reference voltage of 3.73 V and the output voltage was routed to one of the TattleTale A/D converters in the NPS IMU computer. The raw string pot voltages were incorporated into the IMU serial output message and then sent (at 40 Hz) to the onboard FreeWave Modem; via the TattleTale's RS-232 port. The IMU message was received by the AC-104's FreeWave modem and then decoded.

2. Control Surface Position Calibration

Before servo dynamic response measurements could be made the position transducer outputs had to be calibrated. Additionally, the servo rotation limits had to be established so that the RealSim servo test system would not command positions that could result in binding or jammed controls. Both of these tasks required that the Futaba® command uplink be implemented within RealSim. The command uplink signal path is depicted in Figure 3.25.

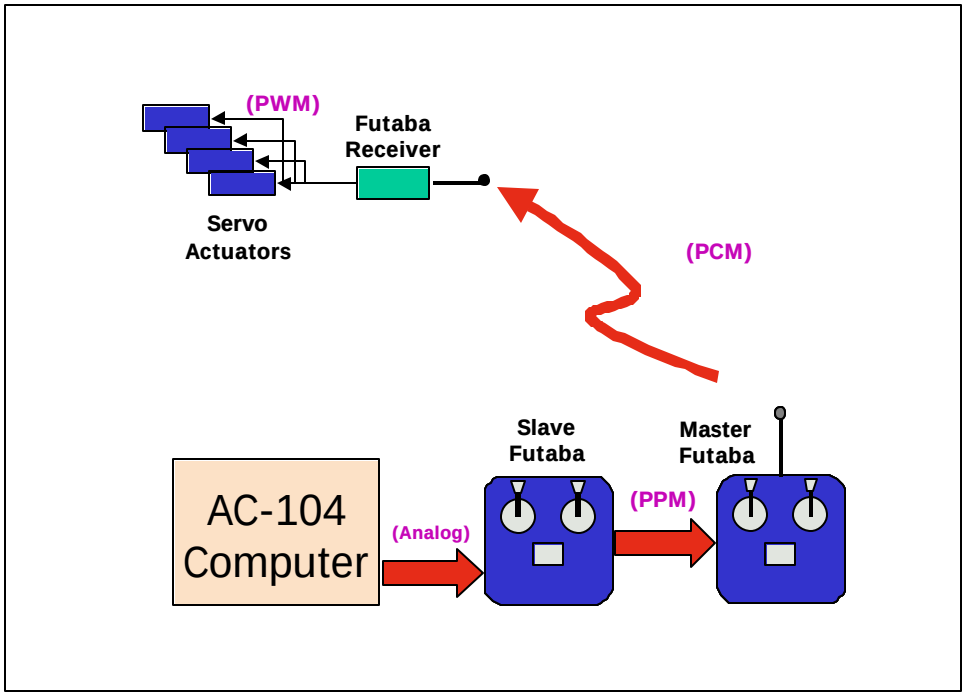

Figure 3.25. The FROG Command Uplink Signal Path.

The design approach for the FROG servo test system uplink was to start at the Futaba® FP-R309DPS receiver's PWM output signals and work backwards towards the AC-104 output. The first task was to determine the valid range of servo PWM commands. This would be the range of pulse widths generated by the Futaba® FP-R309DPS receiver in response to control sweeps from the Safety pilot's radio transmitter. The receiver's PWM output characteristics were determined by connecting the output channels of an FP-R309DPS receiver directly to the digital I/O lines of the AC-104's IP-68322 port. The IP-68322 was programmed to measure pulse width on the receiver output channels and the data was captured within RealSim. The elevator, aileron, throttle and rudder controls on the FP-9ZAP radio control were held in their extreme positions and the corresponding maximum and minimum PWM values were noted. These values were later confirmed using the DSO-2102 digital storage oscilloscope. Once the range of permissible PWM commands had been established the AC-104-to-Slave controller signals had to be calibrated. The Master radio controller was connected to the Futaba FP-8UAP Slave via the trainer cable. The FROG servo test system interfaced with the Slave through the Ruby-MM 12-bit D/A converter. The

D/A's 12-bit resolution provided 4096 (2^{12}) possible command values. With the D/A converter configured for 0 to 5V unipolar operation, the resulting command resolution was $1.22\,mV\ (5V/4096)$. The RealSim servo controller was configured to send discrete digital commands (values from 0 to 4096) to the D/A, which generated proportional voltages for the Slave radio controller. These voltages were converted to PPM and were relayed to the Master radio via the trainer cable. Once in the Master, the PPM signals were recoded as PCM and transmitted to the FP-R309DPS receiver connected to the IP-68322. The receiver decoded the PCM signals and the resulting PWM servo command signals were measured by the IP-68322 and recorded by the AC-104.

The test method was to start the controller with a default output command of 2048 (mid-range for the D/A) and then decrease or increase the command value until the PWM limits were observed. As the D/A digital command was decreased the pulse width decreased until a minimum of approximately 0.9 ms was reached at a D/A command value of 1515 (1.849 V_{out}). Further decreases in D/A command had no effect on the PWM pulse length. As the D/A command was increased an upper pulse width limit of approx. 2.1 ms was reached at a D/A command value of 2685 (3.28 V_{out}). When the D/A command was increased above 2685 the receivers pulse width output abruptly jumped down to 1.5 ms. This is the pulse width that roughly corresponds to the control neutral position. Based on these results limiter was added to the servo command path to keep the D/A commands between 1515 and 2675.

The next task was to determine two transfer functions: D/A command (0 – 4096)-to-control position (deg) and string pot V_{out}-to-Surface Position (deg). In order to determine the translation polynomials an accurate method of measuring each control surface's deflection was required. A laser pointer was attached to the control surface such that the optical axis of the laser was perpendicular to the axis of rotation. The laser's objective lens was equipped with an opaque mask that reduced the exit aperture to 0.014 mm. The laser's beam was aimed at the inside surface of a section of circular cylinder. The cylindrical surface had a radius of 20.0 inches and was marked in 0.2° increments. This permitted accurate angular measurements down to ±0.1°. As the

control surface was moved, the position of the laser dot moved along the graduated scale. The control surface deflection instrumentation is depicted in Figures 3.26 through 3.28.

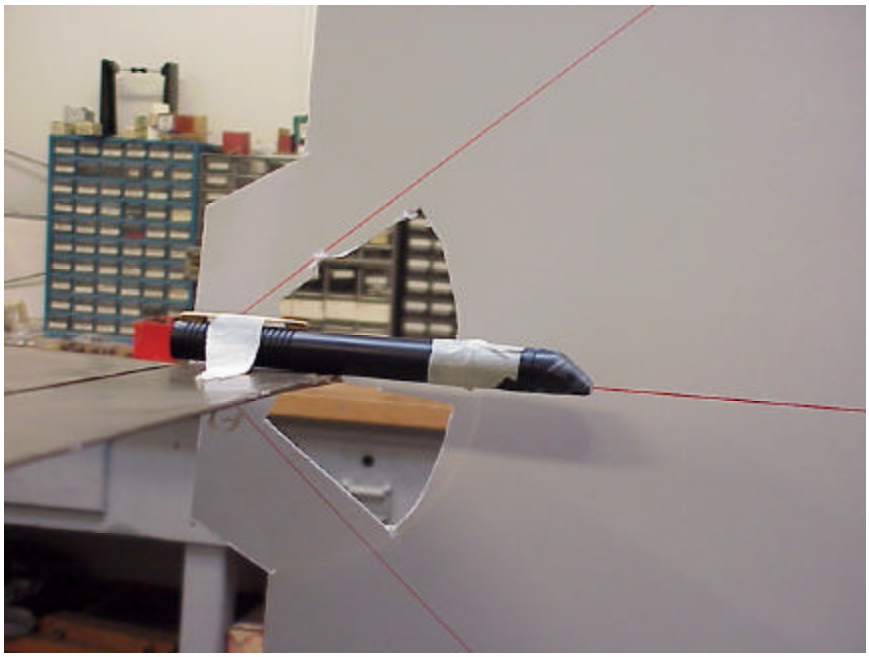

Figure 3.26. Laser Installation for Control Surface Deflection Measurement.

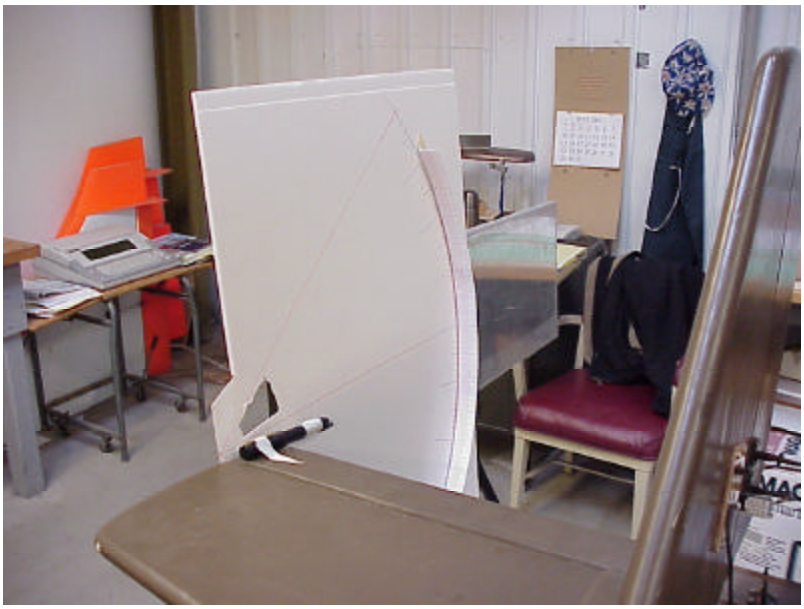

Figure 3.27. Cylindrical Laser Target.

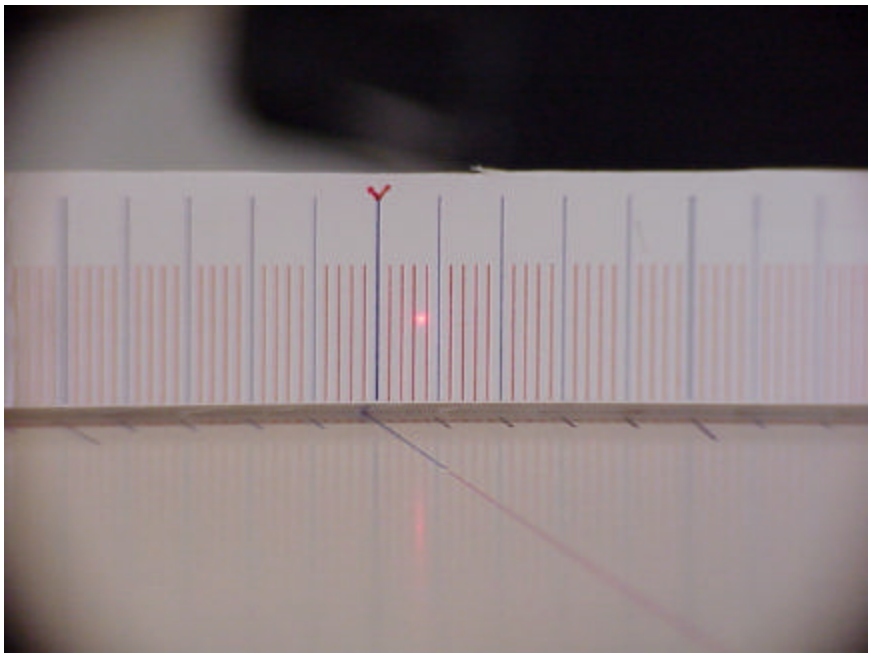

Figure 3.28 Laser Position Indicator Spot on Angle Scale.

The value of D/A command (0-4096), control surface deflection (deg) and string pot output were noted for each position. The data was then curve fit by either a 3rd or 5th order polynomial using a least squares approach. Figure 3.29 depicts the relationship between the FROG's aileron surface positions and AC-104 D/A output values. Figure 3.30 shows the relationship between the FROG's aileron surface positions and the string pot outputs.

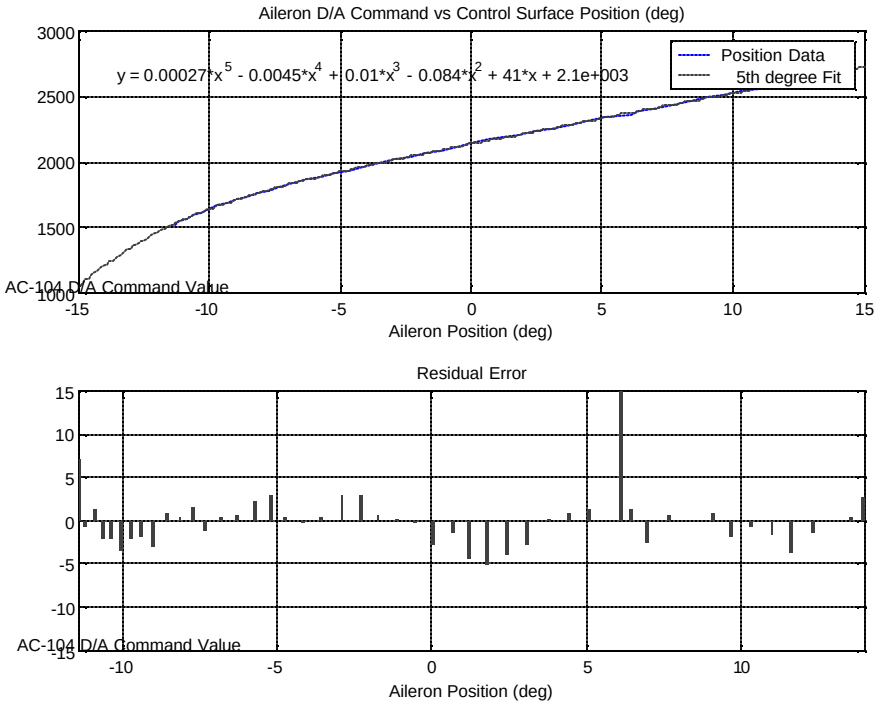

Figure 3.29. FROG Aileron Position Command versus D/A Output Value.

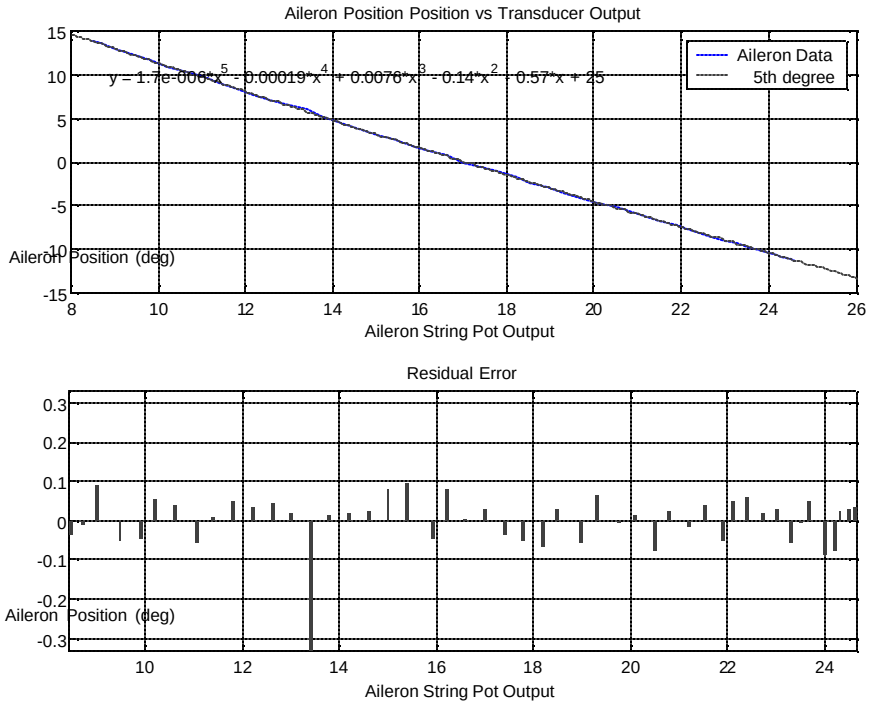

Figure 3.30. FROG Aileron String Pot Output versus Position.

3. FROG Servo Dynamics

Once calibrated the FROG control servo dynamics were assessed in the same manner used for the TS-75 test set servos. The FROG servos were given small and large STEP commands and the dynamic response was determined from the string pot position feedback. Analysis of the STEP input commands and the servo response revealed some unexpected system behavior. The strip chart in Figure 3.31 shows the elevator's response to a 20° STEP command (-10° to +10°). The elevator command and PWM channels were sampled at 1000 Hz. The elevator control surface string pot signal is only updated at 40 Hz (once every 25 ms); as it is part of the IMU downlink message. As can be seen in the figure, the position command from RealSim is nearly an ideal STEP with a rise time of 0.001 seconds. A closer inspection of the data revealed that the STEP commands were not sent simultaneously but were spaced 1 ms. apart, which matched the RTOS scheduler frequency. The first indication of a receiver response occurs some 69 ms later. The Futaba® FP-R309DPS receiver generated a piece-wise constant PWM command for the servos which took an additional 20 ms to reach full pulse width. The desired pulse width was not present in the receiver's output until 89 ms after the AC-104 command was issued. The first indication of control surface response occurred an average of 99 ms after the first pulse width change from the receiver. This was 168 ms after the initial command was issued by the AC-104. This significant command path delay had not been included in the FROG plant model when the flight controller was developed.

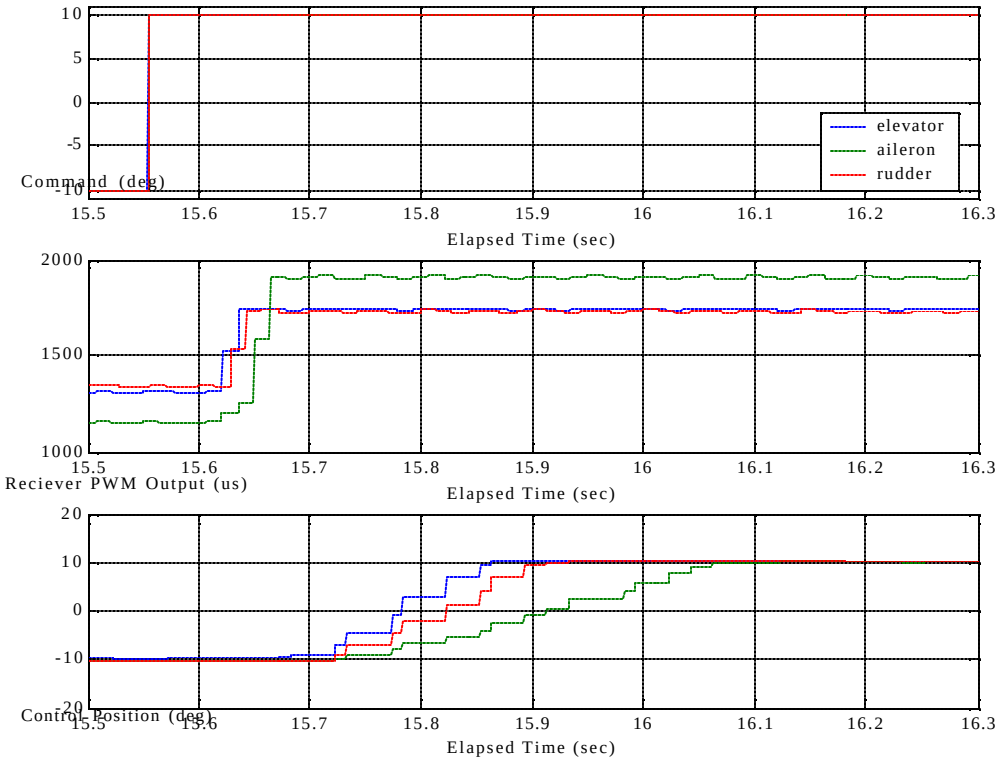

Figure 3.31. FROG Servo Response Delay.

Upon review of these results the servo test sets were tested to see if they showed similar command latency. In the Avionics laboratory test sets the AC-104 generates the PWM command signals with an internal Motorola 68322-based processor. As this is the same unit used to measure pulse width it was not possible to capture the time delay between servo command and pulse width response. It was possible, however, to measure the entire delay between command and servo response, which was found to be 33 ms, depicted in Figure 3.32. This RPFTS command path delay was over five times that found in the equipment used to develop the flight controller.

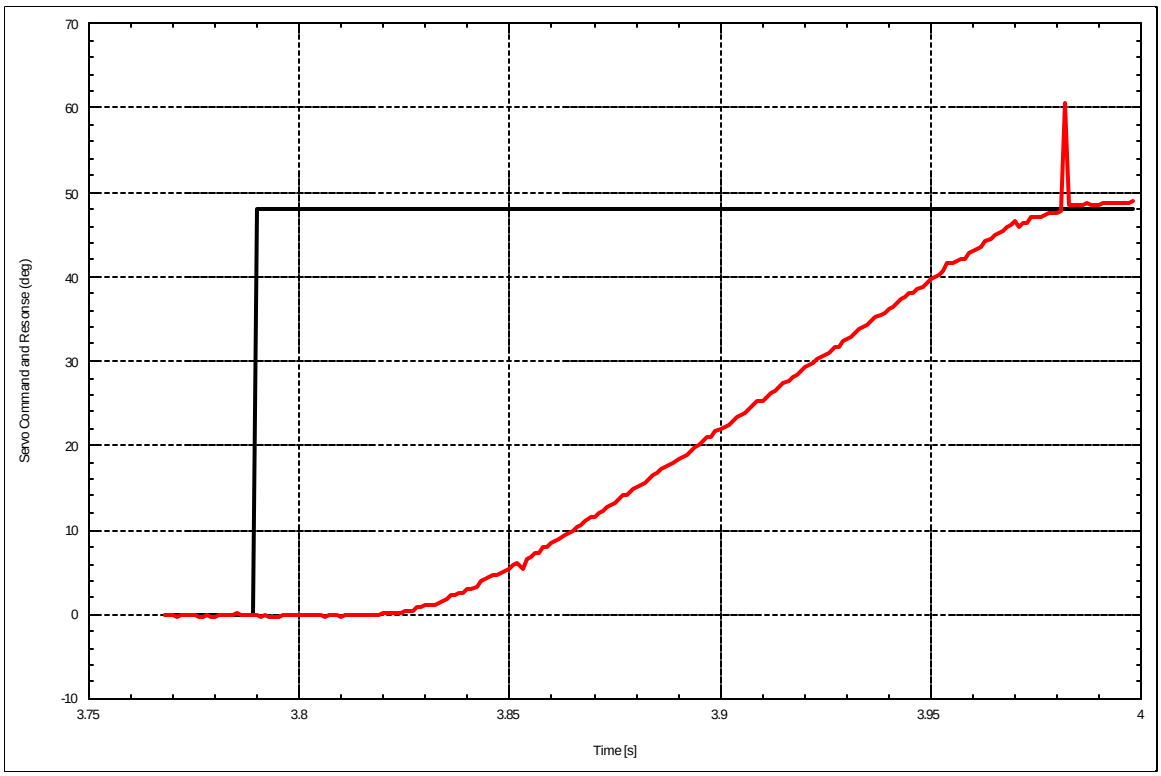

Figure 3.32. TS-75 Servo Test Set Response Delay.

The observed servo responses were used to create 2nd order servo models. The piece-wise PWM command signal gives each servo a dead-beat response and likely masks the true dynamics. As no appreciable over-shoot was present a damping ratio of 0.9 was assumed. The resulting forward loop and feed back gains are listed in Table 3.2.

Control Servo	Gain (K)	Feedback Gain (K_h)
Elevator	2612.8	0.0352
Aileron	444.72	0.852
Rudder	1423.9	0.0477

Table 3.2. FROG Servo Gains.

The servo speed controller used on the TS-75 servos was then incorporated into the FROG servo test system. Figure 3.33 shows the control surface response to variable

feedback gain. The aileron channel became unstable when the variable loop gain exceeded 6.5 (16.3 dB). The rudder and elevator became unstable for when the loop gain exceeded 8.0 (18.0 dB). It must be noted that the presence of considerable phase lag makes clouds this assessment of gain margin.

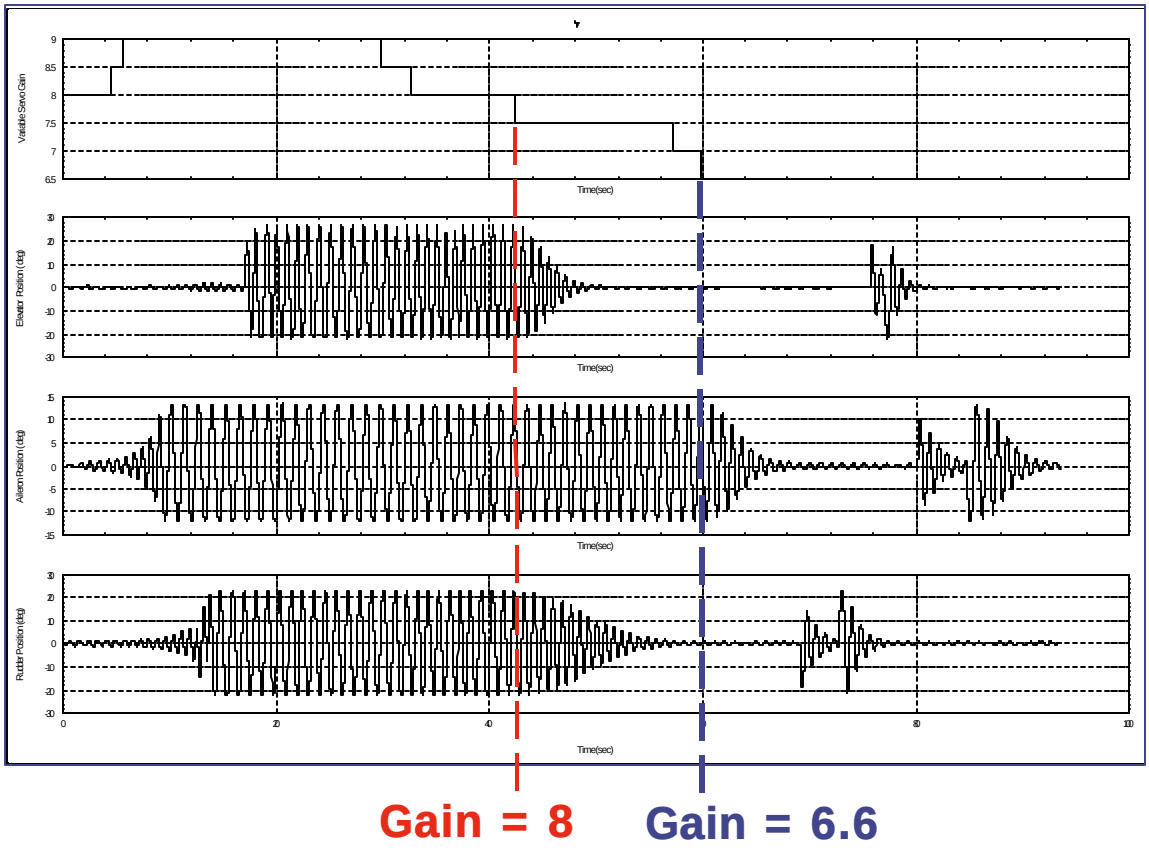

Figure 3.33. FROG Servo-based Speed Controller Performance.

4. FROG Digital Flight Controller Performance

The performance of the RFTPS FROG digital flight controller was tested with the FROG in the loop. The flight controller became unstable as soon as it was initialized and was unable to track in any channel. The autopilot was then evaluated in the lab to determine the influence of the command path latency. Figure 3.34 shows the effect of introducing a 170 ms delay into the flight controller's command output path. The aileron and rudder are 180° out of phase with the controller's commands while the elevator and throttle lag the controller by 110°. The command path delay was reduced to 50 ms and the controller's performance was restored.

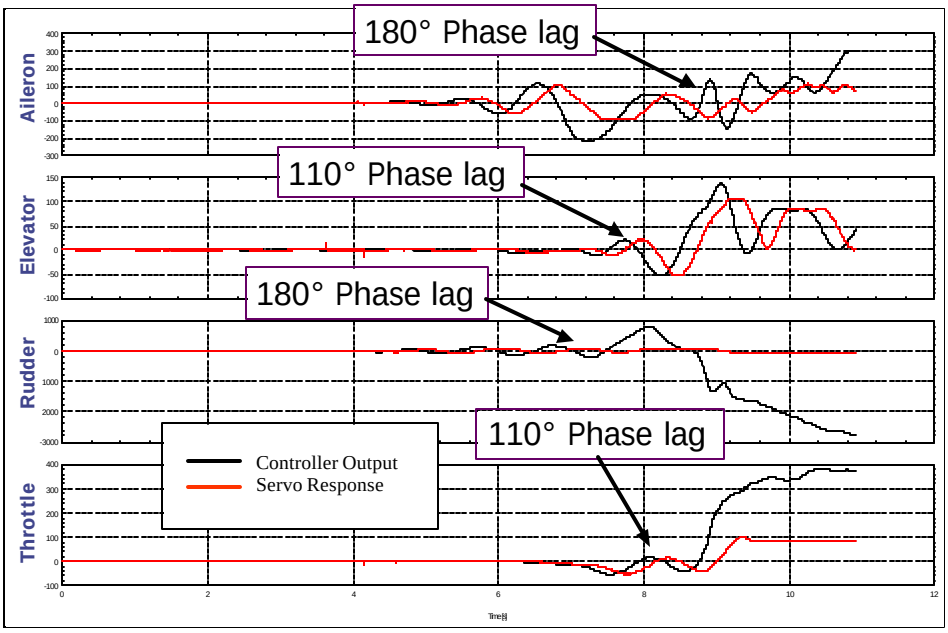

Figure 3.34. Flight Controller with 170 ms Delay.

D. ALTERNATE COMMAND UPLINK

The interest in an alternate uplink method was initially driven by the short range afforded by the Futaba® transmitter. Upon discovery of the magnitude and effects of the RPFTS latency on the digital flight controller the focus of this investigation shifted to reducing latency. The FreeWave modems used for data downlink provide a line of sight range of up to 20 miles and offered the potential for higher speed uplink. Serial commands can be sent from the AC-104, via the IP-serial board, to the FROG where they are decoded in the Tattletale computer. The Motorola 68322 micro-controller, used in the TattleTale and IP-68322, contains a powerful time processing unit (TPU) that can perform match and capture operations on time, freeing the CPU for other tasks. The TPU is in essence a slave processor built into the 68322 that controls two timers and sixteen I/O lines and is capable of generating PWM signals. Each Tattletale 8 provides access to nine of 68322's TPU lines. Two of the TPU lines are currently used as a data bus between the Master and Slave Tattletales. An investigation was conducted to assess the feasibility of using four of the remaining TPU lines to generate PWM command signals in place of the Futaba® receiver. In order to demonstrate the feasibility of this approach, the Futaba® receiver-to-servo PWM signal characteristics had to be determined. On board the FROG a Futaba® FP-R309DPS receiver converts the PCM

command signals to PWM and sends them on to the control servos. The PWM signal characteristics were measured with a DSO-2102 PC—based digital storage oscilloscope, manufactured by Link Instruments, Inc and with the IP-68322 data acquisition card in the AC-104. A Futaba FP-9ZAP digital proportional radio control was used to generate PCM command signals for the FP-R309DPS receiver. For this test, the controller trim settings were set to zero so that the systems neutral command values could be determined. It is noted that in actual practice the FROG controller trim settings are non-zero in order to achieve balanced flight. The receiver generates 2.84 Volt PWM command signals at an average frequency of 70.17 Hz (14.250 ms period). Figure 3.35 shows the receiver PWM output in response to a maximum elevator command and a minimum aileron command.

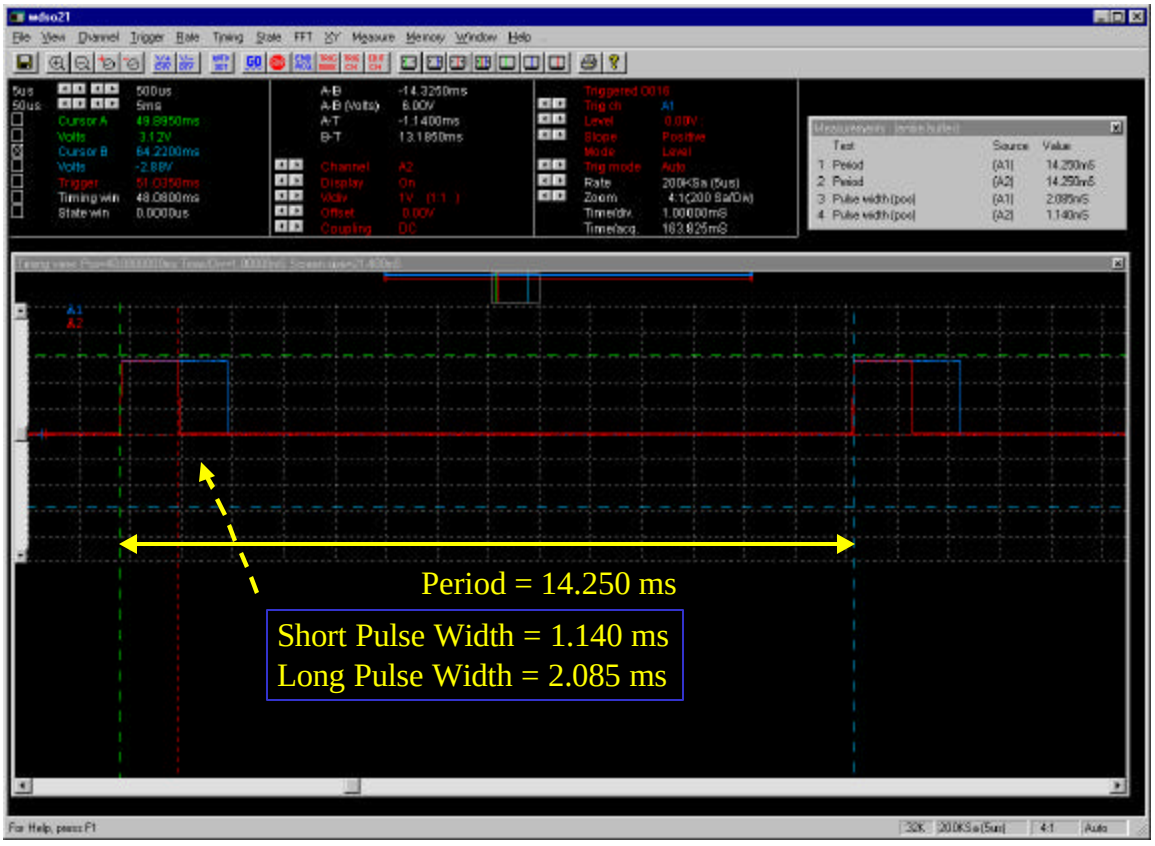

Figure 3.35. FP-R309DPS Receiver Output Signal.

The pulse width limits for each of the control channels were measured on the digital storage oscilloscope and are presented in Table 3.3.

Receiver Channel	Pulse Width (ms)		
	Minimum	Neutral Trim	Maximum
Elevator (ch 1)	0.955	1.520	2.085
Aileron (ch 2)	1.130	1.520	2.120
Throttle (ch 3)	1.235	1.570	1.940
Rudder (ch 4)	1.085	1.515	1.955

Table 3.3. FP-309DPS Pulse Width Response to FP-9ZAP Controller Commands.

The minimum pulse width observed was 0.955 ms for elevator channel. The maximum pulse width was 2.120 ms on the aileron channel. The observed neutral command pulse width matched the servo specified value of 1.52 ms for the elevator and aileron channels and nearly so for the rudder channel. The throttle neutral pulse width however was 1.57 ms. The throttles variation from the specified neutral output was probably due to a voltage (resistance) bias in the throttle control stick. The receiver PWM output, in response to dynamic commands was assessed with the AC-104's IP-68333 programmed to measure pulse period.

Once the baseline PWM signal parameters were determined the operating system software in the NPS IMU was modified to accept and decode a servo command uplink message. The servo command data was then used to set the pulse width duration. Once the Tattletale program was modified the FROG control system was modified to include a serial uplink message. The servo calibration controller was then used to generate serial command signals, which were transmitted via the FreeWave modems to the Tattletale computer. The first timing test was conducted with the PWM generation functions embedded in the existing 3DM TattleTale program. With the IMU data functions enabled the PWM output delay was a disappointing 150 ms. This would result in a servo delay of approximately 180 ms (counting the 33 ms delay observe in the lab). The TattleTale program was then modified and all non-PWM related functions were disabled. With the IMU sampling functions disabled the PWM output delay was reduced to 76 ms. To assess the suitability of this uplink method a 75 ms delay was

inserted into the flight controllers output path and the controller was run with TS-75 HITL, Figure 3.36. The reduction in servo response latency, while significant, was not enough to reduce the restore stability to the system.

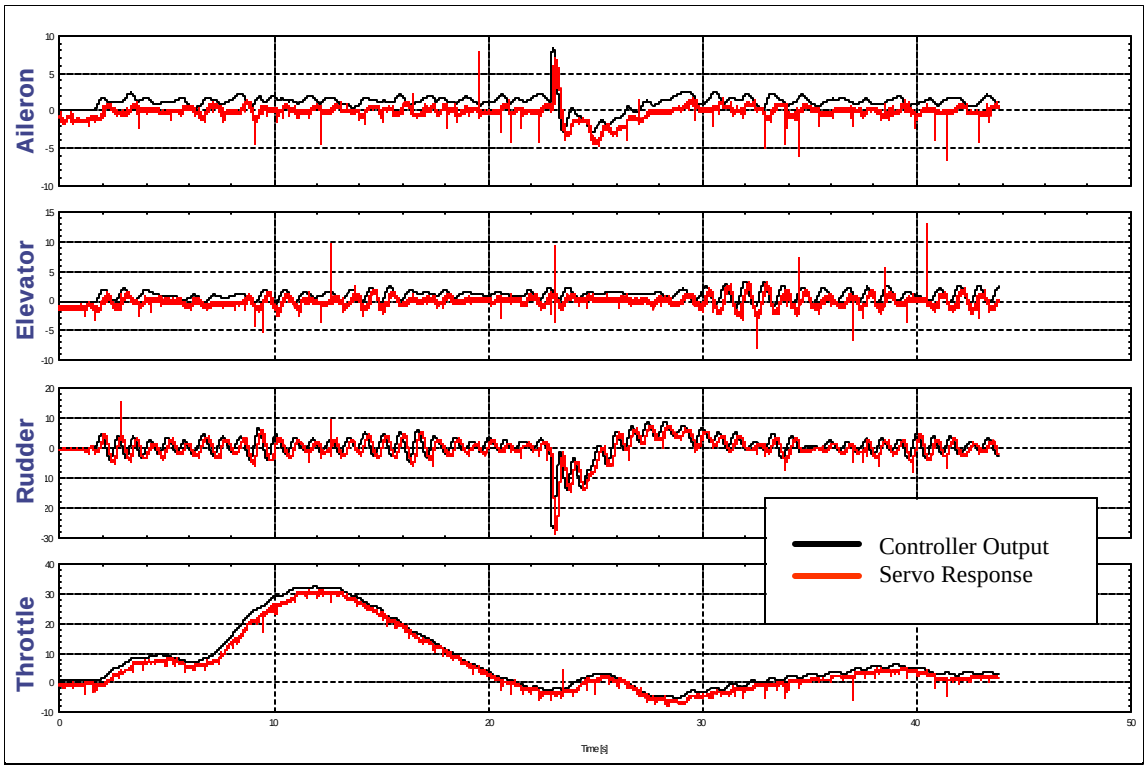

Figure 3.36. Flight Controller Performance with 75 ms Delay.

The 75ms delay produced limit cycle oscillation on each controller channel. Figure 3.37 shows the phase lag between each controller channel and it's associated servo. The altitude controller exhibited a phase lag of 62 degrees compared to a phase margin of 58.6 degrees. The rudder servo had 80 degrees of phase lag which is slightly more than the Yaw damper's phase margin of 76.1 degrees. The heading and speed controllers had phase margins of 80 and 88 degrees respectively. The servo responses to these controllers are less coherent suggesting that the phase margin has not been exceeded yet.

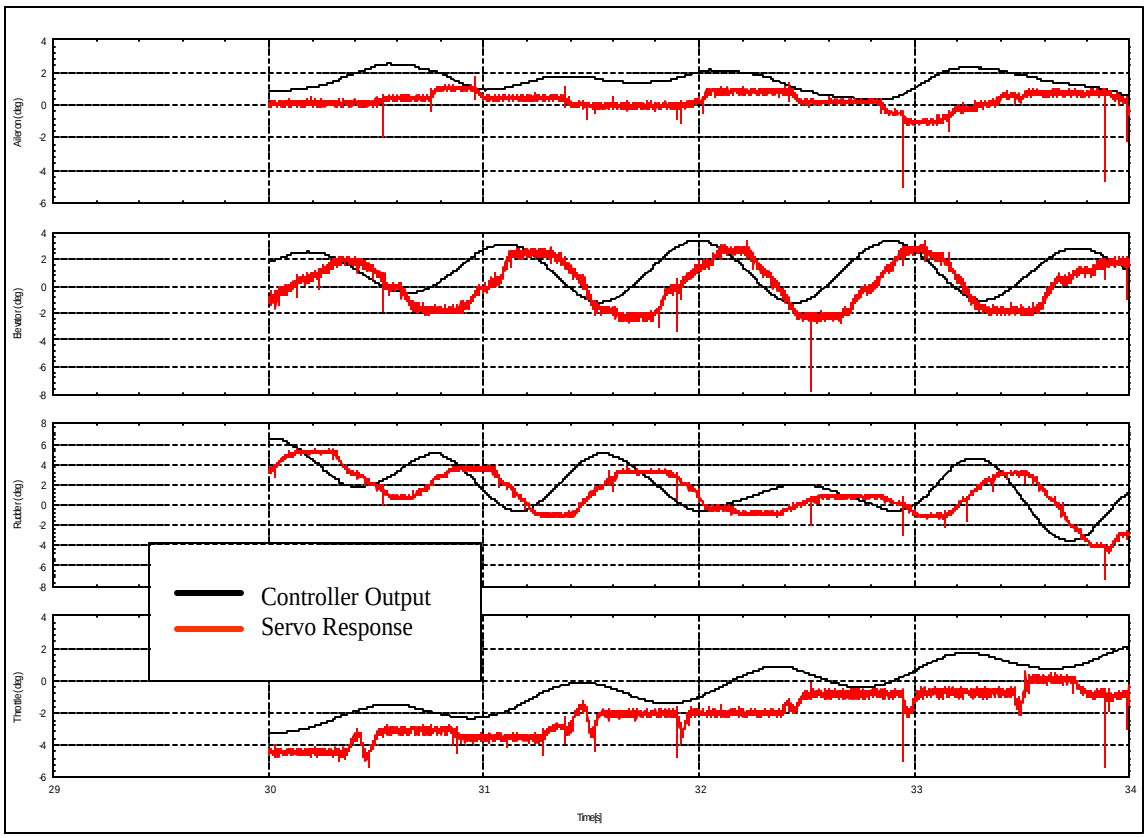

Figure 3.37. Servo Phase Lag with 75 ms Command Delay.

It must be noted that though the digital flight controller does not provide satisfactory performance with 76 ms delay it was not designed to do so. It is quite probable that simply reducing the controller gains would restore stability but this would come with a performance penalty. The prudent approach would be to incorporate the servo response latency into the FROG plant model and re-tune each of the flight controllers' components. This way it may be determined if the desired controller performance can be achieved with the current command path delay and if not a suitable design goal for command path latency can be determined. This test did demonstrate the validity of the serial uplink control scheme to reduce command uplink latency. Little attention was paid to optimization of the TattleTale code and it is recognized that there is room for improvement in this area. What is more important than the demonstration of the TattleTale's ability to replace the Futaba® transmitters and receiver is that principle that serial uplink offers the potential to reduce command latency as well as extend the FROG's usable range.

IV. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

The use of HITL greatly assisted the design and development of the digital flight controller. The laboratory HITL, however, did not provide the required fidelity to be able to create a flight controller for the FROG Rapid Prototyping Flight Test System. The unmodeled command path latency resulted in excessive servo phase lag, which exceeded the controllers phase margin. Reduction of the pulse width delay to 50 ms provided satisfactory performance but likely leaves little phase margin. Inertial sensor noise was not included in the FROG model and its effect on the flight controller's performance was not evaluated.

The Crossbow AHRS does not provide updates at a constant frequency in the angle/ continuous mode of operation.

B. RECOMMENDATIONS

The fidelity of the FROG plant model should be improved by inclusion of the command path latency, FROG servo dynamics and by sensor noise models. The flight controller should also be modified to include limiters that restrict the control surface deflection commands to the measured FROG control surface deflection limits, when running with the TS-75 servo test set. Once these factors have been incorporated the flight controller gains should be re-tuned in an attempt to meet the original performance criteria. If the flight controller performance goals cannot be met an acceptable command path delay should be determined and alternate command architecture should be developed. Particular attention should be paid to serial via the existing FreeWave modems. Consideration should also be given for transitioning to a wireless Ethernet system.

The IMU computer should be modified to provide airborne data processing and management capabilities for the AHRS avionics configuration. Future FROG guidance and control projects will need to make use of analog to digital conversion onboard the airplane. The TattleTale computer and operating software can be easily modified to perform this task.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. NPS IMU OPERATING SOFTWARE

This appendix contains the "Aztec C" code listings for the 3DM and GPS TattleTale microprocessors in the NPS IMU computer. Mathew Comerford originally wrote this code while a student at the Naval Postgraduate School. The code was subsequently modified to include the PWM generation capability by Maj Bock Aeng Lim and the author.

A. 3DM TATTLETALE OPERATING CODE

```
/*--------------------------------------------------------------------------------*/
/*  3DM Tattletale Operating System                                               */
/*  Primary function: IMU sampling, 3DM sampling, & Timing                        */
/*  Secondary function: Merge GPS data into data stream                           */
/*  April 23, 2001
/*  Modified: September 2001                                                      */
/*  Filename:  3DMTTwPWM.c                                                        */
/*--------------------------------------------------------------------------------*/

#include <TT8.h>          /* Tattletale Model 8 Definitions */
#include <tt8lib.h>             /* definitions and prototypes for Model 8 library */

#include <tt8lib.h>        /* for TT8 functions */
#include <tat332.h>        /* 68332 Hardware Definitions */
#include <tpu332.h>        /* 68332 Time Processing Unit Definitions */
#include <sim332.h>             /* 68332 System Integration Module Definitions */

#include <stdlib.h>        /* for malloc */
#include <stdio.h>         /* for printf() */
#include <userio.h>        /* for prompts */
#include <string.h>        /* for sting comparison */
#include "AtoDdata.h"      // for AtoDdata sturcture definition
#include "data3DM.h"       // for data3DM structure definition
#include "3DMio.h"         // for SendAtoD and Send3DM functions
#include "PWM.h"           // for PWM functions definition



// #define TESTMODE            //  if define test mode then program will sent simulated
data

#define SAMPLERATE  40                  //  Main sample frequency 40 Hz
#define SAMPLES     10000               //  Number of samples to log
#define QSIZE       12                  //  Must be a power of 2 {2,4,8,16,32...}
#define TSBUFSIZE   4096                //  Must be 2^QSIZE
// #define UplinkSerBUFSize    2048        //++  Buffer for Uplink PWM command at TPULine



/* Functions Prototype */

int SendAtoD (struct AtoDdata*);            // SendAtoD function prototype
int Send3DM  (struct data3DM*);             // Send3DM function prototype
int Read3DM  (struct data3DM* threeDM);     // Read3DM function prototype
int Send3DMTest (short value);              // Send3dmTest function prototype
int SendAtoDTest (struct AtoDdata *, short);// SendAtoDTest function prototype
int SendGPS (void);                         // SendGPS function prototype
int SendGPSAtoD(void);                      // SendGPSAtoD function prototype
void TPUSetupPWM(short pwmhi1, short pwmper1, short pwmhi2, short pwmper2, short pwmhi3,
short pwmper3, short pwmhi4, short pwmper4);
void TPUChangePWM(short pwmhi1, short pwmper1, short pwmhi2, short pwmper2, short pwmhi3,
short pwmper3, short pwmhi4, short pwmper4);
```

```
int  PWMcmds_from_uplink(short []);
// int  PWMcmds_from_TPU3(short []);

/* Global variable for PWM commands decoding */
int         first_frame_a = 0;
short   last_servo_cmd[8];

 main () {

 ulong sample = 0;                 // The current sample number
 ulong baud;                       // Baud rate
 ulong time = 0;                   // Timing Statistics
 ulong runningTime = 0;            // Timing Statistics
 ulong missed = 0;                 // Timing Statistics
 char timeStr[40];                 // Timing Statistics
 short status;                     // Timing Statistics
 short onTime = 1;                 // Timing Statistics
 short count = 0;                  // Timing Statistics
 float missedPercent = 0;          // Timing Statistics
 XmdmErr        xerr;              // Xmodem error code
 short value = 48;                 // ASCII value for '0' --> to be used in testing
 short i = 0;                      // Generic Counter
 long *valuePtr;                   // Pointer to stored values
 void *serBufPtr = NULL;           // Pointer to serial buffer
 void *inBufPtr = NULL;            // Pointer to serial buffer TPU(14)
 void *inBufPtr1 = NULL;           // Pointer to serial buffer TPU(1)
 void *inBufPtr2 = NULL;           // Pointer to serial buffer TPU(2)
 void *inBufPtr3 = NULL;           // Pointer to serial buffer TPU(3)
 void *outBufPtr = NULL;           // Pointer to serial buffer TPU(13)
 void *UplinkBufPtr = NULL;        //++
 struct AtoDdata AtoD3DM = {0};    // AtoD data structure variable
 struct data3DM threeDM   = {48059,48059,48059,48059,48059,48059};     // Hex BB BB
 struct data3DM threeDM2  = {221,221,221,221,221,221};                // Hex 00 DD
 short  servo_cmd[8];
 short tpumcr = 0x2040;            // TPU MCR register for PWM output



 InitTT8(NO_WATCHDOG,tpumcr);      // Initialize the Model 8

#ifdef TESTMODE
 printf("****************   Attention!!   Data   will   be   produced   in   TEST   mode
************\n");
#endif

 printf("\nSet BAUD rate to 38400\n");
 Sleep(0);
 Sleep (200000);               // Allow 5 second to display message
 // SimSetFSys(16000000);      // Set TT to 16.0 MHz --> fastest speed
 SimSetFSys(14720000);         // Set TT to 14.7 MHz --> best rate for serial transfer
 // baud = SerSetBaud(38400,0);    // Set baud rate
 baud = SerSetBaud(9600,0);
 SetTickRate(40000);           // TPU clock rate to 40 KHz --> 40 ticks = 1 ms

 //  ---------------    Set up Memory Buffers for Serial Input and Output ---------------
---
 serBufPtr = malloc(4096);                          //  Console (Primary) Serial Buffer
    if (serBufPtr == NULL) printf("\nBuffer Memory Allocation Failed\n");
    if (serBufPtr != NULL) printf("\nbufferPtr Memory O.K.\n");

 inBufPtr = malloc(TSBUFSIZE+TSER_MIN_MEM);          //  TPU(14) Serial In Buffer
    if (inBufPtr == NULL) printf("inBufPtr too big\n");
    if (inBufPtr != NULL) printf("inBufPtr Memory O.K.\n");

 inBufPtr1 = malloc(TSBUFSIZE+TSER_MIN_MEM);          //  TPU(1) Serial In Buffer
    if (inBufPtr1 == NULL) printf("inBufPtr1 too big\n");
    if (inBufPtr1 != NULL) printf("inBufPtr1 Memory O.K.\n");

 inBufPtr2 = malloc(TSBUFSIZE+TSER_MIN_MEM);          //  TPU(2//8) Serial In Buffer
    if (inBufPtr2 == NULL) printf("inBufPtr2 too big\n");
```

```
   if (inBufPtr2 != NULL) printf("inBufPtr2 Memory O.K.\n");

 inBufPtr3 = malloc(TSBUFSIZE+TSER_MIN_MEM);           //  TPU(3) Serial In Buffer
   if (inBufPtr3 == NULL) printf("inBufPtr3 too big\n");
   if (inBufPtr3 != NULL) printf("inBufPtr3 Memory O.K.\n");

 outBufPtr = malloc(TSBUFSIZE+TSER_MIN_MEM);          //  TPU(13) Serial Output Buffer
   if (outBufPtr == NULL) printf("outBufPtr too big\n");
   if (outBufPtr != NULL) printf("outBufPtr Memory O.K.\n");



 //  ---------------    Set up Serial Input and Output Parameters ----------------------
---
 // Open Console (Primary) serial port for buffered input & output
 SerSetInBuf (serBufPtr,4096);
 SerInFlush();

 // Open TPU(1) for buffered input
 if(TSerOpen(1,HighPrior,0,inBufPtr1,TSBUFSIZE,33600,'N',8,1) == tsOK) {
    printf ("TSerOpen(1) O.K.\n");
    TSerInFlush(1);
 };

 // Open TPU(2//8) for buffered input// it suppose to be 8 with the same mode 0
 if(TSerOpen(8,HighPrior,0,inBufPtr2,TSBUFSIZE,33600,'N',8,1) == tsOK) {
    printf ("TSerOpen(8) O.K.\n");
    TSerInFlush(8);
 };

  // Open TPU(3) for buffered input
 if(TSerOpen(3,HighPrior,0,inBufPtr3,TSBUFSIZE,9600,'N',8,1) == tsOK) {
    printf ("TSerOpen(3) O.K.\n");
    TSerInFlush(3);
 };

 // Open TPU(13) for buffered output
 if(TSerOpen(13,MiddlePrior,1,outBufPtr,TSBUFSIZE,9600,'N',8,1) == tsOK) {
    printf ("TSerOpen(13) O.K.\n");
};

 // Open TPU(14) for buffered input
 if(TSerOpen(14,HighPrior,0,inBufPtr,TSBUFSIZE,9600,'N',8,1) == tsOK) {
    printf ("TSerOpen(14) O.K.\n");
    TSerInFlush(14);
};



 TPUSetupPWM(500,1000,500,1000,500,1000,500,1000);  // simple values to open PWM channels

 printf ("Tick Rate is %ld\n",GetTickRate());
 printf ("System Clock is %ld\n",SimGetFSys());
 printf ("Baud Rate is %ld\n",baud);
 printf ("Press Control-C on the Primary port to reset to TOM8\n");
 Sleep  (0);
 Sleep  (200000);                       // Allow 5 seconds messages to be displayed.
// if (SerByteAvail()){
//      if (SerGetByte() == 3) ResetToMon();      // Reset to TOM8 Monitor if CNTRL C
Pressed
// }
                                       // Allows reset of EEPROM program w/o hardware
reset

 //TPUSetPin (0,1);                     // Now we use direct connection to transmit data
from TPU0(GPS) to TPU8(3DM)
                                        //  Define TPU0 for digital output and sets to
high level
                                                                //         Enable
Humphrey Gyro Auto Erect feature
                                      // Should this be done entire flight ??????
```

```
//*****************************************************************************
//  This portion of code conducts all of the timing for the IMU gyros and the 3DM
//  accelerometers/magnetometers.  The IMU AtoD data is sampled at 40 Hz while the 3DM
//  is sampled at 20 Hz because of the instrument's limits.  The 3DM is connected on
//  the secondary serial port at 9,600 baud and the output is on the primary serial port
//  Binary GPS data is received from the GPS Tattletale via TPU pin 1.
//*****************************************************************************

  do {
//  ---------------------------------------------------------------------------
//    1st Minor Timing Cycle @ 40Hz  -->  {send AtoD, send 3DM, poll 3DM for data}
//  ---------------------------------------------------------------------------
                                      // Note: ** Indicates debugging code
   Sleep(0);                          // Initialize timer
   count = 0;                         // ** Initialize missed timing slot counter
   if (value >= 56) {value = 48;}     // ** Reset sample counter to ASCII value for '0'

   TSerPutByte (13,144);              // Poll(inquiry) 3dm for data output

   SendAtoD (&AtoD3DM);               // Read and Send 16 bytes of AtoD data
   Send3DM (&threeDM);                // Send 12 bytes of 3dM data
   SendGPS();                         // Send 31 bytes of GPS data if available
   sample += 1;                       // ** Increment sample counter for testing
   value += 1;                        // ** Increment value counter for test data

  printf("\nPWMcmds_from_uplink status = %d\n", PWMcmds_from_uplink(servo_cmd));
//  printf("\nPWMcmds_from_TPU3 status = %d\n", PWMcmds_from_TPU3(servo_cmd));

   //  TPUChangePWM(PWM_default_hi,  PWM_default_per,  2*PWM_default_hi,  PWM_default_per,
3*PWM_default_hi, PWM_default_per, 4*PWM_default_hi, PWM_default_per);
   TPUChangePWM(servo_cmd[0],servo_cmd[1],    servo_cmd[2],    servo_cmd[3],servo_cmd[4],
servo_cmd[5],servo_cmd[6], servo_cmd[7]);


   onTime = Sleep (1000);             // Sleep until 25 ms is over (40 = 1 ms)
   if (!onTime) { count += 1; }       // ** Update missed timing slot statistics


//  ---------------------------------------------------------------------------
//    2nd Minor Timing Cycle @ 40Hz  -->  {Send AtoD, send 3DM, send GPS data}
//  ---------------------------------------------------------------------------

   // StopWatchStart();               // ** Begin timing of Segment
   SendAtoD(&AtoD3DM);                // Read and Send 16 bytes of AtoD data
   Read3DM (&threeDM);                // Read and store 12 bytes of 3DM data
   Send3DM (&threeDM);                // Send 12 bytes of 3dM data
   SendGPSAtoD();                      // Send 18 bytes of GPS Tattletale AtoD data if
available
   // time = StopWatchTime();         // ** End timing of segment

   sample += 1;                       // ** Increment sample counter for testing
   value += 1;                        // ** Increment value counter for test data

   onTime = Sleep (1000);             // Sleep until 25 ms is over (40 = 1 ms)
   if (!onTime) { count += 1; }       // ** Update missed timing slot statistics


//  ---------------------------------------------------------------------------
//    3rd Minor Timing Cycle @ 40Hz  -->  {Send AtoD, send 3DM, poll 3DM for data}
//  ---------------------------------------------------------------------------

   TSerPutByte (13,144);              // Poll(inquiry) 3dm for data output


   SendAtoD (&AtoD3DM);               // Read and Send 16 bytes of AtoD data
#ifdef  TESTMODE
   Send3DMTest (103 - value);        // Simulate 12 bytes from 3dM (ASCII '9'-'0')
#else
   Send3DM (&threeDM);                // Send 12 bytes of 3dM data
#endif
```

```
   SendGPS();                          // Send 31 bytes of GPS data if available

   sample += 1;                        // ** Increment sample counter for testing
   value += 1;                         // ** Increment value counter for test data

   onTime = Sleep (1000);              // Sleep until 25 ms is over (40 = 1 ms)
   if (!onTime) { count += 1; }        // ** Update missed timing slot statistics


//  ---------------------------------------------------------------------------
//    4th Minor Timing Cycle @ 40Hz  -->  {Send AtoD data, send 3DM data}
//  ---------------------------------------------------------------------------

   SendAtoD (&AtoD3DM);                // Read and Send 16 bytes of AtoD data
   Read3DM (&threeDM);
#ifdef  TESTMODE
   Send3DMTest (103 - value);         // Simulate 12 bytes from 3dM (ASCII '9'-'0')
#else
   Send3DM (&threeDM);                 // Send 12 bytes of 3dM data
#endif
   SendGPSAtoD();                       // Send 18 bytes of GPS Tattletale AtoD data if
available

   sample += 1;                        // ** Increment sample counter for testing
   value += 1;                         // ** Increment value counter for test data
   missed += count;                    // ** Total missed timing slot count
   runningTime += time;                // ** Total simulation time

   onTime = Sleep (1000);              // Sleep until 25 ms is over (40 = 1 ms)
   if (!onTime) { count += 1; }        // ** Update missed timing slot statistics


//  ---------------------------------------------------------------------------

/*
// Output real time troubleshooting data
   sprintf(timeStr," T%lu B%d   ",status,TSerByteAvail(1));
    for (i = 0;i <10; i++){
        SerPutByte (((int)timeStr[i]));
    }
   SerPutByte(13);                 //  Carriage Return & Line Feed
   SerPutByte(10);
*/
} while(1);                        // Infinite loop


 //*****************************************************************************
 //  ---------------    Output Timing Statistics and Download Options -------------------
 //  Output timing statistics.  This portion of the code will never be reached during
 //  normal program execution.
 //*****************************************************************************

 missedPercent = (float)missed/(float)sample*100;
 printf("\n# Samples: %ld, # Misses: %ld,  Percent Misses: ,%5.3f, Ave Parse Time: %5.3f,
\n",
    sample, missed, missedPercent,(float)runningTime/(float)sample*4);
 SerPutByte (13);                        // Carriage Return & Line Feed
 SerPutByte (10);

/*
if (QueryYesNo("\nOffload the data?", TRUE)){
   printf("\nStarting XMODEM transfer ... ");
   fflush(stdout);
   xerr = XmodemSendMem(valuePtr, SAMPLES * sizeof(long), 30);
   printf("  Complete [%d]\n", xerr);
}
*/

 SerInFlush();
 SerSetBaud(9600,0);
 ResetToMon();
```

```
 return (0);
}


/*****************************************************************************
**      TPUSetupPWM                   Setup TPU channel for Pulse-Width Modulation
**
**      Notes:
**              pwmper = period in tcr1 cycles [call TPUGetTCR1() for current value]
**              pwmhi = time high in tcr1 cycles
**              priority = LowPrior, MiddlePrior, or HighPrior [defined in tpu.h]
**              read about value limitations in the TPU Reference Manual
*****************************************************************************/


void TPUSetupPWM(short pwmhi1, short pwmper1, short pwmhi2, short pwmper2, short pwmhi3,
short pwmper3, short pwmhi4, short pwmper4)
        {

    /* declarations */
    ulong   tcr1;


        TPUSetPin(PWMChan1, 1);             /*  Configure Pins into Prpoer I/O State  */
        TPUSetPin(PWMChan2, 1);
    TPUSetPin(PWMChan3, 1);
    TPUSetPin(PWMChan4, 1);

    tcr1 = TPUGetTCR1();               /*  Get Current Clock Frequencey */

        *CIER &= ~(1 << PWMChan1);                          /*   don't   want   interrupts
enabled for Chan# */
    *CIER &= ~(1 << PWMChan2);
    *CIER &= ~(1 << PWMChan3);
    *CIER &= ~(1 << PWMChan4);


        FUNSEL(PWMChan1, PWM);                              /* configure PWMChan# for PWM
mode */
    FUNSEL(PWMChan2, PWM);
    FUNSEL(PWMChan3, PWM);
    FUNSEL(PWMChan4, PWM);

                          /* configure first 4 bytes of PWMRAM for PWMChan#  */
        PRAM[PWMChan1][0] = OutputChan | NoChangePAC | (pwmhi1 ? ForceHigh : ForceLow);
        PRAM[PWMChan2][0] = OutputChan | NoChangePAC | (pwmhi2 ? ForceHigh : ForceLow);
        PRAM[PWMChan3][0] = OutputChan | NoChangePAC | (pwmhi3 ? ForceHigh : ForceLow);
        PRAM[PWMChan4][0] = OutputChan | NoChangePAC | (pwmhi4 ? ForceHigh : ForceLow);


    /*  Convert Period & Time Hi input from usec to tcr1 ticks */

    pwmper1 = (float) tcr1 * (float) pwmper1 / 1e6;
        pwmhi1  = (float) tcr1 * (float) pwmhi1 / 1e6;
    pwmper2 = (float) tcr1 * (float) pwmper2 / 1e6;
        pwmhi2  = (float) tcr1 * (float) pwmhi2 / 1e6;
    pwmper3 = (float) tcr1 * (float) pwmper3 / 1e6;
        pwmhi3  = (float) tcr1 * (float) pwmhi3 / 1e6;
    pwmper4 = (float) tcr1 * (float) pwmper4 / 1e6;
        pwmhi4  = (float) tcr1 * (float) pwmhi4 / 1e6;


    /*  Write pulse width and period to PWM RAM */

    PRAM[PWMChan1][2] = pwmhi1;                             /* time hi for PWMChan#  */
        PRAM[PWMChan1][3] = pwmper1;                        /* period for PWMChan#   */
    PRAM[PWMChan2][2] = pwmhi2;
    PRAM[PWMChan2][3] = pwmper2;
    PRAM[PWMChan3][2] = pwmhi3;
    PRAM[PWMChan3][3] = pwmper3;
    PRAM[PWMChan4][2] = pwmhi4;
```

```
    PRAM[PWMChan4][3] = pwmper4;


    HOSTSERVREQ(PWMChan1, 2);                       /* initiate PWMChan1 */
        HOSTSERVREQ(PWMChan2, 2);
    HOSTSERVREQ(PWMChan3, 2);
        HOSTSERVREQ(PWMChan4, 2);


    while (HOSTSERVSTAT(PWMChan1) & 3);             /* await reply */
        while (HOSTSERVSTAT(PWMChan2) & 3);         /* await reply */
    while (HOSTSERVSTAT(PWMChan3) & 3);             /* await reply */
    while (HOSTSERVSTAT(PWMChan4) & 3);             /* await reply */



        }       /* TPUSetupPWM() */




/*************************************************************************
**      TPUChangePWM          Change PWM values for initiated TPU channel
**
**      Notes:
**              pwmper = period in tcr1 cycles
**              pwmhi = time high in tcr1 cycles
*************************************************************************/


void TPUChangePWM(short pwmhi1, short pwmper1, short pwmhi2, short pwmper2, short pwmhi3,
short pwmper3, short pwmhi4, short pwmper4)
        {

    /* declarations */
    ulong   tcr1;


    /*  Get Current Clock Frequencey */
    tcr1 = TPUGetTCR1();

    /*  Convert Period & Time Hi input from usec to tcr1 ticks */

    pwmper1 = (float) tcr1 * (float) pwmper1 / 1e6;
        pwmhi1  = (float) tcr1 * (float) pwmhi1 / 1e6;
    pwmper2 = (float) tcr1 * (float) pwmper2 / 1e6;
        pwmhi2  = (float) tcr1 * (float) pwmhi2 / 1e6;
    pwmper3 = (float) tcr1 * (float) pwmper3 / 1e6;
        pwmhi3  = (float) tcr1 * (float) pwmhi3 / 1e6;
    pwmper4 = (float) tcr1 * (float) pwmper4 / 1e6;
        pwmhi4  = (float) tcr1 * (float) pwmhi4 / 1e6;


    /*  NEED TO DO THIS WRITE COHERENTLY (AS DOUBLE WRITE)  */

        * (ulong *) &PRAM[PWMChan1][2] = ((ulong) pwmhi1 << 16L) | (ulong) pwmper1;
        HOSTSERVREQ(PWMChan1, 1);                           /*  issue  immediate  update
request */
        while (HOSTSERVSTAT(PWMChan1) & 3)          /* await reply */
                ;

    * (ulong *) &PRAM[PWMChan2][2] = ((ulong) pwmhi2 << 16L) | (ulong) pwmper2;
        HOSTSERVREQ(PWMChan2, 1);
        while (HOSTSERVSTAT(PWMChan2) & 3)
                ;

    * (ulong *) &PRAM[PWMChan3][2] = ((ulong) pwmhi3 << 16L) | (ulong) pwmper3;
        HOSTSERVREQ(PWMChan3, 1);
        while (HOSTSERVSTAT(PWMChan3) & 3)
                ;
```

```
    * (ulong *) &PRAM[PWMChan4][2] = ((ulong) pwmhi4 << 16L) | (ulong) pwmper4;
       HOSTSERVREQ(PWMChan4, 1);
       while (HOSTSERVSTAT(PWMChan4) & 3)
              ;

       }      /* TPUChangePWM() */



int  PWMcmds_from_uplink(short servo_cmd[]) {
//*********************************************************************
//
//  This is a function to decode an array of 8 short integers from serial
//  data collected in the Serial Port 1 buffer. The 8 short integers are
//  for use by the TPUChangePWM().
//
//*********************************************************************


    short   w_high, w_low;
    uchar    last_byte=0, curr_byte = 0;
    short   k;
    int       header_found = 0;
    int     i,j, NumBytes;

//  1. Specify default servo_cmd if first entry

    if (first_frame_a == 0)
    {   for (i=0; i<PWM_word_length; i++)
        {   last_servo_cmd[2*i] = (i+1)*1520;
            last_servo_cmd[2*i+1] =14250;
            first_frame_a = 1;
        }
    }




//     2. Will read all serial bytes that arrive by this time.
//     Leave bytes arriving later in the Serial Buffer


//  3. Search for Header

    if (SerByteAvail()!=0)
    {
        last_byte = SerGetByte();

        while (SerByteAvail() != 0)
        {   printf("\nSerByteAvail");
            curr_byte = SerGetByte();
            printf(" Current Byte = %x(x), %u(u)", curr_byte, curr_byte);

            if (last_byte == 255 && curr_byte == 255)
            { header_found = 1;
            break; }
            else {last_byte = curr_byte;}
    }


//  4.  If header was found, compile the servo commands into short int array
       if (header_found == 1)
    {   printf("\nHeader found");
        for ( i=0; i<PWM_word_length; i++)
        {
             w_low = (short) SerGetByte();
            w_high =  (short) SerGetByte();
            k = 256*w_high + w_low;
         last_servo_cmd[i] = k;
          SerInFlush();
```

```
            } /* end for loop */
    }


    } // endif NumBytes != 0

//   5.  Make servo_cmd[] equal latest if new cmds was received. Otherwise, last cmds
used.
    for (j=0;j<PWM_word_length;j++)
    {
        servo_cmd[j] = last_servo_cmd[j];
    }

    printf("\nServo Hi 1 = %u,  Servo Period 1 = %u",servo_cmd[0],servo_cmd[1]);
    return 5;


} /* PWMcmds_from_uplink */


int  PWMcmds_from_TPU3(short servo_cmd[]) {
//*********************************************************************
//
//  This is a function to decode an array of 8 short integers from serial
//  data collected in the Serial Port 1 buffer. The 8 short integers are
//  for use by the TPUChangePWM().
//
//*********************************************************************


    short   w_high, w_low, k;
    char    last_byte=0, curr_byte = 0;
    int        header_found = 0;
    int     i,j, NumBytes;

//  1. Specify default servo_cmd if first entry

    if (first_frame_a == 0)
    {   for (i=0; i<PWM_word_length; i++)
        {   last_servo_cmd[2*i] = (i+1)*1520;
            last_servo_cmd[2*i+1] =14250;
        }
    }


//     2. Will read all serial bytes that arrive by this time.
//     Leave bytes arriving later in the Serial Buffer

    NumBytes = TSerByteAvail(3);

    printf("\nPass Stage 2. NumBytes Avail = %d",NumBytes);


//  3. Search for Header

    if (NumBytes !=0) {     // do only if there are data in buffer, otherwise SerGetByte
                            // will wait till char is received in buffer

        last_byte = TSerGetByte(3);
        NumBytes -= 1;


       while (NumBytes != 0)
       {
           curr_byte = TSerGetByte(3);
           NumBytes -= 1;
          if (last_byte == Header_byte && curr_byte == Header_byte)
          { header_found = 1;
          break; }
        last_byte = curr_byte;
    }
```

```
//  4.  If header was found, compile the servo commands into short int array
       if (header_found == 1)
    {   for ( i=0; i<PWM_word_length; i++)
        {
            w_high = TSerGetByte(3);
            w_low =  TSerGetByte(3);
            k = 256*w_high + w_low;
         last_servo_cmd[i] = k;
               } /* end for loop */
    }


    } // endif NumBytes != 0

//   5.  Make servo_cmd[] equal latest if new cmds was received. Otherwise, last cmds
used.
    for (j=0;j<PWM_word_length;j++)
    {
        servo_cmd[j] = last_servo_cmd[j];
    }

    printf("\nServo Hi 1 = %u,  Servo Period 1 = %u",servo_cmd[0],servo_cmd[1]);
    return 5;


}                     /*                      PWMcmds_from_TPU3                      */
```

B. 3DMIO.H

```
//*********************************************************************
//
//  Filename:       3DMio.h
//  Description:    3DM Tattltetale Input and Output Routines.
//  Purpose:        Read and Send 3DM and AtoD data
//  Data:           13 May, 2001
//  Programmer:     LT Matt B. Commerford
//
//*********************************************************************

#include "data3DM.h"
#include "AtoDdata.h"


int SendAtoD (struct AtoDdata *AtoD) {
//*********************************************************************
//  This function reads all 8 A to D channels on the Tattletale and
//  outputs them on the primary serial port.
//*********************************************************************

   int i = 0;
   char *ptrTemp;
   ptrTemp = (char*)AtoD;       // Cast AtoDdata pointer to character so
                                // each individual byte can be addressed

   // Read A to D Converter Channels 0 - 7;
   AtoD->ch0 = AtoDReadWord(0);
   AtoD->ch1 = AtoDReadWord(1);
   AtoD->ch2 = AtoDReadWord(2);            // Note:  Hardware Rate Sensors for p & q are
wired backwards
   AtoD->ch3 = AtoDReadWord(3);               // Channel 2 and  3 are reversed again in
right order because of Jerry ask 07/30/01 -Vlad
//   AtoD->ch2 = AtoDReadWord(3);    // Note:  Hardware Rate Sensors for p & q are wired
backwards
//    AtoD->ch3 = AtoDReadWord(2);     // Channel 2 and 3 are reversed in software to
correct problem
   AtoD->ch4 = AtoDReadWord(4);
   AtoD->ch5 = AtoDReadWord(5);
   AtoD->ch6 = AtoDReadWord(6);
   AtoD->ch7 = AtoDReadWord(7);
```

```
   SerPutByte (255);            //  AtoD header byte (0xFF)
   SerPutByte (255);            //  AtoD header byte (0xFF)

   for (i = 0; i < 16; i++) {
      SerPutByte (ptrTemp[i]);  //  Output AtoD data
   }

   return 1;
}



int SendAtoDTest (struct AtoDdata *AtoD,short value) {
//*********************************************************************
//  This is a function to output test A to D data. The function accepts
//  an integer value as the least significant byte on each A to D channel
//  and outputs them on the primary serial port.  A to D
//  channels are actually read first to simulate timing requiremnts.
//  The primary purpose is for troubleshooting the output of the code.
//*********************************************************************

   int i = 0;
   char *ptrTemp;
   ptrTemp = (char*)AtoD;        // Cast AtoDdata pointer to character so
                                 // each individual byte can be addressed

   // Read A to D Converter Channels 0 - 7;
   AtoD->ch0 = AtoDReadWord(0);
   AtoD->ch1 = AtoDReadWord(1);
   AtoD->ch2 = AtoDReadWord(2);
   AtoD->ch3 = AtoDReadWord(3);
   AtoD->ch4 = AtoDReadWord(4);
   AtoD->ch5 = AtoDReadWord(5);
   AtoD->ch6 = AtoDReadWord(6);
   AtoD->ch7 = AtoDReadWord(7);

   // Test Values to ensure propper transmission.
   AtoD->ch0 = value;
   AtoD->ch1 = value;
   AtoD->ch2 = value;
   AtoD->ch3 = value;
   AtoD->ch4 = value;
   AtoD->ch5 = value;
   AtoD->ch6 = value;
   AtoD->ch7 = value;


   SerPutByte (255);         //  AtoD header byte (0xFF)
   SerPutByte (255);         //  AtoD header byte (0xFF)

   for (i = 0; i < 16; i++) {
      SerPutByte (ptrTemp[i]);   //  Output AtoD data.
   }

   return 1;
}


int Read3DM (struct data3DM* threeDM) {
//*********************************************************************
//  This is a function to receive 3dm data. The function first reads the
//  diagnostic byte.  65 indicates valid data.  If data is valid two
//  bytes are read for each of the 6 3DM channels (Hx, Hy, Hz, Ax, Ay, Az).
//  Input is on the secondary serial port (TPU 14)
//*********************************************************************

   int i = 0;
   unsigned int temp[12];

   //  Check diagnostic byte:  41h if valid.  6Xh if error where X is error code
```

```
   if (TSerByteAvail(14) >= 13) {
      if (TSerGetByte(14) == 65) {
         //  Read in the 12 bytes in the buffer from the 3DM
         for (i = 0; i < 12; i++) {
            temp[i] = TSerGetByte(14);
            // TSerInFlush(14);                    // Empty input Buffer
         }

         threeDM->Hx = (short)(256*temp[0] + temp[1]);    // MSB, LSB
         threeDM->Hy = (short)(256*temp[2] + temp[3]);    // MSB, LSB
         threeDM->Hz = (short)(256*temp[4] + temp[5]);    // MSB, LSB
         threeDM->Ax = (short)(256*temp[6] + temp[7]);    // MSB, LSB
         threeDM->Ay = (short)(256*temp[8] + temp[9]);    // MSB, LSB
         threeDM->Az = (short)(256*temp[10] + temp[11]);  // MSB, LSB

         return 1;               // Return after successful completion
      }
   }

   TSerInFlush(14);

   return 0;                     // Return after error code
}



int Send3DM (struct data3DM* threeDM) {
//*********************************************************************
//  This is a function to send 3dm data. The function outputs two
//  bytes for each of the 6 3DM channels (Hx, Hy, Hz, Ax, Ay, Az).
//  Output is on the primary serial port.
//*********************************************************************
   int i = 0;
   char *ptrTemp;
   ptrTemp = (char*)threeDM;     // Cast data3DM pointer to character pointer
                                 // so each individual byte can be addressed

   for (i = 0; i < 12; i++){
      SerPutByte(ptrTemp[i]);    // Output 12 bytes of 3DM data
   }

   return 1;
}



int Send3DMTest (short value) {
//*********************************************************************
//  This is a function to output test 3dm data. The function accepts
//  an integer value as the least significant byte on each of the
//  6 channels (Hx, Hy, Hz, Ax, Ay, Az) and outputs them on the
//  primary serial port. The primary purpose of the function is for
//  troubleshooting the output of the code.
//*********************************************************************

   int i = 0;

   // Test Values to ensure propper transmission.
   SerPutByte(00);
   SerPutByte(value);
   SerPutByte(00);
   SerPutByte(value);
   SerPutByte(00);
   SerPutByte(value);
   SerPutByte(00);
   SerPutByte(value);
   SerPutByte(00);
   SerPutByte(value);
   SerPutByte(00);
   SerPutByte(value);
```

```
   return 1;
}


int SendGPS () {
//*********************************************************************
//  This function reads in data on TPU line 1 from the GPS tattletale and
//  retransmitts it out the primary serial port.
//*********************************************************************

   int i = 0;

   if (TSerByteAvail(1) >= 310) { // Flush the buffer if over 1 second of data present
      TSerInFlush(1);
   }

   if (TSerByteAvail(1) >= 31) {  // Check for complete GPS message of 31 bytes
      for (i = 0; i < 31; i++ ) {
         SerPutByte(TSerGetByte(1));
      }
      SerPutByte(13);
      SerPutByte(10);

      return 1;                  // Return after successful transmission
   }

   return 0;                     // Return after not enough bytes available
}


int SendGPSAtoD () {
//*********************************************************************
//  This function reads in AtoD data on TPU line 2(it's supposed to be 8) from the GPS
tattletale and
//  retransmitts it out the primary serial port.
//*********************************************************************

   int i = 0;

   if (TSerByteAvail(8) >= 180) {  // Flush the buffer if over 1 second of data present
      TSerInFlush(8);
   }

   if (TSerByteAvail(8) >= 18) {  // Check for complete GPS AtoD message of 18 bytes
      for (i = 0; i < 18; i++ ) {
         SerPutByte(TSerGetByte(8));
      }
      SerPutByte(13);
      SerPutByte(10);

      return 1;                  // Return after successful transmission
   }

   return 0;                     // Return after not enough bytes available
}
```

C. DATA3DM.H

```
//*********************************************************************
//
//  Filename       data3DM.h
//  Description:   3DM Data Structure
//  Purpose:       Structure to group all 3DM data into single data type
//  Data:          13 May, 2001
//  Programmer:    LT Matt B. Commerford
//
//*********************************************************************

#if defined (__DATA3DM_H)
   //    Avoid multiple header inclusions
#else
```

```
#define __DATA3DM_H

struct data3DM {

   short Hx;         // two bytes for X magnetic vector
   short Hy;         // two bytes for Y magnetic vector
   short Hz;         // two bytes for Z magnetic vector
   short Ax;         // two bytes for X acceleration
   short Ay;         // two bytes for Y acceleration
   short Az;         // two bytes for Z acceleration

};
#endif
```

D. PWM.H

```
//*********************************************************************
//
//  Filename       PWM.h
//  Description:   PWM Command functions
//  Purpose:       Functions to set PWM output from TT8 TPU
//  Data:          Sep 2001
//  Programmer:    Bock Aeng Lim
//
//*********************************************************************



/* Definitions for putting PWM waveform onto TPU(4)-(7) */
#define        PWMChan1       4
#define        PWMChan2       5
#define        PWMChan3       6
#define        PWMChan4       7


/* PSC Pin State Control */
#define               ForceByPAC            0x00
#define               ForceHigh             0x01
#define               ForceLow              0x02
#define               NoForceState   0x03

/* PAC Pin Action Control (Inputs) */
#define               NoTranDet             0x00
#define               DetRising             0x04
#define               DetFalling            0x08
#define               DetEither             0x0c
#define               NoChangePAC           0x10

/* PAC Pin Action Control (Outputs) */
#define               NoChangematch  0x00
#define               HighOnMatch           0x04
#define               LowOnMatch            0x08
#define               ToggleOnMatch  0x0c

/* TBS Time Base/Directionality Control */
#define               InputChan             0x00
#define               OutputChan            0x80
#define               Cap1Match1            0x00
#define               Cap1Match2            0x20
#define               Cap2Match1            0x40
#define               Cap2Match2            0x60
#define               NoChangeTBS           0x100

/* Definitions for Decoding Serial Data from TPU(9) */
// #define     NULL            0;
#define     Header_byte     255
#define     PWM_byte_length 16
#define     PWM_word_length 8
#define     PWM_default_hi  1520;
#define     PWM_default_per 14250;
```

E. GPS TATTLETALE OPERATING CODE

```
/*-------------------------------------------------------------------------------*/
/*            GPS Tattletale Operating System                                    */
/*            Primary function: GPS Serial Data Parsing                          */
/*            Secondary function:  Timing of A/D collection and transmission     */
/*            April 23, 2001                                                     */
/*            Filename:      GPSTT.c                                             */
/*-------------------------------------------------------------------------------*/
#include <TT8.h>              /* Tattletale Model 8 Definitions */
#include <tt8lib.h>           /* for TT8 functions */
#include <tat332.h>           /* 68332 Hardware Definitions */
#include <tpu332.h>        /* 68332 Time Processing Unit Definitions */
#include <sim332.h>     /* 68332 System Integration Module Definitions */  // Do I need?
#include <qsm332.h>     /* 68332 Queued Serial Module Definitions */       // Do I need?
#include <dio332.h>     /* 69332 Digital I/O Port Pin Definitions */       // Do I need?
#include <stdlib.h>           /* for malloc */
#include <stdio.h>            /* for printf() */
#include <userio.h>           /* for prompts */
#include <string.h>           /* for sting comparison */
#include "GPSparse.h"  // for parseGGA, parseRMC,  & parseVTG functions
#include "GPSio.h"            // for printGPS and sendGPS functions
#include "GPSdata.h"   // for GPSdata structure definition
#include "AtoDdata.h"   // for AtoDdata sturcture definition

#define TSerInQBytes   TSerByteAvail  /* return number of bytes in input queue */
#define TSerOutQBytes  TSerByteAvail  /* return number of bytes in output queue */
//uchar                                          /* pointer to the start of the queue
data buffer */
//     TSerGetQueue(                        /* return low level queue information */
//             int           chan,                /* TPU channel, 0..15 */
//             int           *head,               /*  index  into  queue  for  first
character */
//             int           *tail,               /*  index  into  queue  for  last
character -1 */
//             int           *size);              /*  size  of  queue  for  wrap
manipulation */


#define SAMPLERATE  40               //  Main sample frequency 40 Hz
#define SLEEPCOUNT  40000/SAMPLERATE/2  //  Sleep count in of Hertz rate (SAMPLERATE)
#define QSIZE         8                                    //   Must  be  a  power  of  2
{2,4,8,16,32...}
#define TSBUFSIZE     128                       //  Must be 2^QSIZE

int parseGGA (struct GPSdata*);                        // parseGGA function prototype
int parseRMC (struct GPSdata*);                        // parseRMC function prototype
int parseVTG (struct GPSdata*);                 // parseVTG function prototype
int TprintGPS (const struct GPSdata* const);    // TprintGPS function prototype
int    TsendGPS  (const struct GPSdata* const);   // TsendGPS function prototype
int TsendAtoD (struct AtoDdata*);                // TsendAtoD function prototype
int TsendAtoDTest (struct AtoDdata*, short);    // TsendAtoDTest function prototype
int Tsend3dmTest (short value);                // Tsend3dmTest function prototype

main (){

 XmdmErr              xerr;              // Xmodem error code
 ulong baud;                          // baud rate
 ulong time = 0;                                // Current clock tick rate
 ulong sample = 0;                              // Timing Statistucs
 ulong runningTime = 0;              // Timing Statistics
 ulong missed = 0;                   // Timing Statistics
 float missedPercent = 0;            // Timing Statistics
 long *valuePtr;                                // Pointer to stored values
 char timeStr[40];                   // Timing Statistics
 short OnTime;                                  // Test for timeslice overrun
 short value = 48;                   // ASCII value for '0' --> to be used in testing
 int onTime = 1;                     // Timing Statistics
 int i = 0;
```

```
 int count = 0;
 int status;
 void *serBufPtr = NULL;
 void *inBufPtr = NULL;
 void *outBufPtr = NULL;
 struct  GPSdata  GPS  =  {11,22,33,4,55,66,777777,888,99,000000,1,22,33,44,5,66,7,88,9};

 struct GPSdata emptyGPS = {11,22,33,4,55,66,777777,888,99,000000,1,22,33,44,5,66,7,88,9}
;

 struct AtoDdata TTAtoD;


 InitTT8(NO_WATCHDOG,TT8_TPU);      // Initialize the Model 8


//  ---------------   Set up Memory Buffers for Serial Input and Output ----------------
--
 serBufPtr = malloc(4096);                                                    //     Console
(Primary) Serial Buffer
               if (serBufPtr == NULL) printf("\nBuffer Memory Allocation Failed\n");
               if (serBufPtr != NULL) printf("bufferPtr Memory O.K.\n");

 inBufPtr = malloc(TSBUFSIZE+TSER_MIN_MEM);         //  TPU(14) Serial In Buffer
        if (inBufPtr == NULL) printf("inBufPtr too big\n");
        if (inBufPtr != NULL) printf("inBufPtr Memory O.K.\n");

 outBufPtr = malloc(TSBUFSIZE+TSER_MIN_MEM);        //  TPU(13) Serial Output Buffer
        if (outBufPtr == NULL) printf("outBufPtr too big\n");
        if (outBufPtr != NULL) printf("outBufPtr Memory O.K.\n");


//  ---------------   Set up Serial Input and Output Parameters ------------------------
--
 printf("\nSet BAUD rate to 38400\n");
 printf("\nTick Rate is %ld\n",GetTickRate());
 printf("\nCurrent System Freq %ld\n",SimGetFSys());
 printf("\nSystem Clock is %ld\n",SimGetFSys());
 printf("\nBaud Rate is %ld\n",baud);
 PutStr("\nPreparing to Get GPS data\n");


 Sleep(0);                                        // Initialize timer
 Sleep(1200);                                     // Sleep 30 ms to allow screen printout

 SimSetFSys(14720000);
 baud = SerSetBaud(38400,0);
 SetTickRate(40000);
 SerSetInBuf (serBufPtr,4096);
 //TSerOpen(14,LowPrior,0,inBufPtr,TSBUFSIZE,57600,'N',8,1);       //  Open  Port  14  for
buffered input
  TSerOpen(13,HighPrior,1,outBufPtr,QSIZE,57600,'N',8,1);           //  Open  Port  13  for
buffered output


//  ---------------   Attempt to Sync with 1st byte of GPS signal ----------------------
--
 SerInFlush ();                                                      //  Empty  the  Serial
Buffer
 while (!SerByteAvail()) {};                   // Wait until serial byte available
 Sleep(0);                                    // Initialize timer
 Sleep(2000);                                 // Sleep 50 ms to allow the buffer to fill
 SerInFlush();                                // Empty Serial Buffer again to ensure next
                                              // string will be a complete one
 while (!SerByteAvail()) {};                   // Wait until serial byte available
 Sleep(0);                                    // Initialize timer
 Sleep(400);                                  // Sleep 10 ms to allow the buffer to fill



 do {
```

```
        //  ------------------------------------------------------------------------
----
        //                   1st Minor Timing Cycle @ 40Hz  -->  {send AtoD data}
        //  ------------------------------------------------------------------------
----
                                          // Note:  ** Indicates debugging code
     Sleep(0);                            // Initialize timer
        count = 0;                                // ** Initialize missed timing block
counter
     if (value >= 56) {value = 48;}       // ** Reset sample counter to ASCII value for
'0'
     if (sample == 100 ) {missed = 0;}    // ** Reset timing counter.  Measures steady
                                          // ** state errors after 100 initial samples.
     TPUSetPin (0,0);                                    // Clock signal low to 2nd
Tattletale
        TPUSetPin (1,0);                                        // Switch signal low
to Multiplex Switch
     TSerPutByte (13,255);                               // Header Byte (0xFF)
        TSerPutByte (13,255);                            // Header Byte (0xFF)

     TsendAtoDTest (&TTAtoD,value);       // 16 bytes of Test AtoD (ASCII '0'-'9')

        onTime = Sleep (600);                           // Sleep until 15 ms is over
(40 = 1 ms)
        if (!onTime) { count += 1; }             // ** Update missed timing slot
statistics

        //  ------------------------------------------------------------------------
---
     TPUSetPin (0,1);                                    // Clock signal high to 2nd
Tattletale
        TPUSetPin (1,1);                                        // Switch signal high
to Multiplex Switch
     Tsend3dmTest (103 - value);          // ** 12 bytes of Test 3dM (ASCII '9'-'0')

     sample += 1;                         // ** Increment sample counter for testing
     value += 1;                          // ** Increment test value counter

     onTime = Sleep (400);                               // Sleep until 10 ms is over
(40 = 1 ms)
        if (!onTime) { count += 1; }             // ** Update missed timing slot
statistics


        //  ------------------------------------------------------------------------
----
        //                   2nd Minor Timing Cycle @ 40Hz  -->  {Send AtoD data}
        //  ------------------------------------------------------------------------
----
        TPUSetPin (0,0);                                        //  Clock  signal  low
to 2nd Tattletale
        TPUSetPin (1,0);                                        // Switch signal low
to Multiplex Switch

     TsendAtoDTest (&TTAtoD,value);       // 16 bytes from AtoD (ASCII '0'-'9')

        onTime = Sleep (600);                        // Sleep until 15 ms is over (40
= 1 ms)
        if (!onTime) { count += 1; }          // Update missed timing slot statistics


        //  ------------------------------------------------------------------------
---
     TPUSetPin (0,1);                                    // Clock signal high to 2nd
Tattletale
        TPUSetPin (1,1);                                        // Switch signal high
to Multiplex Switch
     Tsend3dmTest (103 - value);          // ** Send 12 bytes from 3dM (ASCII '9'-'0')

     sample += 1;                         // ** Increment sample counter for testing
     value += 1;                          // ** Increment test value counter
```

```
     onTime = Sleep (400);                                  // Sleep until 10 ms is over
(40 = 1 ms)
         if (!onTime) { count += 1; }               // ** Update missed timing slot
statistics


         //  ------------------------------------------------------------------------
----
         //                   3rd Minor Timing Cycle @ 40Hz  -->  {Send AtoD data, parse
GGA & RMC string}
         //  ------------------------------------------------------------------------
----
         TPUSetPin (0,0);                                          //  Clock  signal  low
to 2nd Tattletale
         TPUSetPin (1,0);                                          // Switch signal low
to Multiplex Switch

     TsendAtoDTest (&TTAtoD,value);         // 16 bytes from AtoD (ASCII '0'-'9')

     StopWatchStart();
         parseGGA(&GPS);                        // Parse GGA line of GPS data
         parseRMC(&GPS);                        // Parse RMC line of GPS data
         time = StopWatchTime();

         onTime = Sleep (600);                              // Sleep until 15 ms is over
(40 = 1 ms)
         if (!onTime) { count += 1; }               // ** Update missed timing slot
statistics

     //  ------------------------------------------------------------------------
         TPUSetPin (0,1);                                          // Clock signal high
to 2nd Tattletale
         TPUSetPin (1,1);                                          // Switch signal high
to Multiplex Switch
     Tsend3dmTest (103 - value);            // ** Send 12 bytes from 3dM (ASCII '9'-'0')

     sample += 1;                           // ** Increment sample counter for testing
     value += 1;                            // ** Increment test value counter

     onTime = Sleep (400);                              // Sleep until 10 ms is over
(40 = 1 ms)
         if (!onTime) { count += 1; }               // ** Update missed timing slot
statistics


     //  ------------------------------------------------------------------------
     //               4th Minor Timing Cycle @ 40Hz   -->  {Send AtoD data, send GPS
data}
     //  ------------------------------------------------------------------------
         TPUSetPin (0,0);                                          //  Clock  signal  low
to 2nd Tattletale
         TPUSetPin (1,0);                                          // Switch signal low
to Multiplex Switch

     TsendAtoDTest (&TTAtoD,value);         // 16 bytes from AtoD (ASCII '0'-'9')
     TsendGPS (&GPS);                      // Send GPS data out the secondary port (TPU
13)
     // Reset GPS value for troubleshooting.  This helps determine where data drops out.
     GPS = emptyGPS;
     onTime = Sleep (600);                              // Sleep until 15 ms is over
(40 = 1 ms)
         if (!onTime) { count += 1; }               // ** Update missed timing slot
statistics



    //  ------------------------------------------------------------------------
         TPUSetPin (0,1);                                          // Clock signal high
to 2nd Tattletale
```

```
        TPUSetPin (1,1);                                          // Switch signal high
to Multiplex Switch
     Tsend3dmTest (103 - value);            // Send 12 bytes from 3dM (ASCII '9'-'0')

     sample += 1;                           // ** Increment sample counter for testing
     value += 1;                            // ** Increment test value counter

//   ** Print out timing diagnostics.
     sprintf(timeStr," S%4.d T%4.d",SerByteAvail(),TSerByteAvail(13));
        for (i = 0;i <12; i++){
               TSerPutByte (13, ((int)timeStr[i]));
        }
        TSerPutByte(13,13);                         //  Carriage Return & Line Feed
        TSerPutByte(13,10);

     missed += count;
        runningTime += time;

     onTime = Sleep (400);                                        // Sleep until 10 ms
is over (40 = 1 ms)
        if (!onTime) { count += 1; }

 } while(sample < 3000);     //!TSerByteAvail(14));


 //  ---------------  Output Timing Statistics and Download Options --------------------
----

 missedPercent = (float)missed/(float)sample*100;
 sprintf(timeStr,"\n# Samples: %ld, # Misses: %ld,  Percent Misses: ,%5.3f, Ave Parse
Time: %5.3f, \n",
 sample, missed, missedPercent,(float)runningTime/(float)sample);
 for (i = 0;i <80; i++){
        TSerPutByte (13, ((int)timeStr[i]));
 }
     TSerPutByte (13,13);                                         // Carriage Return &
Line Feed
        TSerPutByte (13,10);

 printf("\nFinished with GPS data...");
 SerInFlush();
 SerSetBaud(9600,0);
 ResetToMon();
 return(0);
}
```

F. GPSPARSE.H

```
//*********************************************************************
//
//     Filename:             GPSparse.h
//     Description:   GPS $GGGGA Parsing Routine.
//     Purpose:              Routine to parse the GGA line of the GPS data
//     Data:                 21 April, 2001
//     Programmer:           LT Matt B. Commerford
//
//*********************************************************************

#include "GPSdata.h"

int parseGGA (struct GPSdata *ptrGPS) {

       int success = 1;
       char temp[100];
       int i = 0;
       int j = 0;
       int gamma = 0 - (int)'0';     //  Correction  factor  between  ASCII  and  binary
values;
```

```
        // ------------------------ Get GPS String Header --------------------------
        while (SerGetByte() != '$') {}        // Find start of a string $
        j = 0;
        do {
                temp[j] = SerGetByte();
        } while (temp[j++] != ',');
        temp[j] = '\0';                                     // NULL terminate the string

    if (!strcmp(temp,"GPGGA")) {             // Procces GGA line else return
         //  Print out error message.
         sprintf(temp,"\nReturn from parseGGA\n");
         for (i = 0;i <12; i++){
             TSerPutByte (13, ((int)temp[i]));
         }
             TSerPutByte(13,13);                 //  Carriage Return & Line Feed
             TSerPutByte(13,10);
         return (0);
    }



        // ------------------------ Get GPS Time stamp -----------------------------
        j = 0;
        do {
                temp[j] = SerGetByte();
        } while (temp[j++] != ',');

        ptrGPS->timeHH = (char)(10*((int)temp[0] + gamma) + ((int)temp[1]+gamma));   //
Hours
        ptrGPS->timeMM = (char)(10*((int)temp[2] + gamma) + ((int)temp[3]+gamma));   //
Minutes
        ptrGPS->timeSS = (char)(10*((int)temp[4] + gamma) + ((int)temp[5]+gamma));   //
Seconds
        ptrGPS->timeDecimalSS = (char)(1*((int)temp[7]+gamma));
                    // Decimal Seconds


        // ------------------------ Get Latitude -------------------------------------
        j = 0;
        do {
                temp[j] = SerGetByte();
        } while (temp[j++] != ',');
        if (j <= 12 ) {
                temp[j++] = '0';      // Pad trailing zeros in array to ensure 7 decimal
places
        };
        temp[j] = '\0';                       // NULL terminate string

        ptrGPS->latDeg = (char)(10*((int)temp[0] + gamma) + ((int)temp[1]+gamma));    //
Lat(deg)
        ptrGPS->latMin = (char)(10*((int)temp[2] + gamma) + ((int)temp[3]+gamma));    //
Lat(min)
        ptrGPS->latDecimalMin = atol(&temp[5]);                                       //
Converts argument to type long


        // ------------------------ Get N/S ----------------------------------------
        j = 0;
        do {
                temp[j] = SerGetByte();
        } while (temp[j++] != ',');


        // ------------------------ Get Longitude ------------------------------------
        j = 0;
        do {
                temp[j] = SerGetByte();
        } while (temp[j++] != ',');
        if (j <= 12 ) {
                temp[j++] = '0';      // Pad trailing zeros in array to ensure 7 decimal
places
```

```
        };
        temp[j] = '\0';                         // NULL terminate string

        ptrGPS->longDeg = (char)(100*((int)temp[0] + gamma) + 10*((int)temp[1]+gamma) +
                ((int)temp[2] + gamma));
                                                // Lat(deg)
        ptrGPS->longMin = (char)(10*((int)temp[3] + gamma) + ((int)temp[4]+gamma));    //
Lat(min)
        ptrGPS->longDecimalMin = atol(&temp[6]);                                        //
Converts argument to type long



        // ------------------------ Get E/W ----------------------------------------
        j = 0;
        do {
                temp[j] = SerGetByte();
        } while (temp[j++] != ',');


        // ------------------------ Get DGPS status byte -----------------------------
        j = 0;
        do {
                temp[j] = SerGetByte();
        } while (temp[j++] != ',');
        ptrGPS->diffGPS = (char)((int)temp[0] + gamma);


        while (SerGetByte() != ',') {};       // Discard Number SV
        while (SerGetByte() != ',') {};       // Discard HDOP


        // ------------------------ Get Antenna Height -------------------------------
        j = 0;
        do {
                temp[j] = SerGetByte();
        } while (temp[j++] != '.');
        temp[j] = '\0';                 // NULL terminate the string
        ptrGPS->altFeet = atoi(temp);

        j = 0;
        do {
                temp[j] = SerGetByte();
        } while (temp[j++] != ',');
        temp[j] = '\0';                 // NULL terminate the string
        ptrGPS->altDecimalFeet = (char)atoi(temp);

        while (SerGetByte() != ',') {};       // Discard 'M' units
        while (SerGetByte() != ',') {};       // Discard Age of DGPS
        while (SerGetByte() != '*') {};       // Discard Base Station ID


        return success;
};



//**********************************************************************
//
//      Description:   GPS $GGRMC Parsing Routine.
//      Purpose:               Routine to parse the RMC line of the GPS data
//      Data:                  21 April, 2001
//      Programmer:            LT Matt B. Commerford
//
//**********************************************************************

int parseRMC (struct GPSdata *ptrGPS) {

        int success = 1;
        char temp[100];
        int i = 0;
```

```
	int j = 0;
	int gamma = 0 - (int)'0';		// Correction for difference between Binary
& Ascii

	while (SerGetByte() != '$') {};	// Ensure start of a string $
	j = 0;
	do {
		temp[j] = SerGetByte();
	} while (temp[j++] != ',');
	temp[j] = '\0';			// NULL terminate the string
	if (strcmp(temp,"GPRMC")) {	// Procces RMC line else return

	//	while (SerGetByte() != ',') {};	// Discard GPRC header
		while (SerGetByte() != ',') {};	// Discard Time
		while (SerGetByte() != ',') {};	// Discard Status
		while (SerGetByte() != ',') {};	// Discard Latitude
		while (SerGetByte() != ',') {};	// Discard Latitude N/S
		while (SerGetByte() != ',') {};	// Discard Longitude
		while (SerGetByte() != ',') {};	// Discard Longitude E/W


		// ------------------------ Get Ground Speed Line ------------------------
-----
		j = 0;
		do {
			temp[j] = SerGetByte();
		} while (temp[j++] != '.');
		temp[j] = '\0';			// NULL terminate the string
		ptrGPS->grndSpeed = atoi(temp);

		j = 0;
		do {
			temp[j] = SerGetByte();
		} while (temp[j++] != ',');
		temp[j] = '\0';			// NULL terminate the string
		ptrGPS->grndSpeedDecimal = (char)atoi(temp);


		// ------------------------ Get Ground Track Line ------------------------
-------
		j = 0;
		do {
			temp[j] = SerGetByte();
		} while (temp[j++] != '.');
		temp[j] = '\0';			// NULL terminate the string
		ptrGPS->grndTrack = atoi(temp);

		j = 0;
		do {
			temp[j] = SerGetByte();
		} while (temp[j++] != ',');
		temp[j] = '\0';			// NULL terminate the string
		ptrGPS->grndTrackDecimal = (char)atoi(temp);


		while (SerGetByte() != ',') {};		// Discard Date Line


		// ------------------------ Get Magnetic Variation Line ------------------
--------
		j = 0;
		do {
			temp[j] = SerGetByte();
		} while (temp[j++] != '.');
		temp[j] = '\0';			// NULL terminate the string
		ptrGPS->magVar = atoi(temp);

		j = 0;
		do {
			temp[j] = SerGetByte();
		} while (temp[j++] != ',');
```

```
                temp[j] = '\0';              // NULL terminate the string
                ptrGPS->magVarDecimal = (char)atoi(temp);
        }

 //   SerInFlush ();                 // Empty the Serial Buffer

        return success;
}



//*********************************************************************
//
//      Description:   GPS $GGRMC Parsing Routine.
//      Purpose:               Routine to parse the RMC line of the GPS data
//      Data:                  21 April, 2001
//      Programmer:            LT Matt B. Commerford
//
//*********************************************************************

int parseVTG (struct GPSdata *ptrGPS) {

        int success = 1;
        char temp[100];
        int i = 0;
        int j = 0;
        int gamma = 0 - (int)'0';            // Correction for difference between Binary
& Ascii

        while (SerGetByte() != '$') {};      // Ensure start of a string $
        j = 0;
        do {
                temp[j] = SerGetByte();
        } while (temp[j++] != ',');
        temp[j] = '\0';                // NULL terminate the string
        if (strcmp(temp,"GPVTG")) {   // Procces VTG line else return

        //      while (SerGetByte() != ',') {};      // Discard GPRC header
        //      while (SerGetByte() != ',') {};      // Discard Time
        //      while (SerGetByte() != ',') {};      // Discard Status
        //      while (SerGetByte() != ',') {};      // Discard Latitude
        //      while (SerGetByte() != ',') {};      // Discard Latitude N/S
        //      while (SerGetByte() != ',') {};      // Discard Longitude
        //      while (SerGetByte() != ',') {};      // Discard Longitude E/W


                // ----------------------- Get Ground Track Line -----------------------
-------
                j = 0;
                do {
                        temp[j] = SerGetByte();
                } while (temp[j++] != '.');
                temp[j] = '\0';              // NULL terminate the string
                ptrGPS->grndTrack = atoi(temp);

                j = 0;
                do {
                        temp[j] = SerGetByte();
                } while (temp[j++] != ',');
                temp[j] = '\0';              // NULL terminate the string
                ptrGPS->grndTrackDecimal = (char)atoi(temp);

        while (SerGetByte() != ',') {};      // Discard "T" for True North
        while (SerGetByte() != ',') {};      // Discard ","
        while (SerGetByte() != ',') {};      // Discard ","


        // ----------------------- Get Ground Speed Line ----------------------------
                j = 0;
                do {
```

```
                temp[j] = SerGetByte();
            } while (temp[j++] != '.');
            temp[j] = '\0';              // NULL terminate the string
            ptrGPS->grndSpeed = atoi(temp);

            j = 0;
            do {
                temp[j] = SerGetByte();
            } while (temp[j++] != ',');
            temp[j] = '\0';              // NULL terminate the string
            ptrGPS->grndSpeedDecimal = (char)atoi(temp);


         // Fake Magnetic Variation Value for compatibility purposes
         ptrGPS->magVar = 15;
         ptrGPS->magVarDecimal = 3;

                while (SerGetByte() != ',') {};             // Discard Date Line
         while (SerGetByte() != ',') {};          // Discard ","


        }

//   SerInFlush ();                 // Empty the Serial Buffer

        return success;
}
```

```
//*********************************************************************
//
//     Filename:             GPSio.h
//     Description:   GPS Input and Output Routines.
//     Purpose:              Print GPS structure data as ASCII or Binary data.
//     Data:                 28 April, 2001
//     Programmer:           LT Matt B. Commerford
//
//*********************************************************************

#include "GPSdata.h"
#include "AtoDdata.h"


int TprintGPS (const struct GPSdata* const ptrGPS) {
//*********************************************************************
//  This function converts the GPSdata structure to an ASCII string using
//  the sprintf function.  The string is output to the secondary serial
//  port (TPU 13).  The primary use is to provide a readable output to
//  be used in troubleshooting the code
//*********************************************************************


    char gpsLine[200];
        char gpsLine2[80];
        int i = 0;
        sprintf (gpsLine,"%d:%d:%d.%1.d  %d %d.%ld  %d %d.%ld  %d  %d.%d  ",
                ptrGPS->timeHH,        ptrGPS->timeMM,        ptrGPS->timeSS,        ptrGPS-
>timeDecimalSS, ptrGPS->latDeg,
                ptrGPS->latMin, ptrGPS->latDecimalMin, ptrGPS->longDeg, ptrGPS->longMin,
                ptrGPS->longDecimalMin,        ptrGPS->diffGPS,   ptrGPS->altFeet,   ptrGPS-
>altDecimalFeet);
        sprintf (gpsLine2,"%d.%d   %d.%d   %d.%.d   %d.%.d\n\0",ptrGPS->grndSpeed, ptrGPS-
>grndSpeedDecimal,
                ptrGPS->grndTrack,   ptrGPS->grndTrackDecimal,   ptrGPS->magVar,   ptrGPS-
>magVarDecimal);
        strncat (gpsLine,gpsLine2,30);
        for (i = 0;i <66; i++){
                TSerPutByte (13, ((int)gpsLine[i]));
        }
        TSerPutByte(13,13);
        TSerPutByte(13,10);

        return 1;
}



int TsendGPS (const struct GPSdata* const ptrGPS) {
//*********************************************************************
//  This function sends the GPSdata structure in binary format to the
//  secondary serial port (TPU 13).  This is the primary GPS output routine.
//  The function accepts a pointer to a character array.  In calling this
//  function the GPSdata structure pointer must be cast to a character
//  pointer so the bytes in the array can be referrenced individually.
//  The data stored in the GPSdata structure is not necessarily stored
//  continuously.  There are gaps in the data because variables in the
//  structure must be stored on 16 bit word boundaries in the Motorolla
//  chip.  Thus, there are gaps of meaningless data that are not output
//  from the program {17, 21, 25, 29}.
//*********************************************************************

        char *ptrTemp;
        ptrTemp = (char*)ptrGPS;    // Cast AtoDdata pointer to character so
                                  // each individual byte can be addressed


//              TSerPutByte (13,255);                                   // Header Byte
//              TSerPutByte (13,255);                                   // Header Byte

                TSerPutByte(13,(int)ptrTemp[0]);             // GPS.timeHH
```

```
              TSerPutByte(13,(int)ptrTemp[1]);            // GPS.timeMM
              TSerPutByte(13,(int)ptrTemp[2]);            // GPS.timeSS
              TSerPutByte(13,(int)ptrTemp[3]);            // GPS.timeDecimalSS

              TSerPutByte(13,(int)ptrTemp[4]);            // GPS.latDeg
              TSerPutByte(13,(int)ptrTemp[5]);            // GPS.latMin
              TSerPutByte(13,(int)ptrTemp[6]);            // GPS.latDecimalMin
              TSerPutByte(13,(int)ptrTemp[7]);
              TSerPutByte(13,(int)ptrTemp[8]);
              TSerPutByte(13,(int)ptrTemp[9]);

              TSerPutByte(13,(int)ptrTemp[10]);           // GPS.longDeg
              TSerPutByte(13,(int)ptrTemp[11]);           // GPS.longMin
              TSerPutByte(13,(int)ptrTemp[12]);           // GPS.longDecimalMin
              TSerPutByte(13,(int)ptrTemp[13]);
              TSerPutByte(13,(int)ptrTemp[14]);
              TSerPutByte(13,(int)ptrTemp[15]);

              TSerPutByte(13,(int)ptrTemp[16]);           // GPS.diffGPS

              TSerPutByte(13,(int)ptrTemp[18]);           // GPS.altFeet
              TSerPutByte(13,(int)ptrTemp[19]);
              TSerPutByte(13,(int)ptrTemp[20]);           // GPS.altDecimalFeet

              TSerPutByte(13,(int)ptrTemp[22]);           // GPS.grndSpeed
              TSerPutByte(13,(int)ptrTemp[23]);
              TSerPutByte(13,(int)ptrTemp[24]);           // GPS.grndSpeedDecimal

              TSerPutByte(13,(int)ptrTemp[26]);           // GPS.grndTrack
              TSerPutByte(13,(int)ptrTemp[27]);
              TSerPutByte(13,(int)ptrTemp[28]);           // GPS.grndTrackDecimal

              TSerPutByte(13,(int)ptrTemp[30]);           // GPS.magVar
              TSerPutByte(13,(int)ptrTemp[31]);
              TSerPutByte(13,(int)ptrTemp[32]);           // GPS.magVarDecimal

//            TSerPutByte(13,13);
//            TSerPutByte(13,10);

              return 1;

};



int TsendAtoD (struct AtoDdata *TTAtoD) {
//*********************************************************************
//  This function reads all 7 A to D channels on the Tattletale and
//  outputs them on the secondary serial port (TPU 13).
//*********************************************************************

    int i = 0;
       char *ptrTemp;
       ptrTemp = (char*)TTAtoD;    // Cast AtoDdata pointer to character so
                                   // each individual byte can be addressed


    // Read A to D Converter Channels 0 - 7;
    TTAtoD->ch0 = AtoDReadMilliVolts(0);
       TTAtoD->ch1 = AtoDReadMilliVolts(1);
       TTAtoD->ch2 = AtoDReadMilliVolts(2);
       TTAtoD->ch3 = AtoDReadMilliVolts(3);
       TTAtoD->ch4 = AtoDReadMilliVolts(4);
       TTAtoD->ch5 = AtoDReadMilliVolts(5);
       TTAtoD->ch6 = AtoDReadMilliVolts(6);
       TTAtoD->ch7 = AtoDReadMilliVolts(7);

//     TSerPutByte(13,238);        //  AtoD header byte (0xEE)
//     TSerPutByte(13,238);        //  AtoD header byte (0xEE)
     for (i = 0; i < 16; i++) {
         TSerPutByte (13,ptrTemp[i]);
```

```
     }
//     TSerPutByte(13,13);         //  Carriage Return & Line Feed
//     TSerPutByte(13,10);

return 1;
}



int TsendAtoDTest (struct AtoDdata *TTAtoD,short value) {
//*********************************************************************
//  This is a function to output test A to D data. The function accepts
//  an integer value as the least significant byte on each A to D channel
//  and outputs them on the secondary serial port (TPU 13).  A to D
//  channels are actually read first to simulate timing requiremnts.
//  The primary purpose is for troubleshooting the output of the code.
//*********************************************************************

    int i = 0;
       char *ptrTemp;
       ptrTemp = (char*)TTAtoD;    // Cast AtoDdata pointer to character so
                                  // each individual byte can be addressed

    // Read A to D Converter Channels 0 - 7;
    TTAtoD->ch0 = AtoDReadMilliVolts(0);
       TTAtoD->ch1 = AtoDReadMilliVolts(1);
       TTAtoD->ch2 = AtoDReadMilliVolts(2);
       TTAtoD->ch3 = AtoDReadMilliVolts(3);
       TTAtoD->ch4 = AtoDReadMilliVolts(4);
       TTAtoD->ch5 = AtoDReadMilliVolts(5);
       TTAtoD->ch6 = AtoDReadMilliVolts(6);
       TTAtoD->ch7 = AtoDReadMilliVolts(7);

    //  Test Values to ensure propper transmission.
 //   value = 0;
    TTAtoD->ch0 = value;
    TTAtoD->ch1 = value;
    TTAtoD->ch2 = value;
    TTAtoD->ch3 = value;
    TTAtoD->ch4 = value;
    TTAtoD->ch5 = value;
    TTAtoD->ch6 = value;
    TTAtoD->ch7 = value;


//     TSerPutByte(13,238);    //  AtoD header byte (0xEE)
//     TSerPutByte(13,238);    //  AtoD header byte (0xEE)

     for (i = 0; i < 16; i++) {
         TSerPutByte (13,ptrTemp[i]);
     }

//     TSerPutByte(13,13);     //  Carriage Return & Line Feed
//     TSerPutByte(13,10);

return 1;
}



int Tsend3dmTest (short value) {
//*********************************************************************
//  This is a function to output test 3dm data. The function accepts
//  an integer value as the least significant byte on each of the
//  6 channels (Hx, Hy, Hz, Ax, Ay, Az) and outputs them on the
//  secondary serial port (TPU 13). The primary purpose is for
//  troubleshooting the output of the code.
//*********************************************************************

    int i = 0;
```

```
    //  Test Values to ensure propper transmission.
       TSerPutByte(13,00);
       TSerPutByte(13,value);
       TSerPutByte(13,00);
       TSerPutByte(13,value);
       TSerPutByte(13,00);
       TSerPutByte(13,value);
       TSerPutByte(13,00);
       TSerPutByte(13,value + 10);
       TSerPutByte(13,00);
       TSerPutByte(13,value + 10);
       TSerPutByte(13,00);
       TSerPutByte(13,value + 10);

return 1;
}
```

G. GPSDATA.H

```
//     File Name:     GPSdata.h
//    Description:    GPS Data Structure
//    Purpose:        Structure to group all GPS data into a single data type
//    Data:            21 April, 2001
//    Programmer:        LT Matt B. Commerford
//
//**********************************************************************

#if defined (__GPSDATA_H)
    //        Avoid multiple header inclusions
#else
#define __GPSDATA_H

struct GPSdata {

//  Note:   One byte data represented with CHAR data type
//          Two byte data represented with INT data type
//         Four byte data represented with LONG data type
//          No floating point numbers used.  Conversions are
//          done after data is transmitted.

    char  timeHH;            // one byte of Hours
    char  timeMM;            // one byte of Minutes
    char  timeSS;            // one byte of Seconds
    char  timeDecimalSS;     // one byte of tenths of Seconds

    char  latDeg;            // one byte of degrees Latitude
    char  latMin;            // one byte of minutes Latitude
    long  latDecimalMin;     // four bytes of decimal mintues Latitude

    char  longDeg;           // one byte of degrees Longitude
    char  longMin;           // one byte of minutes Longitude
    long  longDecimalMin;    // four bytes of decimal mintues Longitude

    char  diffGPS;           // one byte of DGPS status

    short altFeet;           // two bytes of altitude in feet
    char  altDecimalFeet;    // one byte of decimal feet

    short grndSpeed;         // two bytes of groundspeed in ft/s
    char  grndSpeedDecimal;  // one byte of groundspeed in decimal ft/s

    short grndTrack;         // two bytes of ground track in degrees (0-359)
    char  grndTrackDecimal;  // one byte of ground track in decimal degrees

    short magVar;            // two bytes of magnetic variation in degrees
    char  magVarDecimal;     // one byte of mag variation in decimal degrees

//    short     gpsCRC;      // two byte Cyclic Reduncancy Check (CRC);
};
#endif
```

H. ATODDDATA.H

```
//*********************************************************************
//
//  Filename   :    AtoDdata.h
//      Description:   AtoD Data Structure
//      Purpose:              Structure to group all AtoD data into a single data type
//      Data:                 29 April, 2001
//      Programmer:           LT Matt B. Commerford
//
//*********************************************************************

#if defined (__ATODDATA_H)
       //             Avoid multiple header inclusions
#else
#define __ATODDATA_H

struct AtoDdata {

       short ch0;                    // two bytes for A to D channel (0)
       short ch1;                    // two bytes for A to D channel (1)
       short ch2;                    // two bytes for A to D channel (2)
       short ch3;                    // two bytes for A to D channel (3)
       short ch4;                    // two bytes for A to D channel (4)
       short ch5;                    // two bytes for A to D channel (5)
       short ch6;                    // two bytes for A to D channel (6)
       short ch7;                    // two bytes for A to D channel (7)

//     short    gpsCRC;       // two byte Cyclic Reduncancy Check (CRC);
};
#endif
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. CROSSBOW AHRS NOISE ANALYSIS

The CrossBow AHRS was evaluated to determine the amount and nature of the sensor noise when operating in the continuous/angle mode. The AHRS was initialized while sitting on a large stable concrete pad. Sufficient time was given for the Kalman filter to converge and data was collected using the Gyroview version 2.1. The data was then imported into Matlab where a statistical analysis was performed. The resulting Matlab output follows.

```
Crossbow Static Noise Test
------------------------------------------------------------
------------------------------------------------------------

Angular Rates
Mean Rates:    p = 0.0060913   q = 0.0059416   r = 0.0032076
Standard Devs: p = 0.087604    q = 0.085688    r = 0.07939

Covariance of Rates: p,q,r =

    0.0076745   -0.0002478    0.00020486
   -0.0002478    0.0073424   -7.9876e-005
    0.00020486  -7.9876e-005  0.0063027

Orientation Angle
Mean Angle:        roll = 0.9201   pitch = 0.038009   yaw = -78.9903
Standard Devs:     roll = 0.021981 pitch = 0.019398   yaw =   0.024684
Covariance of Angle: roll, pitch, yaw =

   0.00048315     8.8618e-006   2.5846e-005
   8.8618e-006    0.0003763     9.3887e-005
   2.5846e-005    9.3887e-005   0.00060928

Linear Acceleration
Mean Accel:     Ax = -0.00066671  Ay = 0.016047   Az = 0.99781
Standard Devs:  Ax =  0.00037249  Ay = 0.00045495 Az = 0.00032578
Covariance of Accel: Ax, Ay, Az =

  1.3875e-007  -5.3833e-009   1.1636e-009
 -5.3833e-009   2.0698e-007  -8.1784e-009
  1.1636e-009  -8.1784e-009   1.0613e-007
```

Magnetic Flux

Mean Flux: Hx = 0.045627 Hy = 0.24452 Hz = 0.48059
Standard Devs: Hx = 0.00083978 Hy = 0.00085417 Hz = 0.0010617
Covariance of Flux: Hx, Hy, Hz =

7.0523e-007	-2.1908e-008	3.7182e-007
-2.1908e-008	7.2961e-007	-1.8335e-007
3.7182e-007	-1.8335e-007	1.1272e-006

LIST OF REFERENCES

1. Futaba Corporation of America, "Futaba® PCM1024ZA Instruction & Operation Manual," Irvine, CA.

2. Commerford, M., "FROG UAV Inertial Measurement Unit Software Documentation," Unpublished Paper, June 2001.

3. Hallberg, E., "On Integrated Plant, Control and Guidance Design," Dissertation, Naval Postgraduate School, Monterey, CA, September 1997.

4. Papageorgio, E., "Development of a Dynamic Model for a UAV," Master's Thesis, Naval Postgraduate School, Monterey, CA, March 1997.

5. Pollard, S., "Development and Verification of an Aerodynamic Model for the NPS FROG UAV using the CMARC Panel Code Software Suite," Aeronautical Engineer's Thesis, Naval Postgraduate School, Monterey, CA, September 1998.

6. Integrated Systems, "RealSim User's Guide, Sunnyvale, CA, February 1999.

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. Doctor Isaac I. Kaminer, Dept. of Aeronautics and Astronautics
   Naval Postgraduate School
   Monterey, California

4. Oleg Yakimenko, Dept. of Aeronautics and Astronautics
   Naval Postgraduate School
   Monterey, California

5. Max Platzer, Dept. of Aeronautics and Astronautics
   Naval Postgraduate School
   Monterey, California